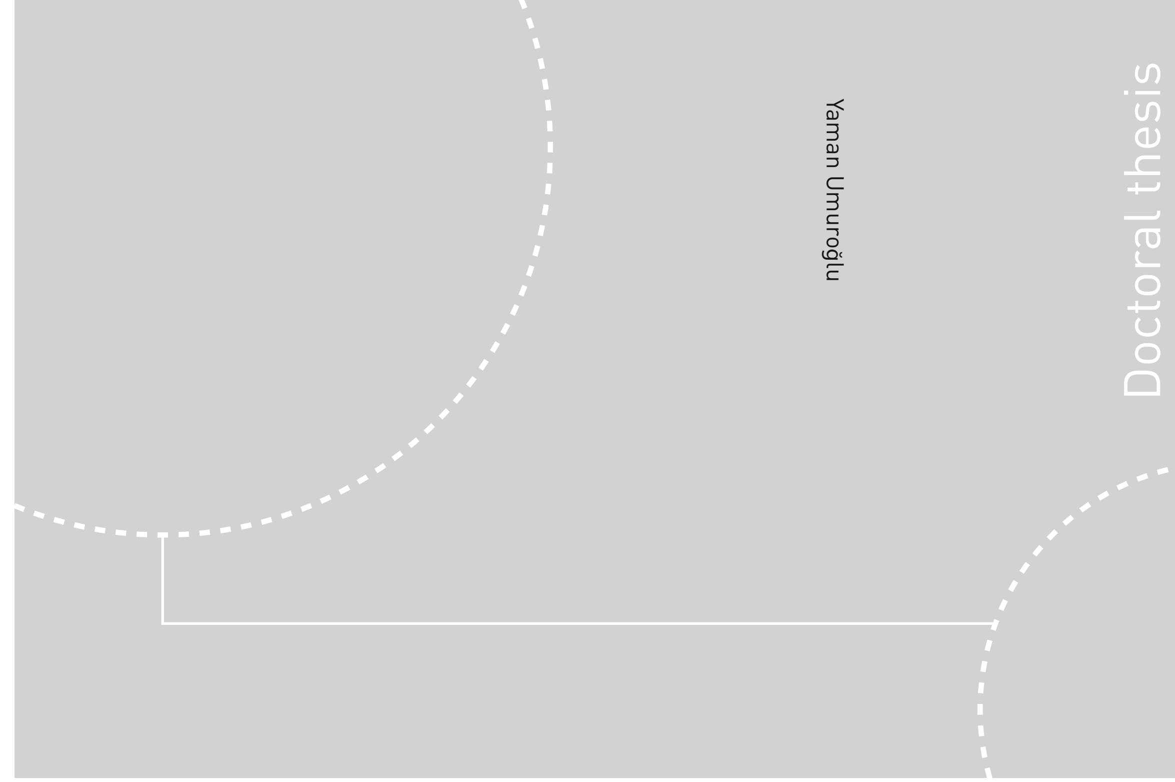
Doctoral theses at NTNU, 2018:1

Yaman Umuroğlu

Accelerating Sparse Linear Algebra and Deep Neural Networks on Reconfigurable Platforms



ISBN 978-82-326-2776-9 (printed ver.)
ISBN 978-82-326-2777-6 (electronic ver.)
ISSN 1503-8181

Doctoral theses at NTNU, 2018:1

NTNU
Norwegian University of Science and Technology
Thesis for the Degree of Philosophiae Doctor
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Yaman Umuroğlu

Accelerating Sparse Linear Algebra and Deep Neural Networks on Reconfigurable Platforms

Thesis for the Degree of Philosophiae Doctor

Trondheim, April 2018

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Norwegian University of
Science and Technology

NTNU
Norwegian University of Science and Technology

Thesis for the Degree of Philosophiae Doctor

Faculty of Information Technology and Electrical Engineering
Department of Computer Science

© Yaman Umuroğlu

ISBN 978-82-326-2776-9 (printed ver.)
ISBN 978-82-326-2777-6 (electronic ver.)
ISSN 1503-8181

Doctoral theses at NTNU, 2018:1

Printed by NTNU Grafisk senter

Abstract

Regardless of whether the chosen figure of merit is execution time, throughput, battery life for an embedded system or total cost of ownership for a datacenter, today's computers are fundamentally limited by their energy efficiency. Using specialized hardware-software solutions for particular applications or domains is a well-known approach to increase energy efficiency of computing systems. Reconfigurable logic in the form of Field-Programmable Gate Arrays (FPGAs) is a particularly promising substrate for hardware specialization, owing to its runtime reconfigurability, vastly parallel compute fabric and widespread availability. However, mapping computation to reconfigurable logic in a way which provides performance and efficiency benefits is a significant challenge due to the vast design space. In this thesis, we study how two particular domains can benefit from specialized architectures on reconfigurable logic. We focus on sparse linear algebra and deep neural network inference, whose execution is known to be particularly problematic on today's general-purpose computers.

For sparse linear algebra, lack of spatial and temporal locality in memory accesses pose a fundamental problem. We address this problem by taking advantage of the flexibility of reconfigurable logic to construct specialized memory systems. We propose a hardware-software caching scheme which uses lightweight preprocessing to extract key access pattern information from sparse matrices to offer greatly increased random access efficiency with minimal on-chip memory usage. Furthermore, we demonstrate the broader applicability of the specialization for sparse linear algebra to graph analytics with an accelerator for breadth-first search that uses off-chip memory bandwidth more efficiently compared to prior work.

For deep neural network inference, the sheer energy and hardware resource cost of floating point computation is a fundamental limitation on energy efficiency. Exploiting recent advances in training highly quantized neural networks (QNNs), we demonstrate how FPGAs can be leveraged for accurate, energy-efficient and high-performance neural network inference. We propose the FINN framework to generate customized architectures with compute resources tailored to user-specified performance requirements while exploiting multiple levels of parallelism for high energy efficiency. We also describe mathematical simplifications for making QNN inference more resource-efficient, and show how binary matrix operators can be used as bit-serial building blocks for higher-precision computation.

Preface

This thesis is submitted in partial fulfilment of the requirements for the degree of *philosophiae doctor* (PhD) at the Norwegian University of Science and Technology (NTNU). It is a compilation thesis consisting of an introductory part, seven scientific articles and a concluding part, in accordance with Section 10.1 of the "Regulations for the Doctoral Degree at the Norwegian University of Science and Technology". The articles have been reformatted to keep the appearance of the thesis more consistent, but their scientific content is identical to the published versions.

Associate Professor Magnus Jahre was the main supervisor for the project, with Professor Per Gunnar Kjeldsberg and Doctor Asbjørn Djupdal as the co-supervisors. The research was funded internally by the university, and was carried out mainly at the NTNU Department of Computer Science in Trondheim, Norway and partially at Xilinx Research Labs in Dublin, Ireland.

Acknowledgements

First and foremost, I would like to thank my advisor Associate Professor Magnus Jahre for his unwavering support, apt supervision and good advice. I left every single supervision meeting feeling better than how I came in, and I truly appreciate that he provided me with enough academic freedom to fly with my own wings. I am also extremely grateful to have had the opportunity to work with Michaela Blott during my internship at Xilinx Research Labs in Dublin. Her energy, enthusiasm and dedication are a major source of inspiration to me.

Next, a big round of thanks to my colleagues at the NTNU Computer Architecture Lab and at Xilinx Research Labs. From lunchtime discussions on emergent complexity to payday beers with idle banter, from Rammstein concerts to paper discussion colloquia, they have been an invaluable source of support for both my social wellbeing and my mental growth as a researcher.

And finally, the unspoken heroes of this thesis, whose names appear on no author list of any of my articles but who have made them possible with their endless patience, love and support. My most sincere and heartfelt thanks to my partner Håkon Marthinsen and my parents Betigül Umuroğlu and Fersal Umuroğlu.

Yaman Umuroğlu
Trondheim, 29th November 2017

Contents

A. Overview 1

1. Introduction 3
  1.1. Dark Silicon and the Need for Accelerators . . . 3
  1.2. Choice of Specialization Domains and Substrate . . . 5
  1.3. Research Questions . . . 6
  1.4. Thesis Outline . . . 7

2. Background 9
  2.1. Field-Programmable Gate Arrays . . . 9
  2.2. Hardware/Software Codesign . . . 11
  2.3. Implementing Field-Programmable Gate Array (FPGA) Accelerators 11
  2.4. Memory System Design . . . 12
  2.5. The Roofline Model . . . 14

3. Research Summary 17
  3.1. Overview . . . 17
  3.2. Articles on Sparse Linear Algebra . . . 17
  3.3. Articles on Deep Neural Networks . . . 21
  3.4. Other Articles . . . 23

B. Sparse Linear Algebra 29

1. Introduction 31
  1.1. On Sparsity . . . 31
  1.2. Sparse Kernels . . . 32

2. Background 33
  2.1. Sparse Matrix–Vector Multiplication . . . 33
  2.2. Breadth-First Search as Sparse Linear Algebra . . . 33
  2.3. Computational Properties . . . 35

B1. An Energy Efficient CM Backend for FPGA SpMV Accelerators **39**
B1.1. Introduction 39
B1.2. Background 41
B1.3. Decoupling SpMV Memory Behavior and Computation 44
B1.4. Designing A Bandwidth-Saturating Backend 45
B1.5. Experimental Evaluation 51
B1.6. Related Work 59
B1.7. Conclusion and Future Work 61

B2. A Vector Caching Scheme for Streaming FPGA SpMV Accelerators **65**
B2.1. Introduction 65
B2.2. Background and Related Work 67
B2.3. Vector Caching Scheme 70
B2.4. Experimental Evaluation 76
B2.5. Conclusion and Future Work 80

B3. Random Access Schemes for Efficient FPGA SpMV Acceleration **85**
B3.1. Introduction 86
B3.2. Background 88
B3.3. Frontend Architectures and Random Access 91
B3.4. The Nonblocking Cooperative Vector Caching Scheme 95
B3.5. Experimental Setup 105
B3.6. Results 106
B3.7. Conclusion and Future Work 117

B4. Hybrid BFS on a Single-Chip FPGA-CPU Heterogeneous Platform **121**
B4.1. Introduction 122
B4.2. Background 123
B4.3. Hybrid BFS on an FPGA-CPU Hybrid 126
B4.4. Results 135
B4.5. Related work 141
B4.6. Conclusion and Future Work 142

C. Deep Neural Networks **153**

1. Introduction **155**
1.1. Inference with Deep Neural Networks 155
1.2. Quantized Neural Networks 156

2. Background **159**
2.1. The Anatomy of Inference Computations 159
2.2. Redundancy and Quantization 160

2.3. Computational Properties 162

C1. FINN: A Framework for Fast, Scalable BNN Inference 167
C1.1. Introduction 167
C1.2. Background 169
C1.3. BNN Performance and Accuracy 173
C1.4. BNNs on Reconfigurable Logic 175
C1.5. Evaluation 186
C1.6. Conclusion 193

C2. Scaling BNNs on Reconfigurable Logic 199
C2.1. Introduction 200
C2.2. Background 201
C2.3. BNNs on Reconfigurable Logic 202
C2.4. Padding for BNN Convolutions 205
C2.5. Evaluation 207
C2.6. Conclusion 213

C3. Streamlined Deployment for Quantized Neural Networks 217
C3.1. Introduction 217
C3.2. Streamlined QNNs 219
C3.3. Inference with Few-Bit Weights and Activations on Mobile CPUs . . 222
C3.4. Evaluation 224
C3.5. Conclusion and Future Work 229

D. Concluding Remarks 239

1. Conclusion 241
1.1. Research Question 1 241
1.2. Research Question 2 242

2. Future Work 245
2.1. Sparse Linear Algebra 245
2.2. Deep Neural Networks 247

Part A.

Overview

1. Introduction

Computers have gradually transformed human society, fueled by the extraordinary trend of increasing compute power in smaller form-factors at a cheaper price as predicted by Moore's Law [22]. However, this trend has stagnated and today's computers are now limited by their *energy efficiency*, i.e., computers must perform more computation for the same amount of (or less) electrical energy than what they use today if this trend is to be sustained.

The overarching goal of this thesis is to gain insight into how this problem can be alleviated by means of *specialized hardware for particular types of applications*. The remainder of this chapter will introduce the problem and the proposed approach in more details, then describe the structure of the rest of the thesis.

1.1. Dark Silicon and the Need for Accelerators

For over forty years, improvements in transistor size, supply voltage scaling and architecture fueled the increase in the computational power of general-purpose Central Processing Units (CPUs), but there is evidence that this trend is not sustainable [18]. While the number of transistors per area keeps growing in newer semiconductor technology generations, it is no longer possible to power all the transistors on the chip without significantly increasing the power budget. In many cases, increasing the power budget is not feasible due to limitations in cooling or battery life. Keeping the same power budget results in areas on the chip without power, referred to as *dark silicon* [11].

With Dark Silicon clouding the horizon for computing systems, computer architects have been searching for new solutions. Taylor [27] mentions four generic classes of solutions (referred to as "The Four Horsemen of the Dark Silicon Apocalypse") which could be used to attack the dark silicon problem. These can be summarized as follows:

- *The Shrinking Horseman*, shrinking chip sizes to prune dark silicon areas.

Table 1.1.: Heterogeneity trends in smartphones. Compiled from [21, 35–38].

Product	Year	Cores
Nokia 9000	1996	24 MHz CPU
Palm Treo 650	2004	312 MHz CPU
Samsung SGH-D500	2005	100 MHz CPU + Saturn DSP
Nokia N810	2007	330 MHz CPU + 220 MHz DSP + GPU + imaging accelerator
Motorola Moto X	2013	1.7 GHz dual-core CPU + GPU + imaging accelerator + natural language processing core + contextual computing core

- *The Dim Horseman*, using underclocked logic or near-threshold voltage designs to increase operating efficiency.
- *The Deus Ex Machina Horseman*, fundamental breakthroughs in transistor technology which will lessen or remove the dark silicon problem.
- *The Specialized Horseman*, which is further elaborated below.

The last of these ideas, aptly named "The Specialized Horseman", proposes a move towards specialized, highly energy-efficient co-processors or accelerators. In a manner of speaking, dark silicon is a manifestation of a lack of energy efficiency. Specialized logic targeting particular applications or domains can offer energy efficiency that is orders of magnitude greater than general-purpose processors on an equal or greater level of performance [7, 14, 16, 19, 24]. This would contribute significantly to the overall energy efficiency of the system for running applications targeted by specialized logic.

More energy-sensitive platforms such as smartphones and mobile computing devices have already been demonstrating a trend of heterogeneous processors with increasingly specialized components in the form of System on a Chip (SoC) elements, as illustrated in Table 1.1. There is also a significant shift towards accelerators in the computer architecture research community, which is evidenced by the number of sessions dedicated to specialized hardware in top computer architecture conferences. For instance, in the programme of the 2016 International Symposium on Computer Architecture (ISCA), there were three entire sessions dedicated to neural network accelerators, as well as another session for graph analytics and big data accelerators. In light of this information, **The Specialized Horseman** was chosen to be the method for improving the energy efficiency of computing systems in this thesis.

1.2. Choice of Specialization Domains and Substrate

The very word *specialization* implies that the resulting accelerators are not *general-purpose*, i.e., they cannot execute arbitrary compute workloads, but only the particular ones that they were designed for. We note that the *degree* of specialization is particularly important; due to the costs involved in designing such systems, it is desirable to make them *domain-specific* (suitable for a range of applications) instead of *single-application-specific*. Cong et al. [8] refer to this approach as *customizable domain-specific computing*, and motivate it with three observations:

1. Each user or enterprise typically has high computing demand only in one or a few application domains (e.g., graphics for game developers, circuit simulation for integrated circuit designers, financial analytics for investment banks), which can be targeted with accelerators for high performance and efficiency. Other computing needs, such as e-mail or word processing can be satisfied with available computing technologies.

2. The gap in performance and efficiency between a domain-specific solution and a general-purpose solution can be several orders of magnitude.

3. Designing a custom Application-Specific Integrated Circuit (ASIC) for each and every application is prohibitively expensive, hence some customizability to address a range of applications is needed.

Two key methodology questions in domain-specific accelerator research are *what* to accelerate, and *where* to accelerate it in terms of the hardware substrate. These questions are addressed in the following subsections.

1.2.1. The Domains

As specialized acceleration has limited applicability, the choice of which domain(s) will be accelerated is significant. In this thesis, the following criteria were used to guide the topic and level of specialization for acceleration:

- *Necessity.* The target domains should require acceleration. Computations that already meet performance and efficiency requirements on existing general-purpose computers does not necessitate specialization and acceleration to the same extent as computation which runs too slowly or inefficiently on current devices.

- *Impact.* The acceleration of target domains should have some impact, e.g., they should be considered to be of major importance for the progress of science or possessing large market potential.
- *Granularity.* The chosen level of specialization should be able to address a range of applications, instead of targeting just a particular implementation of a single application.

The "13 Berkeley Dwarfs" proposed by Asanovic et al. [1] were a major source of inspiration in the choice of these focus areas for this thesis. These "dwarfs" are particular algorithmic methods that capture a widely-occurring pattern of computation and communication believed to be of major importance for at least the next decade of computing, which by definition fulfill two of the three criteria listed above. The focus areas for this thesis were chosen to be **Sparse Linear Algebra**, which is explored in Part B, and **Deep Neural Networks**, which is explored in Part C. In terms of the Berkeley Dwarfs, these focus areas fall within the Sparse Linear Algebra and Dense Linear Algebra dwarfs. The introduction sections in each part provide more insight into why these focus areas were chosen.

1.2.2. The Hardware Substrate

A suitable hardware substrate is necessary to study and evaluate how the proposed accelerators will perform. The chosen substrate for specialization in this thesis is reconfigurable logic in the form of **Field-Programmable Gate Arrays (FPGAs)**. As will be described in more detail in Section 2.1, the behavior of any digital circuit can be constructed in the FPGA as long as there are enough logic resources available, and can be changed as many times as desired. FPGAs have received significant attention as a new engine for heterogeneous computing in the last few years, as exemplified by Intel's 2015 acquisition of the world's second-largest FPGA maker Altera, and their subsequent announcements regarding Xeon+FPGA chips.

1.3. Research Questions

The overarching goal for this thesis is *to gain insight into how the energy efficiency of computer systems can be improved via specialization.* Based on the choice of focus domains and substrate discussed in Section 1.2, the goal is formulated as two research questions:

RQ1. How can sparse linear algebra benefit from FPGA acceleration?

RQ2. How can deep neural network inference benefit from FPGA acceleration?

These questions permit exploration on both sides of the traditional hardware-software boundary. That is, we do not restrict the effort to exploring FPGA hardware accelerator architectures, but also accept changes that go across the traditional hardware-software boundary. This may include changing the data layout, data preprocessing and mathematical reformulations, and may be seen as a limited form of hardware/software codesign (Section 2.2). This allows for optimizing the computation as a whole without being limited by the traditional rigid abstraction boundary, potentially yielding greater acceleration and efficiency.

1.4. Thesis Outline

This is a *compilation thesis* organized into four parts; an introductory part followed by two parts containing a compilation of research articles, and a final part with concluding remarks.

This introductory part (Part A) contains an overview of the general problem, the research questions pursued, and the general background necessary to understand the research articles in the thesis. Afterwards, it presents a brief summary of the articles included in the thesis, how they are related to general computing concepts and to each other. The second and third parts consist of a collection of research articles on the design of accelerators for sparse linear algebra (Part B) and deep neural networks (Part C), respectively. As these two domains are substantially different in terms of their computational requirements, they both start with an introduction to and some background on their respective fields to help the reader better understand the context of the articles. The final part of the thesis (Part D) concludes with a summary of results from the research articles and presents several directions for future work.

2. Background

This chapter provides the general background necessary to understand the content of this thesis, pertaining to FPGAs as an acceleration fabric, design of FPGA accelerators and memory systems. Parts B and C contain their own background chapters with further domain-specific details on sparse linear algebra and deep neural networks.

2.1. Field-Programmable Gate Arrays

Field-Programmable Gate Arrays (FPGAs) are chips containing arrays of Look-Up Tables (LUTs), fixed-function blocks, signal routing and clock networks, as exemplified in Figure 2.1. By configuring the contents of the LUTs and how they are connected together with each other and fixed-function blocks, the behavior of any digital circuit can be constructed in the FPGA, and this configuration can be changed as many times as desired. These capabilities of realizability and reconfigurability are of prime importance for the choice of FPGAs as an accelerator substrate for this thesis. In general, a circuit implemented in reconfigurable logic will always be slower and less efficient than a comparable, dedicated circuit in silicon. However, this efficiency gap is smaller for applications that can make use of the fixed-function blocks, and off-the-shelf reconfigurable chips do not carry the prohibitive price tags of mask costs or post-production errors for ASICs.

Due to the repetitive and regular structure of their internal components, FPGAs have benefited substantially from the improvements in semiconductor process technology, and today's models possess substantial on-chip memory capacity, bandwidth and compute capabilities. They have received significant attention as a new engine for heterogeneous computing in the last few years, and significant acceleration on key-value stores [15], financial option pricing [10], and genome sequencing [26] have been demonstrated on FPGAs. Besides the potential performance advantages, Hoe [13] points out that FPGA acceleration is first and foremost about energy efficiency, owing to their high-degree of slow-clocked parallelism and avoidance of Von Neumann programmability overheads.

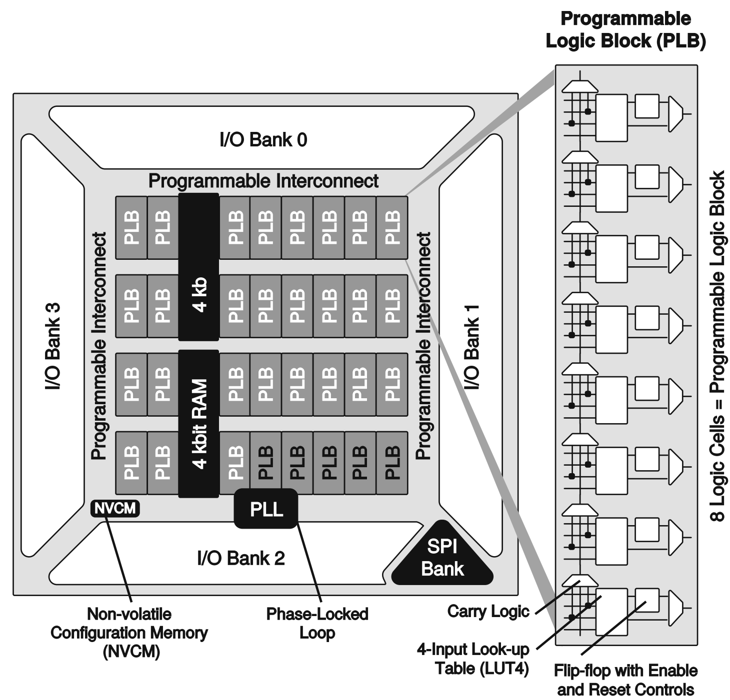


Figure 2.1.: The organization of a Lattice Semiconductor iCE40 FPGA. Reproduced from [23].

2.2. Hardware/Software Codesign

Implementations of modern computer systems can be quite complex, even when targeting a single domain or application. For instance, a modern SoC such as the Xilinx Zynq family of chips may contain several different (heterogeneous) computing engines as well FPGA fabric. Achieving desired functionality on such a system requires exploring a vast design space that encompasses hardware architecture exploration, mapping the functionality to available resources and assessing performance and efficiency to remain within system constraints. This process is referred to as *hardware/software codesign* [28] and as the name implies, it involves the concurrent design of hardware and software. The key advantage obtained in this method is the ability to approach the problem as a larger whole, potentially increasing the performance and efficiency of the final solution by making adjustments without being limited by the traditional abstraction layers in computer architecture.

Several research articles in this thesis use codesign-like techniques, such as partioning an algorithm for hybrid execution between a CPU and an FPGA accelerator, performing mathematical simplifications on neural networks at compile time, and preprocessing data in software to extract relevant characteristics for reconfigurable hardware.

2.3. Implementing FPGA Accelerators

FPGAs are notorious for being difficult to program [3]. The entry point for most FPGA vendor tools are Register Transfer Level (RTL) designs expressed in Verilog or VHDL, which introduce several challenges for rapid development. As the chosen evaluation technique for this thesis is FPGA implementation, other languages and tools were used to overcome two of the traditional RTL design challenges:

1. *Language features.* In terms of languages, neither Verilog nor VHDL were originally conceived to express synthesizable RTL constructs – they were intended for *simulating* hardware, which means it is possible to express programs in them that cannot be synthesized into FPGA circuits. Furthermore, both languages first appeared in the 1980s, and lack the powerful abstraction facilities that are common in modern programming languages, which further decreases design productivity. Chisel was proposed by Bachrach et al. [2] to address these problems. It is embedded into the Scala language, which has extensive capabilities for building Domain-Specific Languages (DSLs). A Chisel program is a *parametrizable hardware generator* that expresses which RTL primitives are instantiated and how they are connected together, and barring syntax errors its output is guaranteed to be synthesizable. By using different backends, it is possible to

generate Verilog or a cycle-accurate C++ simulation of the design from the same source code, which boosts productivity. Chisel was used in the implementation of the papers contained in Part B of this thesis.

2. *Abstraction level.* Even with the capabilities of languages like Chisel, RTL design is less productive and more challenging than software development. Namely, in RTL design, the designer must carefully specify (and debug) the cycle-by-cycle behavior of a complex circuit by cobbling together simple components, designing state machines and control logic to ensure correct behavior, and adding pipeline registers to increase the clock frequency when desired. High-Level Synthesis (HLS) tools such as Vivado HLS [40] aim to close this gap by allowing the designer to specify the behavior and desired performance of the circuit at a higher level of abstraction, and synthesizing RTL from this specification. This approach works especially well for circuits with static schedules, such as matrix multiplications whose sizes are known at compile time. To pursue these benefits, Vivado HLS was used for the accelerator implementations in Part C of this thesis.

2.4. Memory System Design

How data is laid out, accessed, stored and moved between parts of the system is a key component for any computer architecture, including accelerators. This section provides background on several memory system design concepts that are used in the thesis.

2.4.1. On-Chip Memory

FPGAs contain a limited amount of On-Chip Memory (OCM), sometimes referred to as Block RAM (BRAM). Any address in any BRAM can be accessed with a single clock cycle of latency, which can deliver multiple terabytes per second of memory bandwidth in modern FPGAs. As with any other component on the FPGA, on-chip memory blocks do not enforce a particular architecture upon the designer, and can be flexibly connected to other components to shape the memory system as designed. For instance, they can be configured as read-only memory (ROM) whose contents are set at FPGA configuration time, as a FIFO buffer storage by adding enqueue/dequeue logic, or as caches by adding appropriate management logic.

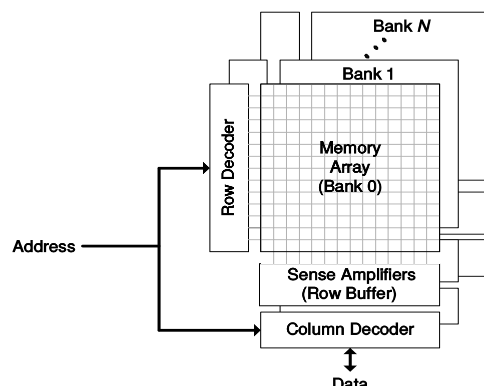


Figure 2.2.: The internal three-dimensional organization of DRAM, showing banks, rows and columns. Reproduced from [20].

2.4.2. Off-Chip Memory

Any memory that is accessed via external I/O pins is *off-chip memory*, although in this thesis the term is used to refer to off-chip DRAM, commonly found on FPGA boards to provide a reasonably large-capacity memory at low cost. These DRAM chips are interfaced by a *memory controller* on the FPGA which services memory requests from the accelerator(s) by implementing the signaling standard required by DRAM. Due to the internal organization of DRAM chips shown in Figure 2.2, the way in which memory accesses are scheduled has a large impact on the achievable throughput and access latency [20]. Namely, data can be read or written only as an entire DRAM row in each bank. This makes accessing sequential positions as large bursts very effective, since the entire DRAM row buffer can be utilized. However, performing fine-grained random accesses involves repeatedly fetching new rows without utilizing the entirety of the previously fetched row buffer, which increases the access latency and decreases effective bandwidth substantially. For instance, orders-of-magnitude worse bandwidth is reported on the RandomAccess benchmark included in the HPCC benchmark suite [17] compared to sequential accesses. This is a key problem for sparse linear algebra kernels with random memory access patterns, which Part B of this thesis focuses on.

2.4.3. Decoupled Access-Execute and Latency-Insensitive Design

Decoupled access-execute architectures originally proposed by Smith [25] refers to architectures with a dedicated *access* component, which is responsible for memory

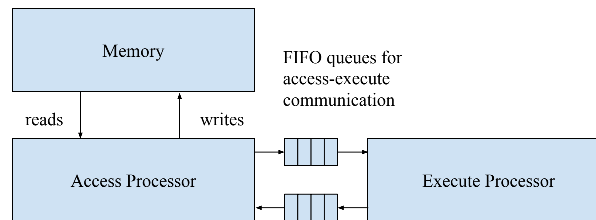


Figure 2.3.: Conceptual overview of a decoupled access-execute architecture.

operations such as reads and writes, and a dedicated *execute* component, which is responsible for carrying out computation on accessed memory and producing results. These two components communicate via FIFO queues, as conceptually illustrated in Figure 2.3. Such an architecture enables clean separation and separate optimization of these two main components, and also lets the designer easily analyze potential performance issues by examining the flow rate and backpressure in the communication queues.

Beyond decoupling memory accesses and computation, a similar principle of *latency-insensitive design* [5] in the form of *synchronous elastic architectures* [9] can be applied to system design at a finer-grained level. Namely, the system can be decomposed into multiple parallel components which communicate via FIFO queues. The requirement from each component is that it can be *stalled with backpressure*, i.e., when one or more FIFOs that it wants to write to are full. As long as each component consumes available elements from its input FIFOs as quickly as possible, the entire system will self-schedule and naturally process data at the highest throughput permitted by the slowest component. This design paradigm was shown to be successful for construction of FPGA accelerators by the Stream Computations Organized for Reconfigurable Execution (SCORE) project [6]. Both decoupled access-execute and latency-insensitive design have served as inspiration for the accelerator architectures in this thesis.

2.5. The Roofline Model

The roofline [39] is a well-known visual performance model that provides an upper bound on the performance of a particular computational kernel when running on a particular computer architecture. The key property that must be known about the computational kernel is the *arithmetic intensity*, which is the ratio of computation to communication measured in compute operations per byte read from off-chip memory. For the architecture that this computation will be run on, the peak off-chip memory

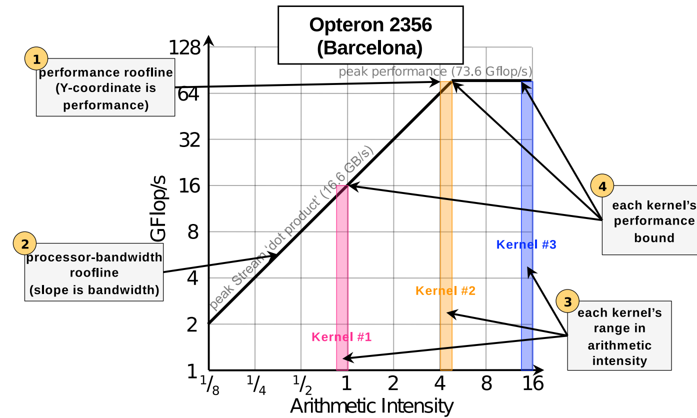


Figure 2.4.: Example of a roofline model. Reproduced from [4].

bandwidth and the peak compute capability for the kinds of operations used in the kernel must be known. With these numbers available, the roofline model can be constructed as illustrated in Figure 2.4. The model considers two principal performance bounds, computation and communication, and considers the execution time of the application bounded by each of these to determine which of them will be the bottleneck, and how much performance may be attained. In the visual model, the sloping part of the roofline is the communication-bound region, whereas the flat part is the compute-bound region. When a vertical line corresponding to the arithmetic intensity of the computational kernel in question is drawn, it will intersect the roofline in one of these regions, indicating what will be the performance bottleneck for this kernel.

Since it considers platform and kernel characteristics at a quite abstract level, the roofline model is also useful for estimating performance for accelerators. It is used in Part C of this thesis to help estimate the performance gains from quantization for deep neural networks.

3. Research Summary

This section provides an overview of the research articles included in this thesis, how they relate to each other and to general computing concepts. It is intended to serve as a roadmap that helps place the articles included in the thesis into the bigger picture. After the articles have been presented, the final part of this thesis will present a synthesis of the contributions from individual articles into a larger whole.

3.1. Overview

As discussed in Section 1.2, this thesis explores the acceleration of two distinct domains. It includes seven papers, four on sparse linear algebra in Part B, and three on deep neural networks in Part C. Each paper is identified with the part name and number (B1, B2, C1, C2. . .) and will be referred to as such for brevity in the rest of the discussion.

Figure 3.1 presents a mind map linking computing concepts to papers in the thesis in order to provide a visual overview of how the papers align with different themes, and with each other. It can be observed that although the two domains are distinct, a number of common themes such as algorithmic reformulation, reduced precision operations and decoupled hardware architectures are recurring themes in papers in both domains. The following sections provide brief summaries of each paper, and their relationships to each other.

3.2. Articles on Sparse Linear Algebra

The articles in Part B discuss how sparse linear algebra, in particular the Sparse Matrix–Vector Multiplication (SpMV) and Sparse Matrix–Sparse Vector Multiplication (SpM-SpV) kernels, can benefit from acceleration on reconfigurable logic. The key aspect of the investigation here is the memory system and how it can be made efficient, since sparse kernels are known to be bottlenecked by random, indirect memory accesses. As the sparsity structure that determines the memory accesses becomes known only at runtime, it is challenging to devise a single architecture that works well for all sparse

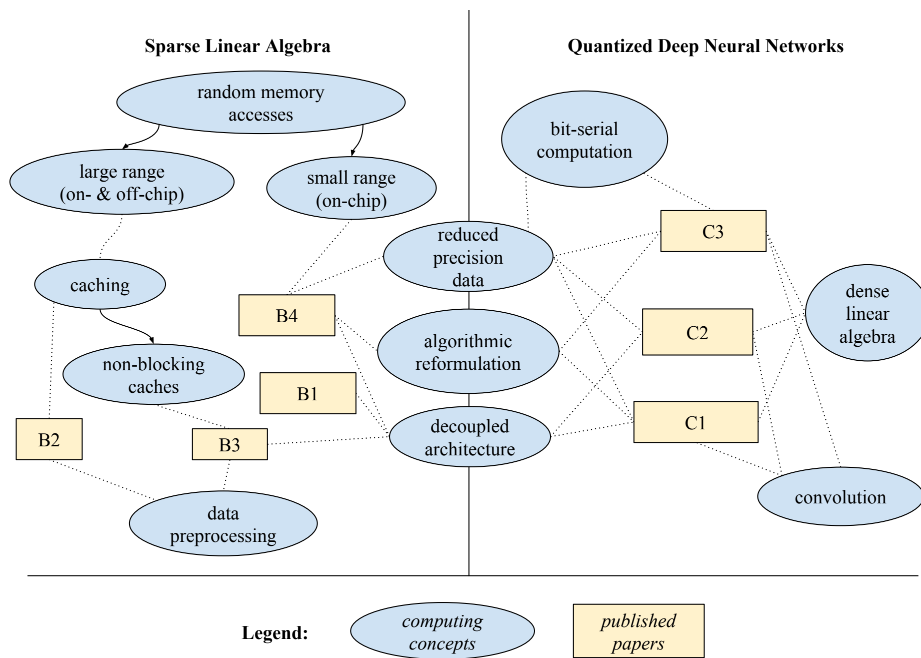


Figure 3.1.: A mind map of publications that are part of this thesis and their relations to computing concepts.

matrices. We use lightweight data preprocessing to adapt to the sparsity structure at hand, taking advantage of the flexibility of reconfigurable logic. We also explore the applicability of accelerated sparse linear algebra to graph algorithm primitives, taking advantage of data layout changes and reduced precision representations to further increase efficiency.

Paper B1: An Energy Efficient Column-Major Backend for FPGA SpMV Accelerators

By *Yaman Umuroglu and Magnus Jahre*
In *Proceedings of the 2014 IEEE 32nd International Conference on Computer Design (ICCD)*
IEEE, 2014 [31]

This work presents a design strategy for SpMV accelerators in the spirit of decoupled access-execute, where the memory and computation are handled by two decoupled units (backend and frontend) with matched throughput. This modular design is adopted by all other articles on sparse linear algebra in this thesis. Focusing on the sequential memory accesses by the backend component, experimental results are presented that indicate that even the sequential memory access component can suffer from degraded external memory bandwidth utilization due to many parallel requests. An alternative, interleaved data layout is then presented to increase DRAM bandwidth and power efficiency for FPGA SpMV accelerators. An SpMV accelerator backend design that uses this layout is proposed and evaluated on a suite of sparse matrices to characterize the performance and efficiency benefits,

Paper B2: A Vector Caching Scheme for Streaming FPGA SpMV Accelerators

By *Yaman Umuroglu and Magnus Jahre*
In *Proceedings of the 2015 International Symposium on Applied Reconfigurable Computing (ARC)*
Springer International Publishing, 2015 [30]

In terms of the modular SpMV design strategy presented in Paper B1, this paper focuses on the frontend and the random memory access problem. A hardware-software caching scheme for handling the SpMV random access component is presented. The

essence of the technique is to increase the resource efficiency and performance of the hardware cache by using lightweight data preprocessing in software to extract relevant access pattern information. The software part of the scheme involves preprocessing the sparse matrix to extract the required cache capacity for stall-free acceleration, and adding a single bit of information for each matrix row to avoid cold misses in the cache. In turn, the proposed hardware cache can be resized to the determined capacity (which is typically much smaller than keeping the entire random access component on-chip), and is able to use the extra information to avoid cold misses. The merits of the hardware-software caching approach are validated with a range of sparse matrices.

Paper B3: Random Access Schemes for Efficient FPGA SpMV Acceleration

By *Yaman Umuroglu, Donn Morrison and Magnus Jahre*
In *Microprocessors and Microsystems, volume 47 part B*
Elsevier, 2016 [32]

This work is an invited journal paper extension of Paper B2, offering deeper insights into the problem, improving the caching technique and providing more results. How prior work has handled the random memory access component in SpMV is discussed and grouped under three alternatives (all on-chip, all off-chip, caching). The hardware-software vector caching scheme for SpMV from Paper B2 is combined with nonblocking caches, a classical memory system technique in out-of-order processors that is used to extract more Memory-Level Parallelism (MLP) and increase bandwidth utilization. The improved caching technique is validated with extensive experiments, demonstrating how multiple outstanding requests can increase SpMV performance. The cost of preprocessing is also factored into the overall performance, showing how the performance gains from preprocessing outweigh the time cost of preprocessing in relatively few iterations.

Paper B4: Hybrid Breadth-First Search on a Single-Chip FPGA-CPU Heterogeneous Platform

By *Yaman Umuroglu, Donn Morrison and Magnus Jahre*
In *Proceedings of the 2015 25th International Conference on Field Programmable Logic and Applications (FPL)*
IEEE, 2015 [34]

The acceleration of the Breadth-First Search (BFS) graph traversal kernel is studied on a single-chip FPGA-CPU system. By using the *matrices over semirings* concept, BFS can be viewed as SpMV or SpMSpV on the Boolean semiring. This makes it possible to apply insights from papers B1 and B2 to offer significant acceleration on BFS. This work also describes how a simple algorithmic reformulation can be used to make the random access range much smaller. Namely, the distances to the root node can be generated with an additional step instead of on-the-fly, effectively decoupling the distance generation from the traversal, thus keeping the OCM footprint for random accesses very small. Additionally, a computational property of BFS on small-world graphs is exploited for hybrid execution, where some iterations of the kernel are processed on the CPU, and the remainder on the FPGA. The resulting BFS implementation is more efficient (over twice as efficient compared to prior work) in terms of utilizing the external memory bandwidth.

3.3. Articles on Deep Neural Networks

The articles in Part C explore how to apply reconfigurable logic to deep neural network inference. Due to their inherent memory footprint and resource benefits over floating point networks, we focus on Quantized Neural Networks (QNNs), quantifying and demonstrating the performance and energy benefits of QNN inference on FPGAs using roofline models and prototypes with adjustable throughput according to user requirements. By applying our approach on larger networks and FPGAs, we identify on-chip memory storage and bandwidth as the key limitations to scalability. We also study how QNN inference can be performed efficiently on commodity CPUs using bit serial arithmetic, provide performance measurements, and observe that FPGAs have significant performance benefits over CPUs in this regard.

Paper C1: FINN: A Framework for Fast, Scalable Binarized Neural Network Inference

By *Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers*
In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*
ACM, 2017 [29]

This paper explores inference with Binarized Neural Networks (BNNs), the most extreme form of QNNs, on FPGAs. We quantify the potential of BNNs on FPGAs via the

roofline model, and present the FINN framework for building BNN accelerators. FINN composes hardware blocks implementing fully-connected, convolutional and pooling layers according to user-specified throughput requirements. The composed architecture is *streaming* in that it executes all layers of the network in parallel, and *heterogeneous* in that each layer's hardware allocation reflects its computational requirements. A mathematical simplification to absorb floating point batch normalization layers into thresholding activations is also presented, which further lowers resource cost. At the time of its publication, FINN was generating the lowest-latency, highest-throughput and highest-energy-efficiency Deep Neural Network (DNN) image classification accelerators in literature. This performance is possible owing to a combination of massive parallelism, low-precision computations and high arithmetic intensity.

Paper C2: Scaling Binarized Neural Networks on Reconfigurable Logic

By *Nicholas J. Fraser, Yaman Umuroglu, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers*
In *Proceedings of the 8th Workshop and 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM)*
ACM, 2017 [12]

This work builds upon Paper C1 investigates how BNN performance and accuracy scales on larger FPGAs and larger networks. We first show how binary convolutions can be padded with -1 values to increase the accuracy in image recognition tasks, and implement this with minor changes to the Sliding Window Unit from Paper C1. An accelerator classifying the CIFAR-10 dataset with close to state-of-the-art accuracy using a large BNN and 12 thousand frames per second is demonstrated, and the obtained performance is compared to predictions from the roofline models. One of the key findings regarding the limitations in scaling is the utilization of OCM, which is revealed to be relatively poor due to the way in which matrix-matrix multiplications are implemented. A matrix-multiple vectors engine is proposed as a potential solution to the OCM utilization problem.

Paper C3: Streamlined Deployment for Quantized Neural Networks

By *Yaman Umuroglu and Magnus Jahre*
In *The 2017 International Workshop on Highly Efficient Neural Networks Design (HENND), part of ESWEEK*
arXiv:1709.04060 [33]

Trained QNNs still contain some floating point parameters and computation. In this work, we extend and generalize the transformation proposed in Paper C1 to QNNs by moving, collapsing and absorbing linear transforms into thresholding layers. This transformation is called *streamlining* and simplifies the trained network at compile time. Additionally, we explore how bit-serial processing can be used to run QNN inference efficiently on existing CPUs. Although this work evaluates the presented techniques on CPUs and not FPGAs, these techniques are directly applicable to FPGA accelerators as well. Namely, the simplification by streamlining prevents floating point computation from taking up valuable logic and DSP slice resources for FPGA accelerators. Additionally, the proposed bit-serial processing technique is also suitable for applying on FPGAs, supplementing the binary matrix primitives proposed in Paper C1 with the ability to support arbitrary integer quantization at runtime. Finally, this work establishes a CPU baseline which further highlights the benefits of FPGA acceleration for QNNs, since the resulting CPU performance is orders of magnitude worse than the accelerators proposed in Papers C1 and C2.

3.4. Other Articles

Several research articles that were produced during the PhD work are not included in this thesis, either due to their extended versions being included instead, or because of their focus falling outside the scope of the thesis.

- Yaman Umuroglu and Magnus Jahre. "Work-in-Progress: Towards Efficient Quantized Neural Network Inference on Mobile Devices". In *Proceedings of the 2017 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*.
- Michaela Blott, Thomas Preusser, Yaman Umuroglu, Miriam Leeser, Nicholas Fraser, Kenneth O'Brien and Giulio Gambardella. "Scaling Neural Network Performance through Customized Hardware Architecture". In *Proceedings of the 2017 IEEE International Conference on Computer Design (ICCD)*.

- Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella and Michaela Blott. "A C++ Library for Rapid Exploration of Binary Neural Networks on Reconfigurable Logic". Presented at *The Second International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC) at Supercomputing 2016 (SC).*

Bibliography

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams et al. *The landscape of parallel computing research: A view from Berkeley*. Tech. rep. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[2] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek and K. Asanović. 'Chisel: constructing hardware in a Scala embedded language'. In: *Proc. of the Design Automation Conference*. 2012.

[3] D. F. Bacon, R. Rabbah and S. Shukla. 'FPGA programming for the masses'. In: *Communications of the ACM* 56.4 (2013), pp. 56–63.

[4] D. H. Bailey, R. F. Lucas and S. Williams. *Performance tuning of scientific applications*. CRC Press, 2010.

[5] L. P. Carloni, K. L. McMillan and A. L. Sangiovanni-Vincentelli. 'Theory of latency-insensitive design'. In: *IEEE Transactions on computer-aided design of integrated circuits and systems* 20.9 (2001), pp. 1059–1076.

[6] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek and A. DeHon. 'Stream computations organized for reconfigurable execution (SCORE)'. In: *FPL*. Springer. 2000, pp. 605–614.

[7] E. S. Chung, P. A. Milder, J. C. Hoe and K. Mai. 'Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus?' In: *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 2010, pp. 225–236.

[8] J. Cong, V. Sarkar, G. Reinman and A. Bui. 'Customizable domain-specific computing'. In: *IEEE Design & Test of Computers* 28.2 (2011), pp. 6–15.

[9] J. Cortadella, M. Kishinevsky and B. Grundmann. 'Synthesis of synchronous elastic architectures'. In: *Proceedings of the 43rd annual Design Automation Conference*. ACM. 2006, pp. 657–662.

[10] C. De Schryver, I. Shcherbakov, F. Kienle, N. Wehn, H. Marxen, A. Kostiuk and R. Korn. 'An energy efficient FPGA accelerator for monte carlo option pricing with the heston model'. In: *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*. IEEE. 2011, pp. 468–474.

[11] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam and D. Burger. 'Dark silicon and the end of multicore scaling'. In: *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*. IEEE. 2011, pp. 365–376.

[12] N. J. Fraser, Y. Umuroglu, G. Gambardella, M. Blott, P. Leong, M. Jahre and K. Vissers. 'Scaling Binarized Neural Networks on Reconfigurable Logic'. In: *Proceedings of the 8th Workshop and 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*. Vol. 2017. 2017.

[13] J. C. Hoe. 'Technical Perspective: FPGA Compute Acceleration is First About Energy Efficiency'. In: *Commun. ACM* 59.11 (Oct. 2016), pp. 113–113. DOI: 10.1145/2996866. URL: http://doi.acm.org/10.1145/2996866.

[14] T. Instruments. *OMAP Applications Processors*. 2013. URL: http://www.ti.com/product/OMAP4470.

[15] Z. István, G. Alonso, M. Blott and K. Vissers. 'A flexible hash table design for 10gbps key-value stores on fpgas'. In: *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE. 2013, pp. 1–8.

[16] R. C. Larsen and O. Janbu. *Introduction to EFM32 Microcontrollers*. URL: http://cdn.energymicro.com/dl/pdf/efm32_introduction_white_paper.pdf.

[17] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner and D. Takahashi. 'The HPC Challenge (HPCC) benchmark suite'. In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. 2006, p. 213.

[18] J. Preshing. *A Look Back at Single-Threaded CPU Performance*. Feb. 2012. URL: `http://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance`.

[19] Qualcomm. *Qualcomm Snapdragon Processors*. 2013. URL: `http://www.qualcomm.com/snapdragon`.

[20] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson and J. D. Owens. 'Memory access scheduling'. In: *ACM SIGARCH Computer Architecture News*. Vol. 28. 2. ACM. 2000, pp. 128–138.

[21] Samsung. *Samsung SGH-D500 Service Manual*. Nov. 2004. URL: `http://www.manualslib.com/manual/261262/Samsung-Sgh-Sgh-D500.html?page=8#manual`.

[22] R. R. Schaller. 'Moore's law: past, present and future'. In: *Spectrum, IEEE* 34.6 (1997), pp. 52–59.

[23] L. Semiconductor. *iCE40 LP/HX Family Data Sheet*. Oct. 2017. URL: `http://www.latticesemi.com/~/media/LatticeSemi/Documents/DataSheets/iCE/iCE40LPHXFamilyDataSheet.pdf`.

[24] D. E. Shaw, M. M. Deneroff, R. O. Dror, J. S. Kuskin, R. H. Larson, J. K. Salmon, C. Young, B. Batson, K. J. Bowers, J. C. Chao et al. 'Anton, a special-purpose machine for molecular dynamics simulation'. In: *ACM SIGARCH Computer Architecture News*. Vol. 35. 2. ACM. 2007, pp. 1–12.

[25] J. E. Smith. 'Decoupled access/execute computer architectures'. In: *ACM SIGARCH Computer Architecture News*. Vol. 10. 3. IEEE Computer Society Press. 1982, pp. 112–119.

[26] O. O. Storaasli. *Accelerating genome sequencing 100-1000X with FPGAs*. Tech. rep. Oak Ridge National Laboratory (ORNL), 2008.

[27] M. Taylor. 'A Landscape of the New Dark Silicon Design Regime'. In: vol. PP. 99. 2013, pp. 1–1. DOI: `10.1109/MM.2013.90`.

[28] J. Teich. 'Hardware/software codesign: The past, the present, and predicting the future'. In: *Proceedings of the IEEE* 100.Special Centennial Issue (2012), pp. 1411–1430.

[29] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre and K. Vissers. 'FINN: A Framework for Fast, Scalable Binarized Neural Network Inference'. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. 2017, pp. 65–74.

[30] Y. Umuroglu and M. Jahre. 'A Vector Caching Scheme for Streaming FPGA SpMV Accelerators'. In: *Applied Reconfigurable Computing*. Vol. 9040. 2015.

[31] Y. Umuroglu and M. Jahre. 'An Energy Efficient Column-Major Backend for FPGA SpMV Accelerators'. In: *Computer Design, IEEE Int. Conf. on*. 2014.

[32] Y. Umuroglu and M. Jahre. 'Random Access Schemes for Efficient FPGA SpMV Acceleration'. In: *Microprocessors and Microsystems* 47 (2016), pp. 321–332.

[33] Y. Umuroglu and M. Jahre. 'Streamlined Deployment for Quantized Neural Networks'. In: *arXiv preprint arXiv:1709.04060* (2017).

[34] Y. Umuroglu, D. Morrison and M. Jahre. 'Hybrid Breadth-First Search on a Single-Chip FPGA-CPU Heterogeneous Platform'. In: *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*. 2015.

[35] Wikipedia. *Nokia 9000 Communicator*. Feb. 2013. URL: `http://en.wikipedia.org/wiki/Nokia_9000_Communicator`.

[36] Wikipedia. *Nokia N810*. Sept. 2013. URL: `http://en.wikipedia.org/w/index.php?title=Moto_X`.

[37] Wikipedia. *Nokia N810*. Aug. 2013. URL: `http://en.wikipedia.org/wiki/Nokia_N810`.

[38] Wikipedia. *Treo 650*. Feb. 2013. URL: `http://en.wikipedia.org/wiki/Treo_650`.

[39] S. Williams, A. Waterman and D. Patterson. 'Roofline: an insightful visual performance model for multicore architectures'. In: *Communications of the ACM* 52.4 (2009), pp. 65–76.

[40] Xilinx Inc. 'Vivado Design Suite User Guide: High-Level Synthesis'. In: *White Paper* (2016).

Part B.

Sparse Linear Algebra

1. Introduction

1.1. On Sparsity

Viewing the universe that we live in as entities and their interactions is a useful model [9] for analyzing it. At the non-quantum scale, the myriad entities in our universe –be they physical objects or abstract concepts– are often *sparsely connected*: not everything directly interacts with (or is directly related to) everything else. For instance, the human brain contains approximately $9 \cdot 10^7$ neurons, but each neuron is connected to only 7000 other neurons on average [26]. Similarly, Wikipedia had 2.3 million articles but an average of about 24 links per article in 2008 [24].

Figure 1.1 illustrates the concepts of dense and sparse connections using two nine-node graphs. Sparse structures are often represented as sparse matrices where zero entries correspond to unconnected entity pairs, as exemplified in Figure 1.2. Sparse matrix representations can be found in many contexts and domains, including in neural networks [42], graph and network analysis [15], computational biology and chemistry, electronic design automation, N-body simulations and structural mechanics [22].

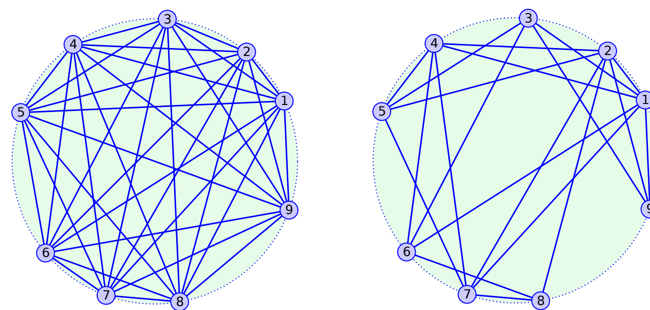


Figure 1.1.: Examples of dense and sparse connections, illustrated as graphs.

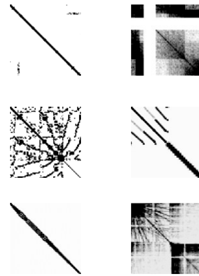


Figure 1.2.: Sparsity patterns of six example matrices from [22].

1.2. Sparse Kernels

Although any sparse structure can be represented as an equivalent dense structure with zero-weight links and computed as such, sparse representation presents a much smaller memory footprint and requires less computation, making it possible to scale to larger problem instances. In many cases, the part of the computation that operates directly on sparse data is similar and can be viewed as a pattern of computation and communication –a *sparse linear algebra kernel*– in the spirit of the Berkeley Dwarfs [3].

As sparse kernels arise in a variety of important domains, it is beneficial to be able to handle them in a fast and efficient manner. However, their idiosyncratic computational properties (discussed further in Section 2.3) align poorly with the design choices of general-purpose computers, which leads to underutilization of hardware resources when processing sparse linear algebra kernels on general-purpose CPUs. This makes sparse linear algebra a good match with the criteria for choice of acceleration domains specified in Part A Section 1.2, and justify its selection as a focus area in this thesis.

2. Background

2.1. Sparse Matrix–Vector Multiplication

Sparse Matrix–Vector Multiplication (SpMV) is a fundamental operation in sparse linear algebra, and an important computational kernel used in numerous areas of science and engineering. Perhaps the most well-known usage area for the kernel is the iterative solving of sparse linear systems that arise from the modeling of physical phenomena, where SpMV operations dominate the execution time [70]. The SpMV operation $\vec{y} = \mathbf{A} \cdot \vec{x}$ consists of multiplying an $m \times n$ sparse matrix $\mathbf{A}$ with NZ nonzero elements by a dense vector $\vec{x}$ of size n to obtain a result vector $\vec{y}$ of size m. A more specialized case of SpMV occurs when the vector $\vec{x}$ is also sparse, which is deemed *Sparse Matrix–Sparse Vector Multiplication (SpMSpV)*. In this thesis, SpMV is studied both as a stand-alone kernel and as a pattern or building block for other algorithms and applications.

The sparse matrix is commonly stored in a format which allows storing only the nonzero elements of the matrix. This can be done via a variety of different storage formats; here we only present the widely-adopted Compressed Sparse Row (CSR) format. The CSR format, which is exemplified in Figure 2.1b, consists of three one-dimensional arrays: a `values` array to store the nonzero values in row-major order, a `colind` array to store the column index of each nonzero, and a `rowptr` array which points to the start of each row in the `values` array. Many other formats also exist, which try to take advantage of particular shapes that may exist in the sparsity pattern [40, 65]. The vectors $\vec{x}$ and $\vec{y}$ are stored as standard one-dimensional arrays `x` and `y`, unless they are sparse, in which case a CSR-like representation may be used for them as well.

2.2. Breadth-First Search as Sparse Linear Algebra

The "matrices over semirings" concept [38] can be used to express graph processing operations as sparse linear algebra operations, thereby increasing the range of applications that a sparse linear algebra accelerator can address. We now give a brief overview of how this can be done for BFS. BFS is a key building block for exploring

$$\begin{bmatrix} 1.1 & 0 & 0 \\ 0 & 2.2 & 3.3 \\ 4.4 & 0 & 5.5 \end{bmatrix}$$

(a) Sparse matrix

```
rowptr={0 1 3 5}
values={1.1 2.2 3.3 4.4 5.5}
colind={0 1 2 0 2}
```

(b) CSR representation

$$\begin{bmatrix} 1.1 & 0 & 0 \\ 0 & 2.2 & 3.3 \\ 4.4 & 0 & 5.5 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ -1 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 4.4 \\ 11.0 \end{bmatrix}$$

(c) The SpMV operation.

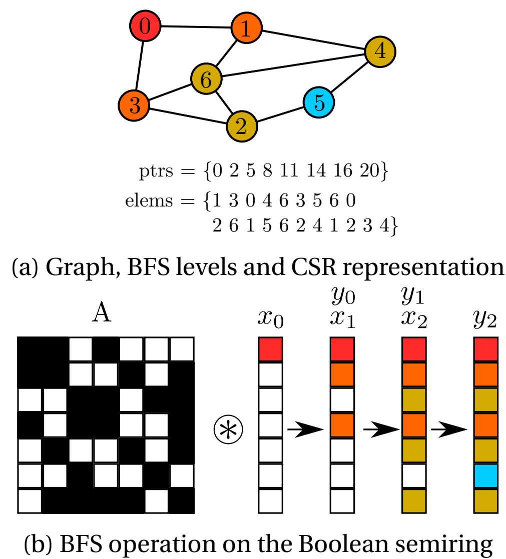
$$\begin{bmatrix} 1.1 & 0 & 0 \\ 0 & 2.2 & 3.3 \\ 4.4 & 0 & 5.5 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ -2.2 \\ 0 \end{bmatrix}$$

(d) The SpMSpV operation.

Figure 2.1.: Sparse matrix representations, SpMV and SpMSpV.

graphs, and is fundamental to a variety of graph metrics such as counting connected components, calculating graph diameter and radius [18]. Consider an undirected, unweighted graph of the form $G = (V, E)$ with sets of $|V|$ vertices (nodes) V and $|E|$ edges E. A BFS begins at a root node v_r contained within the largest connected component V_c and traverses each edge e_{rj} for every neighbor v_j. As such, the graph is traversed in *levels*, where all nodes at each level are explored before the next level is processed. Here, we consider the variant of the kernel that produces the distance array (`dist` of each visited node from the root node as the output. Large real-world networks are generally sparse, meaning that most nodes are not neighbors. To take advantage of this, the graph is typically stored in a sparse adjacency matrix form such as CSR as illustrated in Figure 2.2a.

The core idea in bridging BFS and sparse linear algebra is to substitute the number data type and the operators for multiplication and addition in linear algebra to express a variety of algorithms as matrix-vector operations. Specifically, we make use of the matrix-times-vector operation on the Boolean semiring to perform BFS. In practice, this operation "multiplies" a binary matrix and a binary vector, with the regular multiply and add operations substituted with the Boolean AND and OR operators, respectively. To disambiguate from regular matrix-vector multiply over real numbers, we use $\circledast$ to denote this operator. As illustrated in Figure 2.2b, each $y_t = A \circledast x_t$ operation



(a) Graph, BFS levels and CSR representation

(b) BFS operation on the Boolean semiring

Figure 2.2.: Two representations of the breadth-first search algorithm.

corresponds to a breadth-first step, and each result vector y_t is the representation of the visited nodes in the graph after step t. The matrix A in the operation is the adjacency matrix of the graph, while the initial input vector x_0 is initialized to all zeroes, except a single 1 at the location of the root node. The result vector y_t is used as the input vector x_{t+1} of the next step, which in turn generates more visited nodes in its result vector until the result converges (i.e., no more nodes can be visited).

2.3. Computational Properties

Sparse linear algebra is a well-studied acceleration target on a variety of architectures, including multi-core CPUs [70], General-Purpose Graphics Processing Units (GPGPUs) [10] and FPGAs [25]. The general computational properties of sparse linear algebra as reported by prior work can be summarized as follows:

- **Low arithmetic intensity.** Arithmetic intensity is a measure of how many computational operations are performed for every byte read from memory. Sparse computations have typically much lower arithmetic intensity [71] compared to dense ones, which makes their performance dependent on memory bandwidth. This is partially due to the way sparsity is represented (i.e., the positions of the

connections need to be specified, in addition to values), and partially due to sparsity itself (i.e., a read element will not be re-used many times, since it does not have many connections).

- **Sparsity-dependent, complex memory access patterns.** The sparsity pattern decides where memory accesses will take place. For structured sparsity, this will translate into strided accesses, whereas irregular sparsity will result in seemingly random accesses, which the memory system must cope with [68].
- **Dynamic behavior.** The sparsity pattern will also influence the dependencies and amount of work available in the computation. For instance, when traversing a small-world graph, the amount of work that can be done in parallel changes every iteration in a sparsity-dependent manner [8]. From a parallel computing perspective, this makes workload distribution and load balancing more difficult.
- **Dependent on large, external memory.** Sparse data can be very large in size and has tended to grow even larger over time, in part due to the Big Data trend [15]. Sparse computation therefore requires access to a large, external memory. Combined with irregular memory accesses and low arithmetic intensity, this creates a significant challenge in designing memory systems for sparse computation, since today's external memory devices are optimized for sequential accesses.
- **Iterative.** Most applications that make use of SpMV do so in an iterative fashion, multiplying the same sparse matrix with many different vectors. A typical example is iterative solving of sparse linear systems, where each iteration involves multiplying the same sparse matrix with a different vector [70]. This makes it possible to ameliorate the cost of preprocessing the sparse matrix across multiple iterations, if each iteration becomes faster as a result of the preprocessing.

Paper B1

An Energy Efficient Column-Major Backend for FPGA SpMV Accelerators

Yaman Umuroğlu and Magnus Jahre

Published in
Proceedings of the 2014 IEEE International Conference on Computer Design (ICCD)

B1. An Energy Efficient Column-Major Backend for FPGA SpMV Accelerators

Abstract. FPGAs are promising candidates for energy efficient acceleration of sparse matrix-vector multiplication (SpMV), a kernel with important applications in scientific computing and engineering. SpMV is characterized by matrix-dependent performance and high external memory bandwidth demands, which makes bandwidth utilization an important performance indicator. Existing FPGA SpMV accelerators focus on datapath optimizations instead of memory behavior, and exhibit matrix-dependent bandwidth utilization. In this work, we propose to decouple the SpMV computation and memory behavior, and focus on the backend which handles the latter. We describe a scalable backend architecture that exploits column-major traversal and interleaving to achieve high bandwidth utilization. Our experiments show that a single backend is able to sustain 96% of its assigned memory port bandwidth on average, and scales well with increased bandwidth by instantiating multiple parallel units. Compared to a baseline scheme, our scheme offers up to 1.5x higher DRAM power efficiency and up to 20% higher aggregate bandwidth. The results indicate that our scheme improves the average bandwidth utilization of existing FPGA SpMV accelerators by 15 to 77%.

B1.1. Introduction

With the phenomenon of dark silicon limiting the performance increase for microprocessors [57], specialized accelerator architectures are becoming a popular enhancement to computing systems. These accelerators can offer high energy efficiency and performance for the computational kernels they target. Sparse matrix-vector multiplication (SpMV) is an important computational kernel used in numerous areas of science and engineering. Perhaps the most well-known usage area for the kernel is

the iterative solving of sparse linear systems that arise from the modeling of physical phenomena, where SpMV operations dominate the execution time [70]. Despite its prevalence, SpMV is notorious for its poor performance on microprocessors, which can be mainly attributed to the following properties of the kernel [29]:

1. *Memory-to-computation ratio:* SpMV has a low ratio of floating point operations per byte since it performs $O(n^2)$ operations on $O(n^2)$ amount of data. This property makes the kernel memory-intensive and sensitive to memory bandwidth.

2. *Irregular memory accesses:* The most popular way of storing sparse matrices is to store only the nonzero entries, which causes irregular accesses to the vector and makes the performance sensitive to the distribution of nonzeroes of the matrix.

Additionally, the input data size used in the aforementioned applications can be very large and is typically stored in high-latency high-capacity external DRAM [30], which magnifies the effects of performance sensitivity to input patterns and bandwidth. These challenges have made SpMV a popular research topic, and numerous methods for increasing performance on general-purpose microprocessors have been proposed [66, 70].

With large memory bandwidth and latency-hiding multithreading techniques, GP-GPUs were shown to be a viable candidate for accelerating SpMV [11]. FPGA-based implementations are another strong candidate for efficient SpMV acceleration, as one can create a customized architecture that aligns well with the requirements of the kernel. Competitive floating-point performance, ample parallelism and high external bandwidth owing to the large pin count are additional motivators for FPGA SpMV accelerators. A number of FPGA SpMV accelerators have already been proposed [31, 34, 39, 75] and demonstrate the potential of FPGAs as an SpMV acceleration substrate. However, many of these works focus on developing efficient floating-point accumulators and not on memory behavior. As a result, they under-utilize the available external memory bandwidth and suffer from matrix-dependent performance.

The key observation in this work is that SpMV accelerators should be designed with focus on the main performance bottleneck: memory bandwidth. In order to achieve this, we decouple the computation from the memory behavior and choose column-major matrix traversal to ensure sequential access patterns on the input data. We analyze SpMV–DRAM interaction with regard to parallelism and data layout. Based on our observations, we propose an architecture that is able to exhibit consistently high bandwidth utilization and good scaling with increasing DRAM banks. Our experiments show that the proposed scheme can consume 96% of the memory bandwidth, with

improvements of up to 1.5x in power efficiency and 20% in bandwidth utilization compared to the baseline.

This work makes the following contributions:

- We introduce a component-based design strategy for SpMV accelerators, with a *frontend* handling the computation and a *backend* for the memory behavior to enable independent optimization.

- We analyze SpMV memory behavior with the compressed sparse column layout and show that SpMV exhibits matrix- and parallelism-dependent performance.

- We design a scalable backend architecture that utilizes an interleaved data layout, and present its experimental evaluation to show it achieves up to 96% in average DRAM bandwidth utilization and up to 1.95 GB/s·watt in power efficiency.

B1.2. Background

An SpMV operation $\vec{y} = \mathbf{A} \cdot \vec{x}$ consists of multiplying an $m \times n$ sparse matrix $\mathbf{A}$ with NZ nonzero elements by a dense vector $\vec{x}$ of size n to obtain a result vector $\vec{y}$ of size m. The sparse matrix is commonly stored in a format which allows storing only the nonzero elements of the matrix. This can be done via a variety of different storage formats; here we only present the widely-adopted Compressed Sparse Row (CSR) format [29], and its column-major counterpart Compressed Sparse Column (CSC). The CSR format, which is exemplified in Figure B1.1b, consists of three one-dimensional arrays: a `values` array to store the nonzero values in row-major order, a `colind` array to store the column index of each nonzero, and a `rowptr` array which points to the start of each row in the `values` array. The CSC format is the column-major equivalent of CSR and is shown in Figure B1.1c. The vectors $\vec{x}$ and $\vec{y}$ are stored as one-dimensional arrays `x` and `y`.

We define two basic data types, *ValueType* and *IndexType*, that are used in a CSC or CSR SpMV operation. *ValueType* is typically a double-precision floating-point value, representing each element of the *values* array of the matrix and vectors. *IndexType* is for expressing element indices or counts in the storage structures, such as the *rowptr* or *colind* arrays. A four-byte integer is sufficient for indexing matrices of up to 2^{32} nonzeroes. In line with previous work [29, 30], we assume *sizeof(ValueType)* = *ValSize* = 8 and *sizeof(IndexType)* = *IndSize* = 4 throughout the rest of this paper.

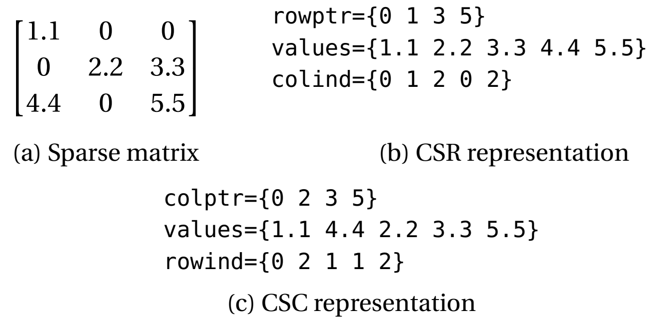


Figure B1.1.: CSR and CSC representations of a sparse matrix.

B1.2.1. Memory Access Patterns

The SpMV kernel exhibits different memory access patterns on its working set. The input matrix representation is accessed in a sequential manner, although there is no temporal reuse for this data due to the properties of the kernel [29]. The vector accesses are dependent on the traversal type: one vector can be accessed sequentially and with maximum temporal locality, whereas the other vector requires indirect, input-dependent memory accesses for every operation. This can be observed from the pseudocode in Figures B1.2a and B1.2b. For column-major traversal, the result vector `y` is accessed based on the `rowind` array data and `x` is accessed sequentially; the converse is true for row-major access. Since these indirect access patterns are unknown prior to runtime, they may be considered *random accesses* from the perspective of the accelerator. The column-major access pattern is exemplified in Figure B1.2c, where the sequential no-reuse `values` accesses, the sequential maximum-reuse `x` accesses and the pattern-dependent `y` accesses can be observed.

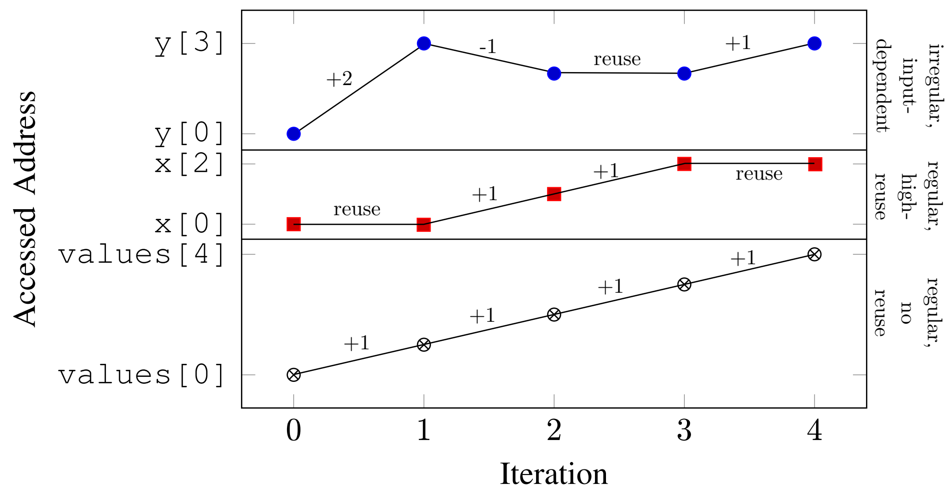
The working set of a SpMV operation, which includes the matrix and both vectors, can be very large. For FPGA SpMV accelerators, the input is assumed to be stored in external DRAM prior to execution [30, 75] to provide cost-effective capacity and bandwidth. However, since DRAM latencies can be large for random accesses, the vector to be random-accessed (`x` for CSR and `y` for CSC) is typically buffered on-chip. This on-chip buffer may be a hardware-controlled cache [30, 39] or algorithm-controlled block RAM buffers [34, 75].

```
for(i=0 to m-1)
  for(j=rowptr[i] to rowptr[i+1])
    y[i] += values[j] * x[colind[j]]
```

(a) CSR pseudocode

```
for(j=0 to n-1)
  for(i=colptr[j] to colptr[j+1])
    y[rowind[i]] += values[j] * x[j]
```

(b) CSC pseudocode



(c) CSC SpMV memory access schedule for Figure B1.1a.

Figure B1.2.: SpMV pseudocode and access patterns.

B1.2.2. DRAM Organization and Memory Controllers

DRAM accesses are critical for SpMV performance as well as a major contributor to system energy consumption [76], which warrants a closer examination of DRAM power and performance. Modern DRAM units have an internal three-dimensional structure that exhibits variable access latency and power use depending on the accessed memory locations. Each DRAM chip consists of a number of independent *banks*, each of which is organized as a two-dimensional array of *rows* and *columns*. Each bank has a *row buffer*, which is capable of holding a single row. Accesses to a particular memory location in a bank require that the corresponding row is already read into the row buffer [51], which is termed *row activation*.

Since row activation is costly in terms of performance and energy [44], minimizing the number of required row activations is important for achieving high DRAM performance and energy efficiency. The memory controller has the responsibility for mapping memory addresses to banks, rows and columns, as well as generating control signals to perform the requested memory accesses. Additionally, it may reorder memory requests to exploit row buffer locality, since ad-hoc data assignment and access scheduling can result in bank conflicts and decrease DRAM performance dramatically [51]. As an example, the latest generation of Xilinx FPGAs include reordering memory controllers that are claimed to reduce row activations by 16% for reads and writes to random locations [19]. SpMV and other kernels with irregular accesses can benefit from these capabilities.

B1.3. Decoupling SpMV Memory Behavior and Computation

Utilizing the external memory bandwidth available in the platform is essential for achieving high SpMV performance [39]. To analyze the memory behavior and identify the sources of inefficiency, we adopt a perspective that decouples memory and computation for SpMV accelerators. The proposed decomposition contains a *backend* for handling memory behavior and a *frontend* for computation, as illustrated in Figure B1.3.

In this form, an SpMV accelerator may be viewed as a dataflow architecture in which raw data is sent from the DRAM into the backend. The backend consumes and translates the raw data into a stream of *work units*, each of which consist of a matrix element index i, a dense vector value x_j and a matrix element value A_{ij}. The $frontend$ consumes this work unit stream by computing the matrix-vector product result vector, $y_i = \sum A_{ij} \cdot x_j$, which may be read directly from the frontend or written back to the

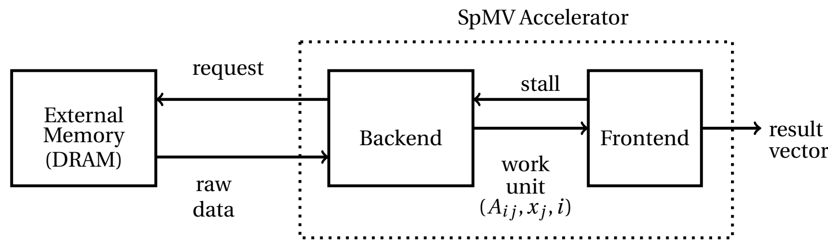


Figure B1.3.: Decoupled SpMV memory and computation.

DRAM. In addition to the flow of data, there is also control flow in the reverse direction. The backend is responsible for issuing requests to the external memory to produce the raw data stream. Similarly, the frontend can control the work unit stream from the backend by a stall signal if it is not able to accept any more work units.

To obtain full utilization of this pipeline, we note that each data consumer should be able to match the rate at which its input data is produced. In essence, the backend should be able to consume raw data as quickly as the external memory produces it. In turn, the frontend should be able to consume work units at the same rate the backend is producing them. This also implies that the rate of memory requests from the backend should be sufficient to saturate the external memory bandwidth, and that the frontend should be stall-free. A design with mismatched rates will have low utilization and waste area as well as energy. We can thus summarize the requirements for a high-utilization SpMV pipeline as a *bandwidth-saturating backend* and a matching *stall-free frontend.*

B1.4. Designing A Bandwidth-Saturating Backend

As will be discussed in Section B1.6, there has been a significant amount of research into stall-free FPGA SpMV frontends. However, the construction of SpMV backends has yet to be addressed in detail. In this work, we assume a stall-free SpMV frontend, and concentrate on developing a backend which is able to sustain the required data rate in order to keep the accelerator running at close-to-peak utilization regardless of the matrix structure.

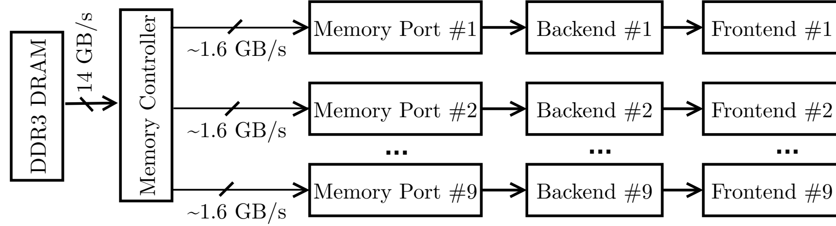


Figure B1.4.: Example of a saturating design for a 14 GB/s DDR3 memory with nine 64-bit 200 MHz backends.

B1.4.1. The Need for Backend Parallelism

As the clock frequency of FPGA user logic is low compared to CPUs and GPGPUs, FPGA-based computational accelerators typically contain many parallel computation units to achieve high throughput [75]. This can be interpreted as *frontend parallelism* in terms of the system presented in Figure B1.3. We emphasize that *backend parallelism* is also necessary not only to keep the frontends fed with data, but also to take advantage of increasing DRAM bandwidth. If the frontend and the memory bandwidth are scaled up without scaling the backend, the backend will become a bottleneck and the design will have decreased energy efficiency. The problem arises due to the mismatch between the DRAM data rate and the maximum FPGA user clock frequency. For instance, a DDR4 DRAM can perform up to 3.2 billion transfers per second, whereas FPGA user logic is typically clocked at 200–300 MHz. As extremely wide input ports make place-and-route operations difficult, a single SpMV backend is not suitable for consuming this much bandwidth. Therefore, we argue that FPGA SpMV accelerators should be designed with sufficient number of parallel backends to saturate the available bandwidth. Figure B1.4 shows an example of such a design, a theoretical FPGA SpMV accelerator connected to a PC3-14400 DIMM supplying 14 GB/s of bandwidth. This external bandwidth is made available to the backends through the memory ports of the FPGA memory controller. If each backend has a width of 64 bits and is operating at 200 MHz, it is able to consume approximately 1.6 GB/s, requiring 9 backends to consume the total available bandwidth.

B1.4.2. Choice of Traversal Order

Although matrix element traversal order can be thought of as a property of the computation performed by the frontend, it has important consequences on SpMV memory behavior. In terms of reading input data from the DRAM, column-major traversal

Table B1.1.: Access streams for the CSC format. $AvgColLen = \frac{NZ}{n}$ is the average number of nonzero elements per column.

Stream Type	Access Size	Access Period
Matrix Value	8 bytes	1
Matrix Row Index	4 bytes	1
Matrix Column Length	4 bytes	*AvgColLen*
Dense Vector Value	8 bytes	*AvgColLen*

offers several advantages compared to row-major. Notably, the input dense vector x is accessed in a sequential manner instead of irregularly, which makes it possible to exploit DRAM row buffer locality and enables simpler memory request generation logic. Additionally, it enables maximum reuse of dense vector elements, since a new dense vector element is not read before the old element is multiplied by each matrix element that requires it. Finally, if the result vector is completely buffered on-chip, column-major traversal does not require any writes to DRAM, thereby eliminating any bus turnaround delays.

Since our goal is to design a bandwidth-saturating backend, we chose column-major traversal for our design to shift the irregularity from input to output and take advantage of the DRAM-related benefits. We note that the concepts of backend/frontend decomposition, the need for backend parallelism and parallelism-aware DRAM data placement (which will be discussed in the following sections) apply to all SpMV accelerators in equal degree, regardless of traversal or storage format.

B1.4.3. Interleaved Column-Major Storage

We now discuss the interaction of SpMV access patterns with DRAM structure. When column-major traversal is desired, the CSC representation of the matrix is a natural match for the SpMV kernel. With this representation, column-major SpMV maps to four sequential access streams due to the *structure-of-arrays* nature of the format, as summarized in Table B1.1.

Each CSC backend introduces a new instance of each stream type. When multiple backends are requesting in parallel, the memory requests are serialized in the memory controller as there is a single control interface for the entire DRAM. Even though each stream is sequential in itself, serializing multiple sequential streams can create an access pattern that appears random and exhibits variable access latencies. As we show

```
colptr={0 2 3 5}            rowind={0 2 1 1 2}
values={1.1 4.4 2.2 3.3 5.5}      x={-3 -1 -9}
```

(a) Input data for a CSC SpMV operation

col 0 length	x [0]	values [0]	rowind [0]	values [1]	rowind [1]	col 1 length	x [1]	values [2]	rowind [2]
2	**-3**	**1.1**	**0**	**4.4**	**2**	**1**	**-1**	**2.2**	**1**

(b) Interleaved representation of the same input data for interleave factor 1

col 0 length	col 1 length	x [0]	x [1]	values [0]	rowind [0]	values [2]	rowind [2]	values [1]	rowind [1]
2	**1**	**-3**	**-1**	**1.1**	**0**	**2.2**	**1**	**4.4**	**2**

(c) Interleaved representation with interleave factor 2

Figure B1.5.: SpMV input data in regular and interleaved forms.

in Section B1.5, this pattern causes matrix- and parallelism-dependent bandwidth utilization and energy efficiency, even with a reordering memory controller.

To address this issue, we propose to store the inputs as a per-backend *array-of-structures*. In this interleaved representation, each backend requires only one stream of sequential memory accesses and can be assigned to a single DRAM bank to provide independent bandwidth. An example of this storage format is illustrated in Figure B1.5. Since this is a simple rearrangement of the data in CSC, the interleaving can be done on-the-fly while the data in CSC format is being transferred into the accelerator's DRAM memory.

We also note that different ways of interleaving are possible, which allows catering for the parallelism requirements of the frontend. We define the consecutive number of columns in the interleaved data as the *interleave factor*. For instance, a frontend that computes two columns in parallel (intra-column parallelism) would benefit from an interleaving factor of two, which is exemplified in Figure B1.5c.

B1.4.4. Proposed Backend Architecture

We now describe the internal architecture of our backend, whose most prevalent property is its *proactive* nature: instead of waiting for the frontend to issue work unit requests, the backend continuously pulls data from the DRAM as fast as possible, translates it into work units and sends them to the frontend. This allows long DRAM

burst accesses that can be continuously issued without waiting for the returned data, and is made possible by the interleaved storage in DRAM and the stall-free frontend.

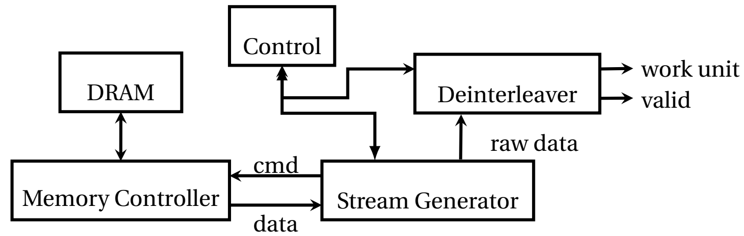
A top-level block diagram of the backend architecture is shown in Figure B1.6a. Before the backend operation starts, it is assumed that the interleaved SpMV data is present in the DRAM. When the backend is running, the stream generator continuously issues maximum-length burst read commands to the memory controller, which replies with a stream of interleaved data. This data is received by the deinterleaver, which turns the raw data into work units and forwards them to the frontend. The control module exposes registers for monitoring the busy status of the backend, configuring the number of columns and nonzeroes in the matrix, as well as the start address of the interleaved data. This information is used to configure how many iterations the stream generator and the deinterleaver perform.

B1.4.5. Deinterleaving Raw Data

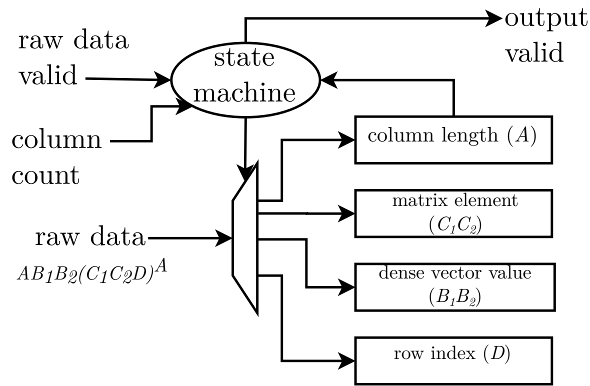
Since the entire SpMV input data is stored in interleaved form, the raw data arriving into the backend may contain a mixture of dense vector data, matrix element data, column length data and row index data. This data must be parsed and made available to the frontend in the form of work units. The deinterleaver, which consists of a state machine and several registers as depicted in Figure B1.6b, fulfills this responsibility.

In order to consume the entire raw data window of W bytes in a single clock cycle and parse it into its assorted fields, we use a state machine. This state machine is responsible for moving the current raw data into the relevant work unit registers depending on the current state, and determining the next state depending on the number of elements left in the column. State transitions occur only when valid raw data from the stream generator is available. The state machine is constructed by enumerating the possible combinations of fields that fit into the raw data window. The complexity of the state machine thus depends on the window size, the size of each field, and the interleave factor.

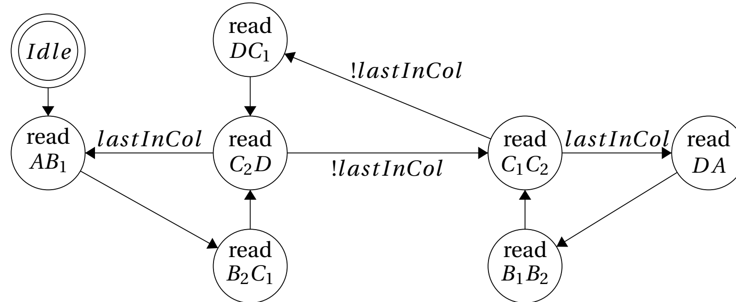
This design allows for high work unit throughput to the frontend, delivering a new work unit every T_{nz} cycles inside the same column, and an additional T_{col}-cycle delay while switching columns. The calculation of these values is shown in Equations B1.1 and B1.2. Assuming new raw data is delivered every cycle, the work unit throughput R from the backend can be computed as shown in Equation B1.3. Since $NZ \gg n$, the throughput will approach $R_{max} = \frac{W}{ValSize+IndSize}$ as the average column length $\frac{NZ}{n}$ becomes larger.



(a) Block diagram of the proposed backend



(b) Deinterleaver architecture



(c) Deinterleaver state machine for W=8

Figure B1.6.: Details of proposed backend architecture.

$$T_{nz} = \frac{sizeof(A_{ij}) + sizeof(i)}{W} = \frac{ValSize + IndSize}{W} \quad \text{(B1.1)}$$

$$T_{col} = \frac{sizeof(x_j) + sizeof(m)}{W} = \frac{ValSize + IndSize}{W} \quad \text{(B1.2)}$$

$$R = \frac{WorkUnits}{TotalCycles} = \frac{NZ}{NZ \cdot T_{nz} + n \cdot T_{col}} \quad \text{(B1.3)}$$

To give an example, we assume an interleave factor of one, and a window size $W = 8$ bytes. We define the regular expression $AB_1B_2(C_1C_2D)^A$ which represents one column of interleaved data. Each symbol is four bytes; the column length A and row index D are encoded by one symbol, while the double-precision dense vector value B_1B_2 and matrix value C_1C_2 are encoded with two symbols. The resulting state machine is shown in Figure B1.6c. The peak throughput for this backend design is $R_{max} = \frac{8}{8+4} = 0.66$ work units per cycle.

B1.5. Experimental Evaluation

B1.5.1. FPGA Implementation

To test the feasibility of the design in real hardware, we implemented our backend in VHDL and evaluated its performance on an Avnet Spartan-6 LX16 evaluation kit. This board contains 64 MB of LPDDR external memory, which can deliver 800 MB/s of external memory bandwidth. The block diagram of the implemented architecture and relevant data rates are shown in Figure B1.7. A MicroBlaze soft processor was used to initialize the matrix data in the LPDDR and for interfacing the backened. The work unit throughput is measured via a dummy frontend connected to the deinterleaver output. Due to the low external memory bandwidth on this platform, a single backend with $W = 8$ running at 100 MHz was sufficient to saturate the bandwidth (800 MB/s = 100 MHz $\cdot$ 64 $\frac{\text{bits}}{\text{cycle}}$).

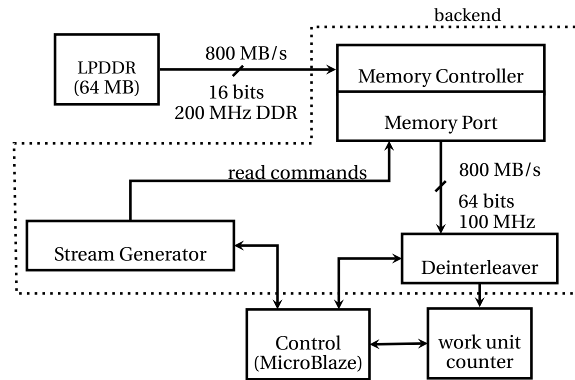


Figure B1.7.: Implementation on Avnet S6LX16 evaluation kit.

Table B1.2.: Logic resources and power for backend components.

Component	Average Power	Logic Slices	Device Occupation
Deinterleaver	4.4 mW	142	6%
Stream Generator	0.6 mW	63	3%
Memory Controller	216 mW	153	7%

Power and Area

We provide synthesis results from XST and average power estimates from XPower in Table B1.2. The backend (including the memory controller) occupies 16% of the logic resources of this small FPGA, and consumes less than 250 mW on average, which is similar to the LPDDR power consumption. For high-bandwidth DDR modules, the power consumption of the backend will likely be dwarfed by the DRAM power, which can be several watts (Figure B1.9c). Finally, we emphasize that the choice of a 100 MHz clock does not reflect the maximum achievable frequency (which is about 200 MHz for this FPGA), but rather the minimum frequency to saturate the external memory bandwidth. The synthesis results indicate that the design is fast, compact and consumes little power, all of which contribute to scaling to multiple backends.

Testing on real matrices

The implementation was tested with matrices taken from Tim Davis' collection [22], representing several different problem types, sizes, patterns and sparsities. For each matrix, we compute the *sustained memory bandwidth* by dividing the number of bytes transferred by the elapsed time, and the *sustained work unit throughput* (WUT) by dividing the number of nonzeroes in the matrix by the elapsed time. We compare these metrics to their theoretical peak values, which are 800 MB/s for the bandwidth and $R_{max} = \frac{8}{8+4} = 0.66$ work units per cycle for the WUT, as per Equation B1.3. The tested matrices and corresponding results can be found in Table B1.3.

In terms of sustained DRAM bandwidth, our backend consistently displays high utilization across all matrices, averaging at **96%** of the peak. The WUT is more dependent on the matrix structure due to the temporal reuse properties of SpMV. Figure B1.9 displays the trends for sustained memory bandwidth and WUT with increasing average column length. The minimum WUT is observed for matrix 1, which is extremely sparse (a single nonzero per column). As column length and dense vector values make up half the input data in this case, a throughput of half the peak is expected. This may be alleviated through alternative matrix representations, or the increased slack may be exploited by the frontend for RAW hazard avoidance. The WUT quickly increases with

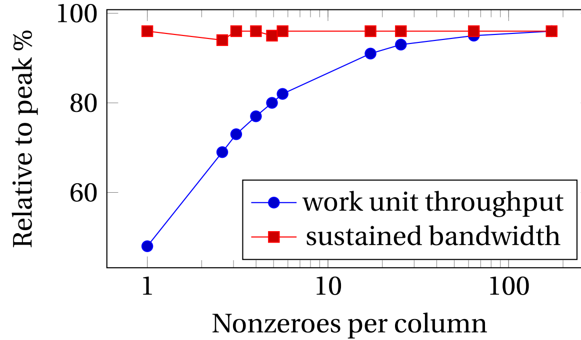


Figure B1.8.: Backend performance with increasing column length.

increasing average column length, and matrices with more than 5 nonzeroes per column exhibit over 80% of the peak work unit throughput.

B1.5.2. Performance with Parallel Accesses

As a single backend can nearly saturate the 800 MB/s bandwidth on our FPGA platform, we ran simulation-based experiments with higher memory bandwidth (12.8 GB/s) to evaluate the performance and energy efficiency of our scheme with multiple backends. To illustrate the benefits of our scheme, we also compare it against CSC without any interleaving. Since our experiments show that DRAM power dominates the total power consumption for the backend operation, we focus on DRAM power and performance. We use DRAMSim2 by Rosenfeld et al. [52], a cycle-accurate memory system simulator, with the parameters listed in Table B1.4.

For our scheme, we assign backends to DRAM banks in a round-robin manner. For the baseline implementation, we assume that CSC matrix data is laid out as-is (sequentially) in memory, with the dense vector data placed after the matrix, and with each backend assigned an equal number of columns. The bank bits are moved to after the column bits to provide more bank parallelism without explicit data-to-bank assignment, which allows a fairer comparison. We note that the simulated memory controller reorders requests to maximize row buffer hits, similar to the reordering memory controllers found in newer

Table B1.3.: Matrices for evaluation, and performance results for a single backend.

	Matrix Properties						Sustained Performance %	
#	Matrix	Dimensions	Nonzeroes	NZ/col	Sparsity %	Type of Problem	Bandwidth	Work Units
1	bcsstm21	3600x3600	3600	1.0	0.03	structural	96	48
2	p0548	176x724	1887	2.6	0.36	linear programming	94	69
3	webbase-1M	1000005x1000005	3105536	3.11	0.0003	web connectivity matrix	96	73
4	mc2depi	525825x525825	2100225	3.99	0.001	Markov model	96	77
5	cryg2500	2500x2500	12349	4.9	0.20	materials	95	80
6	scircuit	170998x170998	958936	5.61	0.003	circuit simulation	96	82
7	ex36	3079x3079	53099	17.2	0.56	computational fluid dynamics	96	91
8	shipsec1	140874x140874	3568176	25.33	0.018	ship section detail	96	93
9	cant	62451x62451	4007383	64.17	0.103	FEM cantilever	96	95
10	psmigr_1	3140x3140	543160	173.0	5.51	economics	96	96

Table B1.4.: Simulation parameters for DRAMSim2.

Parameter	Value
Memory module	PC3-12800 DDR3 DIMM
Total capacity	2048 MB
Organization	1 rank, 8 banks, 64 bit width
Row buffer policy	open-page
Row hit limit	unlimited
Command queue	32 entries
Peak Bandwidth	12.8 GB/s
Address Map (interleaved)	3 bits bank : 14 bits row : 11 bits column
Address Map (baseline)	14 bits row : 11 bits column : 3 bits bank

FPGAs [19]. For both schemes, the DRAM command queue is always kept full with requests, which is easy to implement in hardware for our single-stream scheme, but may present difficulties for the four-stream matrix-dependent baseline scheme. The results are summarized in Figure B1.9.

Aggregate Bandwidth

For up to sixteen backends, the average aggregate bandwidth of our method remains close to 90% of the device peak, as illustrated in Figure B1.9a. We consistently outperform the baseline, with up to 20% higher average aggregate bandwidth for sixteen backends. With an in-order memory controller, the baseline CSC scheme is likely to exhibit worse performance, whereas our scheme would maintain its performance since it already exploits the row buffer locality in each bank as much as possible.

Bandwidth Variability

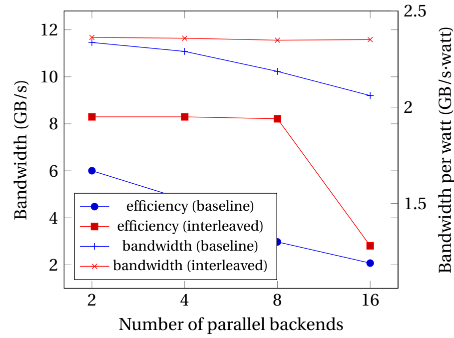
Figure B1.9b plots the minimum observed performance from our scheme (16 backends) and the sustained bandwidth for different matrices and backend counts for the baseline. In contrast to our scheme, which exhibits consistent

sustained bandwidth utilization across all matrices, the baseline has considerable performance variation for different matrices and backend counts, especially for the larger matrices 3, 4, 6 and 8. As there is a fixed mapping of memory addresses to DRAM banks, the matrix structure and parallel access patterns determine the number of row activations and bank conflicts for the baseline method. We note that the reordering capabilities of the memory controller may lessen the impact of access patterns on the baseline performance variability.

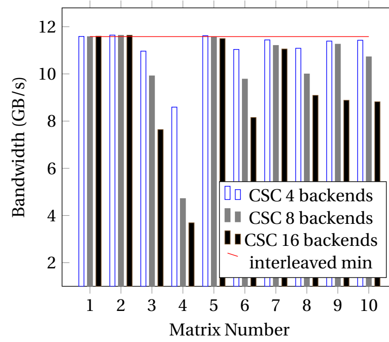
Power Efficiency

To compare the power efficiency of our scheme to the baseline, we present the average bandwidth per watt in Figure B1.9a. Our method offers 1.5x higher energy efficiency compared to the baseline as long as there is a single backend per DRAM bank. To perform a deeper analysis, we present the DRAM power breakdown in Figure B1.9c. For our method, the infrequency of row activations is evident from the level of activate/precharge power, which is less than 0.05 W. In contrast, the baseline requires an average of 1.67 W for two backends, with almost linear increase in the required activate/precharge power when the number of backends double. The majority of total DRAM power for our scheme (about 70%) is spent for burst accesses to the row buffer. With sixteen backends, there are two backends per DRAM bank. In this case, although the memory controller is able to avoid performance penalties by reordering, bank conflicts increase significantly as evidenced by the steep increase in the activate/precharge power and a drop in power efficiency.

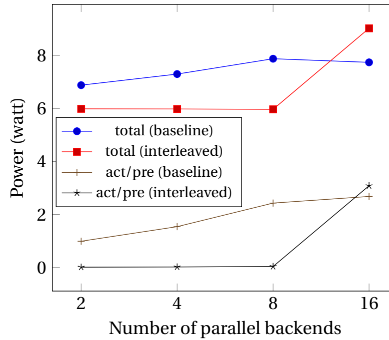
These results suggest that our backend scheme is able to maintain high bandwidth utilization and power efficiency with increased parallelism, as long as at most two backends are assigned to a single memory bank. We also outperform the baseline CSC scheme by up to 20% in terms of average aggregate bandwidth, up to 1.5x in power efficiency, and offer consistent performance regardless of the matrix structure.



(a) Average performance and power efficiency



(b) Performance variability



(c) DRAM activate/precharge power

Figure B1.9.: Comparison of parallel baseline CSC and interleaved backend operation.

B1.6. Related Work

B1.6.1. SpMV and irregular memory behavior

The irregular, pattern-dependent memory behavior of the SpMV kernel is well known and has been thoroughly investigated in the context of microprocessors with hardware caches [29]. Researchers have proposed numerous of techniques to optimize SpMV performance on microprocessors, which have been compiled into an auto-tuning library by Vuduc et al. [66]. In the general context of irregular memory behavior and DRAM interaction, memory controllers with automatic [51] or programmable [74] reordering capabilities have been proposed. Spinean et al. [55] propose a access reordering unit which can be used to enhance in-order memory controllers to offer performance and energy efficiency improvements. Our work offers additional benefits on top of or alternative to the reordering capabilities of memory controllers by changing the data layout itslelf, similar to the work by Matam et al. [44] which proposes a DRAM-aware layout for dense matrices, and offers up to 1.6x improvement in energy efficiency.

B1.6.2. Existing FPGA SpMV accelerators

Numerous FPGA SpMV accelerators were previously proposed, but to our knowledge, this work is the first to propose a decoupled SpMV backend. We restrict our review to existing work which directly interfaces DRAM and implicitly describes a backend. Table B1.5 summarizes the reported bandwidth utilization and storage schemes in previous work. Several of these works use alternative representations to compress matrix data. Jain-Mendon et al. [34] propose a blocked streaming storage format (VCDB) to remove indirect memory accesses. Kestur et al. [39] utilize the bit-level manipulation capabilities of the FPGA to compress the matrix nonzero pattern using delta encoding. Our scheme does not include any compression or blocking and can be further enhanced with these techniques to offer higher work unit throughput to the frontend. In terms of rearranging SpMV data to match DRAM structure, our work is similar to that of Gregg et al. [30], who use column-major traversal and one DRAM

Table B1.5.: Comparison with existing SpMV backends.

	DRAM Bandw. Util.		Backend	
Work	Average	Min-Max	Traversal[1]	Storage
Jain-Mendon et al. [34]	70%	17-94%	BL	VCDB
Kestur et al. [39]	14%	4-19%	RM	CVBV
Halstead et al. [31]	48%[2]	10-72%	RM	CSR
Gregg et al. [30]	71%[2]	13-74%	CM	SPAR
Zhang et al. [75]	76%	64-98%	RM	semi-interleaved
This work	91%	91-96%	CM	interleaved

[1] BL: blocked, RM: row-major, CM: column-major
[2] Bandwidth utilization estimated from reported performance

chip per parallel processing element, and Zhang et al. [75] who interleave row-major values and column indices. However, the experimental results indicate that our scheme outperforms the previous schemes in terms of average bandwidth utilization and performance consistency across different matrices. In constrast with previous work, we also provide power efficiency results from our scheme.

B1.6.3. Stall-Free SpMV Frontends

A significant body of work targets SpMV frontend development, where the goal is to construct a stall-free multiply-accumulate unit. A major hindrance towards this goal is the floating point units in FPGAs; in order to achieve high clock speeds, it is necessary to pipeline these units, which results in read-after-write (RAW) hazards during accumulation [30]. Nevertheless, numerous designs for single-cycle accumulation of an arbitrary number of arbitrary-sized consecutive groups have been proposed and the state of the art is close to achieving this goal. Notable works here include single- and double-strided adders [77] and FAAC [56], which can be utilized to create a stall-free frontend for row-major traversal. Column-major traversal and accumulation of interleaved groups has been studied to a lesser extent. Since column-major traversal switches between different rows, it offers a natural way of RAW hazard avoidance. Hazards may still exist, but Gregg et al. [30] report that hazard-related stalls are few as long

as the number of stages in the floating-point adder is fewer than 10.

B1.7. Conclusion and Future Work

In this paper we have presented a memory-centric view of the SpMV kernel and proposed a novel SpMV FPGA backend and storage scheme. Our backend takes into account the internal organization of DRAMs and utilizes an interleaved SpMV representation to achieve high DRAM bandwidth and power efficiency. The experimental results from a Spartan-6 FPGA board indicate that the proposed design is compact and fast, and is able to achieve close to peak DRAM bandwidth on its assigned memory port, regardless of the matrix structure. We have also demonstrated that our scheme exhibits good scaling behavior and outperforms the CSC baseline, using a cycle-accurate DRAM simulator. These properties make our design a competitive alternative to the naive backends found in previously-proposed SpMV accelerators. Future work will include developing an efficient column-major frontend and evaluating a complete FPGA SpMV accelerator design on a more powerful platform.

Acknowledgements

The authors would like to extend their gratitude to the anonymous reviewers, Nikita Nikitin, Juan M. Cebrian and Benjamin Bjornseth for their valuable comments on the earlier versions of this work.

Paper B2

A Vector Caching Scheme for Streaming FPGA SpMV Accelerators

Yaman Umuroğlu and Magnus Jahre

Published in
Proceedings of the 2015 Symposium on Applied Reconfigurable Computing
(ARC)

B2. A Vector Caching Scheme for Streaming FPGA SpMV Accelerators

Abstract. The sparse matrix–vector multiplication (SpMV) kernel is important for many scientific computing applications. Implementing SpMV in a way that best utilizes hardware resources is challenging due to input-dependent memory access patterns. FPGA-based accelerators that buffer the entire irregular-access part in on-chip memory enable highly efficient SpMV implementations, but are limited to smaller matrices due to on-chip memory limits. Conversely, conventional caches can work with large matrices, but cache misses can cause many stalls that decrease efficiency. In this paper, we explore the intersection between these approaches and attempt to combine the strengths of each. We propose a hard-ware-software caching scheme that exploits preprocessing to enable performant and area-effective SpMV acceleration. Our experiments with a set of large sparse matrices indicate that our scheme can achieve nearly stall-free execution with average 1.1% stall time, with 70% less on-chip memory compared to buffering the entire vector. The preprocessing step enables our scheme to offer up to 40% higher performance compared to a conventional cache of same size by eliminating cold miss penalties.

B2.1. Introduction

Increased energy efficiency is a key goal for building next-generation computing systems that can scale the "utilization wall" of dark silicon [57]. A strategy for achieving this is accelerating commonly encountered kernels in applications. Sparse Matrix – Vector Multiplication (SpMV) is a computational kernel

widely encountered in the scientific computation domain and frequently constitutes a bottleneck for such applications [70]. Analysis of web connectivity graphs [22] can require adjacency matrices that are very large and sparse, with a tendency to grow even bigger due to the important role they play in the Big Data trend.

A defining characteristic of the SpMV kernel is the *irregular memory access pattern* caused by the sparse storage formats. A critical part of the kernel depends on memory reads to addresses that correspond to non-zero element locations of the matrix, which are only known at runtime. The kernel is otherwise characterized by little data reuse and large per-iteration data requirements [70], which makes the performance memory-bound. Storing the kernel inputs and outputs in high-capacity high-bandwidth DRAM is considered a cost-effective solution [30]; however, the burst-optimized architecture of DRAM constitutes an ever-growing "irregularity wall" in the quest for enabling efficient SpMV implementations.

Recently, there has been increased interest in FPGA-based acceleration of computational kernels. The primary benefit from FPGA accelerators is the ability to create customized memory systems and datapaths that align well with the requirements of each kernel, enabling stall-free execution (termed *streaming acceleration* in this paper). From the perspective of the SpMV kernel, the ability to deliver high external memory bandwidth owing to high pin count and dynamic (run-time) specialization via partial reconfiguration are attractive properties. Several FPGA implementations for the SpMV kernel have been proposed, either directly for SpMV or as part of larger algorithms like iterative solvers [25, 27, 28], some of which present order-of-magnitude better energy efficiency and comparable performance to CPU and GPGPU solutions thanks to streaming acceleration. These accelerators tackle the irregular access problem by buffering the entire random-access data in *on-chip memory (OCM)*. Unfortunately, this *buffer-all strategy* is limited to SpMV operations where the random-access data can fit in OCM, and therefore not suitable for very large sparse matrices.

To address this problem, we propose a specialized vector caching scheme for area-efficient SpMV accelerators that can target large matrices while still preserving the streaming acceleration property. Using the canonical cold-capacity-conflict cache miss classification, we examine how the structure of a sparse

$$\begin{bmatrix} 1.1 & 0 & 0 \\ 0 & 2.2 & 3.3 \\ 4.4 & 0 & 5.5 \end{bmatrix}$$

```
colptr={0 2 3 5}
values={1.1 4.4 2.2 3.3 5.5}
rowind={0 2 1 1 2}
```

```
for(j=0 to n-1)
  for(i=colptr[j] to colptr[j+1])
    y[rowind[i]] += values[j] * x[j]
```

Figure B2.1.: Sparse matrix in CSC representation and SpMV pseudocode. The random-access clause to y is highlighted.

matrix relates to each category and how misses can be avoided. By exploiting preprocessing (which is quite common in GPGPU and CPU SpMV optimizations) to specialize for the sparsity pattern of the matrix we show that streaming acceleration can be achieved with significantly smaller area for a set of test matrices. Our experiments with a set of large sparse matrices indicate that our scheme achieves the best of both worlds by increasing performance by 40% compared to a conventional cache while at the same time using 70% less OCM than the buffer-all strategy. The contributions of this work are four-fold. First, we describe how the structure of a sparse matrix relates to *cold*, *capacity* and *conflict* misses in a hardware cache. We show how cold misses to the result vector can be avoided by marking row start elements in column-major traversal. We propose two methods of differing accuracy and overhead for estimating the required cache depth to avoid all capacity misses. Finally, we present an enhanced cache with cold miss skip capability, and demonstrate that it can outperform a traditional cache in performance and a buffer-all strategy in area.

B2.2. Background and Related Work

B2.2.1. The SpMV Kernel and Sparse Matrix Storage

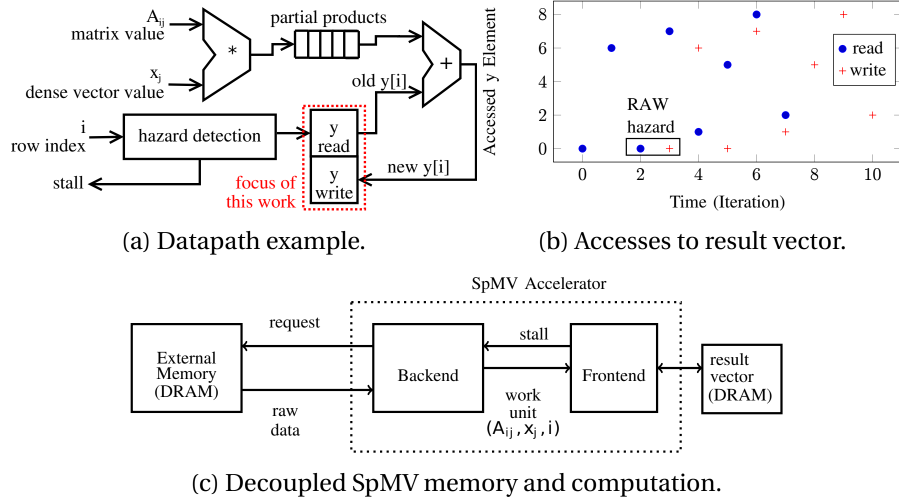
The SpMV kernel $\vec{y} = \mathbf{A} \cdot \vec{x}$ consists of multiplying an $m \times n$ sparse matrix $\mathbf{A}$ with NZ nonzero elements by a dense vector $\vec{x}$ of size n to obtain a result vector $\vec{y}$ of size m. The sparse matrix is commonly stored in a format which allows storing only the nonzero elements of the matrix. Many storage formats for sparse matrices have been proposed, some of which specialize on particular sparsity patterns, and others suitable for generic sparse matrices. In this paper, we will assume an FPGA SpMV accelerator that uses column-major sparse matrix traversal (in line with [25, 30, 62]) and an appropriate storage format

such as Compressed Sparse Column (CSC). Column-major is preferred over row-major due to the advantages of maximum temporal locality on the dense vector access and the natural C-slow-like interleaving of rows in floating point multiplier pipelines, enabling simpler datapaths [25]. Additionally, as we will show in Section B2.3.2 it allows bypassing cold misses, which can contribute significantly to performance. Figure B2.1 illustrates a sparse matrix, its representation in the CSC format, and the pseudocode for performing column-major SpMV. We use the `variable` notation to refer to CSC SpMV data such as `values` and `colptr`. As highlighted in the figure, the result vector `y` is accessed depending on the `rowind` values, causing the random access patterns that are central to this work.

B2.2.2. FPGA SpMV Accelerators and Result Vector Access

The datapath of a column-major SpMV accelerator is a multiply-accumulator with feedback from a random-access memory, as illustrated in Figure B2.2a. New partial products are summed into the corresponding element of the result vector, which can give rise to read-after-write (RAW) hazards due to the latency of the adder, as shown in Figure B2.2b. Addressing this requires a read operation to `y[i]` to be delayed until the writes to `y[i]` are completed, which is typically avoided by stalling the pipeline or reordering the elements.

With growing sparse matrix sizes and typically double-precision floating point arithmetic, the inputs of the SpMV kernel can be very large. Combined with the memory-bound nature of the kernel, this requires high-capacity high-bandwidth external memory to enable competitive SpMV implementations. Existing FPGA SpMV accelerators [25, 27, 28, 30] used DRAM as a cost-effective option for the storing the SpMV inputs and outputs, which is also our approach in this work. These designs typically address the random access problem by buffering the entire random-access vector in OCM [25, 27, 28]. Random accesses to the vector are thus guaranteed to be serviced with a small, constant latency. Unfortunately, this limits the maximum sparse matrix size that can be processed with the accelerator. To deal with `y` vectors larger than the OCM size while avoiding DRAM random access latencies, Gregg et al. [30] proposed to store the result vector in high-capacity DRAM and used a small direct-mapped cache. They also observed that cache misses present a significant penalty, and



(a) Datapath example.

(b) Accesses to result vector.

(c) Decoupled SpMV memory and computation.

Figure B2.2.: A column-major FPGA SpMV accelerator design.

proposed reordering the matrix and processing in cache-sized chunks to reduce miss rate. However, this imposes significant overheads for large matrices. In contrast, our approach does not modify the matrix structure; rather, it extracts information from the sparse matrix to reduce cache misses, which can be combined with reordering for greater effect. Prior work such as [58] analyzed SpMV cache behavior on microprocessors, but includes non-reusable data such as matrix values and requires probabilistic models. FPGA accelerators can exhibit deterministic access patterns for each sparse matrix, which our scheme exploits for analysis and preprocessing.

To concentrate on the random access problem, we base our work on a decoupled SpMV accelerator architecture [62], which defines a *backend* interfacing the main memory and pushing *work units* to the *frontend*, which handles the computation. Our focus will be on the random-access part of the frontend. Since we would like the accelerator to support larger result vectors that do not fit in OCM, we add DRAM for storing the result vector, as illustrated in Figure B2.2c.

B2.2.3. Sparse Matrix Preprocessing

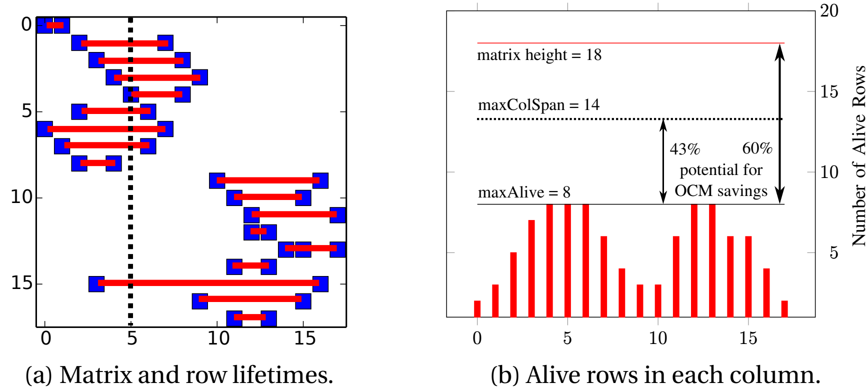
The memory behavior and performance of the SpMV kernel is dependent on the particular sparse matrix used, necessitating a preprocessing step at runtime for optimization. Fortunately, algorithms that make heavy use of SpMV tend to multiply the same sparse matrix with many different vectors, which enables ameliorating the cost of preprocesing across speed-ups in each SpMV iteration. This preprocessing can take many forms [59], including permuting rows/columns to create dense structure, decomposing into predetermined patterns, mapping to parallel processing elements to minimize communication and so on. We also adopt a preprocessing step in our scheme to enable optimizing for a given sparse matrix, but unlike previous work, our preprocessing stage produces information to enable specialized cache operation instead of changing the matrix structure.

B2.3. Vector Caching Scheme

To tackle the memory latency problem while accessing the result vector from DRAM, we buffer a portion of the result vector in OCM and use a hardware-software cooperative *vector caching scheme* that enables per-matrix specialization. This scheme will consist of a runtime *preprocessing step*, which will extract the necessary information from the sparse matrix for efficient caching including the required cache size, and *vector cache* hardware which will use this information. Our goal is to shrink the OCM requirements for the vector cache while avoiding stalls for servicing requests from main memory.

B2.3.1. Row Lifetime Analysis

To relate the vector cache usage to the matrix structure, we start by defining a number of structural properties for sparse matrices. First, we note that each row has a strong correspondence to a single result vector element, i.e., `y[i]` contains the dot product of row `i` with `x`. The period in which `y[i]` is used is solely determined by the period in which row `i` accesses it. This is the key



(a) Matrix and row lifetimes.

(b) Alive rows in each column.

Figure B2.3.: Example matrix Pajek/GD01_b and row lifetime analysis.

observation that we use to specialize our vector caching scheme for a given sparse matrix.

Calculating `maxAlive`: For a matrix with column-major traversal, we define the *aliveness interval* of a row as the column range between (and including) the columns of its first and last nonzero elements, and will refer to the interval length as the *span*. Figure B2.3a illustrates the aliveness intervals as red lines extending between the first and last non-zeroes of each row. For a given column `j`, we define a set of rows to be *simultaneously alive* in this column if all of their aliveness intervals contain `j`. The number of alive rows for a given column is the maximum size of such a set. Visually, this can be thought of as the number of aliveness interval lines that intersect the vertical line of a column. For instance, the dotted line corresponding to column 5 in Figure B2.3a intersects 8 intervals, and there are 8 rows alive in column 5. Finally, we define the *maximum simultaneously alive rows* of a sparse matrix, further referred to as `maxAlive`, as the largest number of rows simultaneously alive in any column of the matrix. Incidentally, `maxAlive` is equal to 8 for the matrix given in Figure B2.3a – though the alive rows themselves may be different, no column has more than 8 alive rows in this example.

Calculating `maxColSpan`: Calculating `maxAlive` requires preprocessing the matrix. If the accelerator design is not under very tight OCM constraints, it may be desirable to estimate `maxAlive` instead of computing the exact value in order to reduce the preprocessing time. If we define aliveness interval and

span for columns as was done for rows, the largest column span of the matrix `maxColSpan` provides an upper bound on `maxAlive`. The column 3 in Figure B2.3a has a span of 14, which is `maxColSpan` for this matrix.

B2.3.2. Avoiding Vector Cache Misses

We now use the canonical cold/capacity/conflict classification to break down cache misses into three categories and explain how accesses to the result vector relate to each category. For each category, we will describe how misses can be related to the matrix structure and avoided where possible.

Cold Misses: Cold (compulsory) misses occur when a vector element is referenced for the first time, at the start of the aliveness interval of each row. For matrices with very few elements per row, cold misses can contribute significantly to the total cache misses. Although this type of cache miss is considered unavoidable in general-purpose caching, a special case exists for SpMV. Consider the column-major SpMV operation $y = Ax$ where the y vector is random-accessed using the vector cache. The initial value of each y element is zero, and is updated by adding partial sums for each nonzero in the corresponding matrix row. If we can distinguish cold misses from the other miss types at runtime, we can avoid them completely: a cold miss to a y element will return the initial value, which is zero[1]. Recognizing misses as cold misses is critical for this technique to work. We propose to accomplish this by introducing a *start-of-row bit* marked during preprocessing, as described in Section B2.3.3.

Capacity Misses: Capacity misses occur due to the cache capacity being insufficient to hold the SpMV result vector working set. Therefore, the only way of avoiding capacity misses is ensuring that the vector cache is large enough to hold the working set. Caching the entire vector (the buffer-all strategy) is straightforward, but is not an accurate working set size estimation due to the sparsity of the matrix. While methods exist to attempt to reduce the working set of the SpMV operation by permuting the matrix rows and columns, they are outside the scope of this paper. Instead, we will concentrate on how the working set size can be estimated. This estimation can be used to reconfigure

[1] The more general SpMV form $y = Ax + b$ can be easily implemented by adding the dense vector b after $y = Ax$ is computed.

```
function PreprocessMaxAlive(CSCMatrix A)
    Q ← priorityQueue(), currentAlive ← 0, maxAlive ← 0
    R ← toCSR(A)
    for i ← 0..m − 1 do
        start ← R.colind[R.rowptr[i]]; end ← R.colind[R.rowptr[i + 1] − 1]
        markRowStart(A, start, i)
        Q.insert(prio = start, elem = +1); Q.insert(prio = end, elem = −1)
    end for
    while !Q.empty() do
        currentAlive ← currentAlive + Q.pop(); maxAlive ← max(currentAlive, maxAlive)
    end while
    return maxAlive
end function
```

Algorithm B2.1: Finding `maxAlive` and marking row starts.

the FPGA SpMV accelerator to use less OCM, which can be reallocated for other components. In this work, we make the assumption that a memory location is in the working set if it will be reused at least once to reap all the caching benefits. Thus, the cache must have a capacity of at least `maxAlive` to avoid all capacity misses. This requires the computation of `maxAlive` during the preprocessing phase. If OCM constraints are more relaxed, the `maxColSpan` estimation described in Section B2.3.1 can be used instead. Figure B2.3b shows the row lifetime analysis for the matrix in Figure B2.3a and how different estimations of the required capacity yield different OCM savings compared to the buffer-all strategy.

Conflict Misses: For the case of an SpMV vector cache, conflict misses arise when two simultaneously alive vector elements map to the same cache line. This is determined by the nonzero pattern, number of cachelines and the chosen hash function. Assuming that the vector cache has enough capacity to hold the working set, avoiding conflict misses is an associativity problem. Since content-associative memories are expensive in FPGAs, direct-mapped caches are often preferred. As described in Section B2.4.2, our experiments indicate that conflicts are few for most matrices even with a direct-mapped cache, as long as the cache capacity is sufficient. Techniques such as victim caching [36] can be utilized to decrease conflict misses in direct-mapped caches, though we do not investigate their benefit in this work.

B2.3.3. Preprocessing

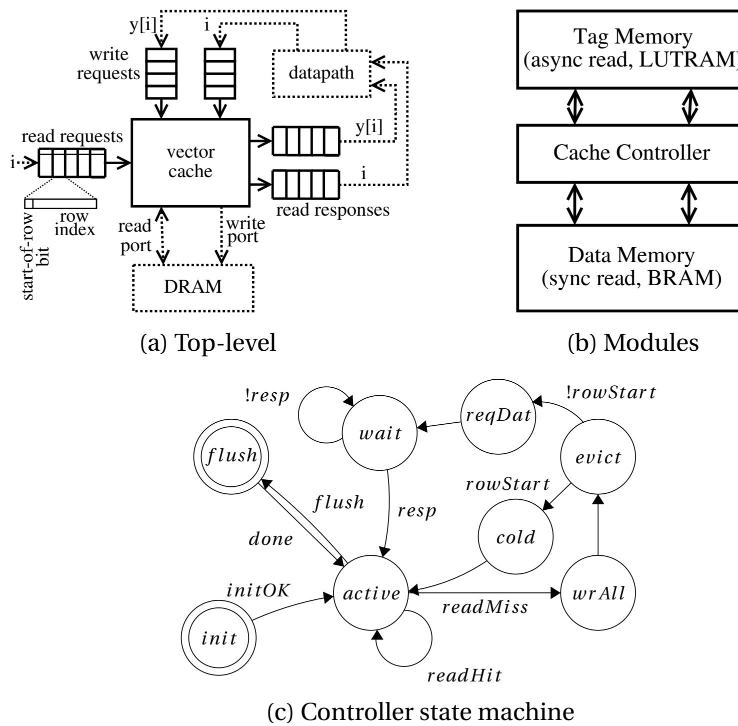
Having established how the matrix structure relates to vector cache misses, we will now formulate the preprocessing step. We assume that the preprocessing step will be carried out by the general-purpose core prior to copying the SpMV data into the accelerator's memory space.

One task that the preprocessing needs to fulfill is to establish the required cache capacity for the sparse matrix via the methods described in Section B2.3.1. Another important function of the preprocessing is marking the start of each row to avoid cold misses. In this paper, we reserve the highest bit of the `rowind` field in the CSC representation to mark a nonzero element as the start of a row. Although this decreases the maximum possible matrix that can be represented, it avoids introducing even more data into the already memory-intensive kernel, and can still represent matrices with over 2 billion rows for a 32-bit `rowind`. At the time of writing, this is 18x larger than the largest matrix in the University of Florida collection [22].

For the case of computing `maxAlive`, we can formulate the problem as constructing an interval tree and finding the largest number of overlapping intervals. Algorithm B2.1 presents an implementation which uses a priority queue to sort the start and end of each row according to their column indices. The values inserted are +1 and -1, respectively for row starts and ends. `maxAlive` is obtained by finding the maximum sum the sorted values during the iteration. We do not present the algorithm for finding `maxColSpan`, as it is simply iterating over each column of the sparse matrix and finding the one with the greatest span.

B2.3.4. Vector Cache Design

The final component of our vector caching scheme is the vector cache hardware itself. Our design is a simple increment over a traditional direct-mapped hardware cache to allow utilizing the start-of-row bits to avoid cold misses. A top-level overview of the vector cache and how it connects to the rest of the system is provided in Figure B2.4a. All interfaces use ready/valid handshaking and connect to the rest of the system via FIFOs, which simplifies placing the



(a) Top-level

(b) Modules

(c) Controller state machine

Figure B2.4.: Design of the vector cache.

cache into a separate clock domain if desired. Row indices with marked start-of-row bits are pushed into the cache as 32-bit-wide read requests. The cache returns the 64-bit read data, as well as the requested index itself, through the read response FIFOs. The datapath drains the read response FIFOs, sums the `y[i]` value with the latest partial product, and writes the updated `y[i]` value into the write request FIFOs of the cache.

Internally, the cache is composed of data/tag memories and a controller, depicted in Figure B2.4b. Direct-mapped associativity is chosen for a more suitable FPGA implementation as it avoids content-associative memories required for multi-way caches. To increase performance and minimize the RAW hazard window, the design offers single-cycle read/write hit latency, but read misses are blocking to respect the FIFO ordering of requests. To make efficient use of the synchronous on-chip SRAM resources in the FPGA while still allowing

single-cycle hits, we chose to implement the data memory in BRAM while the tag memory is implemented as look-up tables. The controller finite state machine is illustrated in Figure B2.4c. Write misses are directly transferred to the DRAM to keep the cache controller simple. Prior to servicing a read miss, the controller waits until there are no more writes from the datapath to guarantee memory consistency. Regular read misses cause the cache to issue a DRAM read request, which prevents the missing read request from proceeding until a response is received. Avoiding cold misses is achieved by issuing a zero response on a read miss with the start-of-row bit set, without issuing any DRAM read requests.

B2.4. Experimental Evaluation

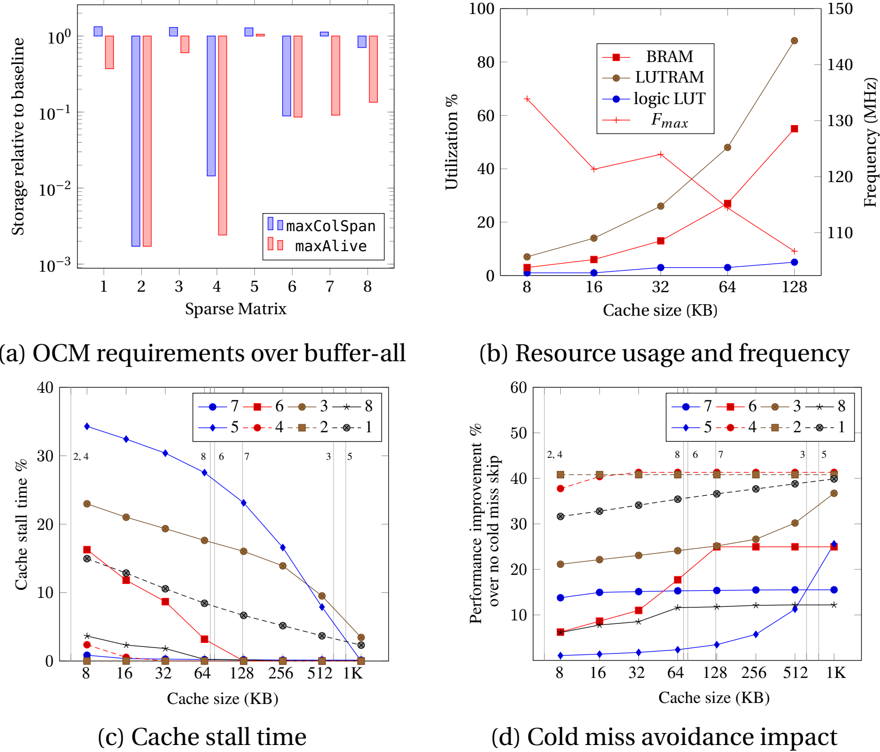
We present a two-part evaluation of our scheme: an analysis of OCM savings using the minimum required capacity estimation techniques, followed by performance and FPGA synthesis results of our our vector caching scheme. For both parts of the evaluation we use a subset of the sparse matrix suite initially used by Williams et al. [70], excluding the smaller matrices amenable to the buffer-all strategy. The properties of each matrix is listed in Table B2.1.

B2.4.1. OCM Savings Analysis

In Section B2.3.2 we described how the minimum cache size to avoid all capacity misses could be calculated for a given sparse matrix, either using `maxColSpan` or `maxAlive`. The rightmost columns of Table B2.1 list these values for each matrix. However, a vector cache also requires tag and valid bit storage in addition to the cache daha storage, which decreases the net OCM savings from our method. We compare the total OCM requirements of `maxColSpan`- and `maxAlive`-sized vector caches against the buffer-all strategy. The baseline is calculated as $64 \cdot m$ bits (one double-precision floating point value per `y` element), whereas the vector cache storage requires $(64 + \lceil log_2(W) \rceil + 1) \cdot$ `W` bits to also account for the tag/valid bits storage overhead, where W is the cache size. Figure B2.5a quantifies the amount of on-chip memory required for the two methods, compared to the baseline. For seven of the eight tested

Table B2.1.: Suite with `maxColSpan` and `maxAlive` values for each sparse matrix.

#	Name	Dimension	Nonzeroes	NZ/col	Problem Type	`maxColSpan`	`maxAlive`
1	webbase-1M	1000005	3105536	3.10	web connectivity matrix	997552	283024
2	mc2depi	525825	2100225	3.99	model of epidemic	770	770
3	scircuit	170998	958936	5.61	circuit simulation	170975	80408
4	mac_econ_fwd500	206500	1273389	6.17	macroeconomic model	2481	431
5	cop20k_A	121192	2624331	21.65	accelerator cavity design	121052	99843
6	shipsec1	140874	3568176	25.33	ship section detail	10145	9797
7	pwtk	217918	11524432	52.88	wind tunnel stiffness matrix	189337	16070
8	consph	83334	6010480	72.13	FEM concentric spheres	46481	9074



(a) OCM requirements over buffer-all

(b) Resource usage and frequency

(c) Cache stall time

(d) Cold miss avoidance impact

Figure B2.5.: Results from vector caching scheme evaluation.

matrices, significant storage savings can be achieved by using our scheme. A vector cache of size `maxAlive` requires 0.3x of the baseline storage on average, whereas sizing according to `maxColSpan` averaged at 0.7x of the baseline. It should be noted that matrices 2, 4 and 6, which have a more regular structure with elements clustered around the diagonal, already gain significant storage benefits from the low-overhead `maxColSpan` estimation. On the other hand, the irregular matrices 1 and 3 have no storage reduction benefits by using a `maxColSpan`-sized cache, so `maxAlive` must be used. For matrix #5, even `maxAlive` is only 17.6% smaller than the entire `y`, and therefore the savings from vector caching is not large enough to offset the tag overhead.

B2.4.2. Vector Cache Evaluation

We use Chisel [6] to create a parametrizable hardware description for the vector cache, which is converted to Verilog via the Verilog backend. The generated Verilog code is fed into XST for FPGA synthesis in order to obtain frequency and area results for the cache, and passed through the Verilator tool to generate a cycle-accurate SystemC model. The model is used in a in-house SpMV frontend simulator, which is stimulated with inputs corresponding to the chosen sparse matrix and models the behavior of the accumulator datapath and DRAM for performance assessment. We assume a 100 MHz clock for the frontend, with delays of 7 cycles for the accumulator datapath and 10 cycles for DRAM reads.

Area and Frequency: We report area and frequency results from synthesis for a Xilinx Spartan-6 LX45 FPGA with -2 speed grade, chosen to demonstrate the potential of the technique with mediocre OCM. Our results indicate that the cold skip enhancement is with very little extra hardware cost (less than 1% in logic LUTs for the largest tested design), hence we do not report separate results for a baseline cache without this enhancement. Figure B2.5b shows the percent utilization of BRAM, LUTRAM and logic resources for a range of vector cache sizes, and the maximum frequency F_{max} reported by the synthesis tool. A vector cache of 128 KB can fit on this relatively small FPGA, which is large enough to accommodate the `maxAlive`-sized working set of 5 of the 8 tested matrices. As can be expected, the utilization of BRAM and LUTRAM increases linearly with cache size, and the LUTRAM used for cache tags ultimately limits scaling to larger caches. Due to the simple design, the LUT utilization for implementing logic is rather small and occupies about 5% of the available resources for the largest design, which leaves plenty of room for implementing the more logic-intensive parts of the accelerator. The maximum attainable frequency is between 106 – 133 MHz for the tested designs, which is similar to the operating frequencies of previous SpMV accelerator designs. Further F_{max} improvements can be achieved by using a more powerful FPGA or design optimizations.

Cache Stall Time: As our goal is to enable a stall-free cache, we evaluate the impact of cache stalls with our scheme. Figure B2.5c depicts the percentage of total execution time the accelerator with up to 1 MB of cache is stalled due to cache misses. The `maxAlive` of each matrix is indicated with numbered

lines in the background. For 6 of the 8 matrices, allocating at least a `maxAlive`-sized cache with cold miss avoidance capability is enough to remove almost all cache misses, also indicating there are very few conflict misses. #3 is an exception, which suffers from conflict misses even with a large cache due to its nonzero pattern. The web connectivity matrix 1 has a working set larger than the maximum tested cache size, although its miss rate is already quite low. Low miss rates with cache sizes smaller than `maxAlive` is also observed for matrices 8 and 7, indicating that more relaxed working set definitions could be used for further reduction in required storage. Overall, by allocating at least `maxAlive`-sized caches, the cache stall time for our scheme is only 1.1% averaged across the test suite.

Cold Miss Avoidance: To show the gains from the cold miss avoidance technique, we plot the overall performance improvement due to removal of cold miss stalls in Figure B2.5d. The baseline for each data point is a vector cache of equal size without cold miss avoidance capabilities. The average performance improvement for at least `maxAlive`-sized caches is 28.6%. As the cache grows larger, fewer capacity misses are encountered and cold misses make up a larger percentage of the total. This increases the benefit from cold miss avoidance, until there are no cache misses left and the benefit levels off. Since larger sparse matrices exhibit more cold misses due to large `y` size, the greatest benefit is observed for the large matrices 1, 2 and 4, with up to 40% improvement. For matrix 5, the number of capacity misses with small caches is very large and very little benefit is observed until a cache size of 16K elements.

B2.5. Conclusion and Future Work

We have studied how matrix structure relates to cache misses, and proposed a scheme that uses preprocessing to enhance the operation of a traditional hardware cache for FPGA SpMV accelerators. Specifically, we have proposed two methods to estimate required cache depth to avoid all capacity misses, and a way of enhancing the matrix representation to avoid all cold misses. Our experiments with a suite of large sparse matrices indicate that the scheme can service random accesses to the result vector with no or few stalls, while avoiding cold miss penalties that hamper traditional hardware caches. Future work will include evaluating the vector caching scheme in a complete FPGA

SpMV accelerator context and developing techniques for eliminating the other sources of stalls, including RAW hazards.

Paper B3

Random Access Schemes for Efficient FPGA SpMV Acceleration

Yaman Umuroğlu and Magnus Jahre

Published in
Microprocessors and Microsystems 47 (2016): 321-332

B3. Random Access Schemes for Efficient FPGA SpMV Acceleration

Abstract. Utilizing hardware resources efficiently is vital to building the future generation of high-performance computing systems. The sparse matrix – dense vector multiplication (SpMV) kernel, which is notorious for its poor efficiency on conventional processors, is a key component in many scientific computing applications and increasing SpMV efficiency can contribute significantly to improving overall system efficiency. The major challenge in implementing SpMV efficiently is handling the input-dependent memory access patterns, and reconfigurable logic is a strong candidate for tackling this problem via memory system customization. In this work, we consider three schemes (all off-chip, all on-chip, caching) for servicing the irregular-access component of SpMV and investigate their effects on accelerator efficiency. To combine the strengths of on-chip and off-chip random accesses, we propose a hardware-software caching scheme named *NCVCS* that combines software preprocessing with a nonblocking cache to enable highly efficient SpMV accelerators with modest on-chip memory requirements. Our results from the comparison of the three schemes implemented as part of an FPGA SpMV accelerator show that our scheme effectively combines the high efficiency from on-chip accesses with the capability of working with large matrices from off-chip accesses.

B3.1. Introduction

High energy efficiency is a key challenge for building the next generation of computing systems where exascale performance levels are desired [50]. One strategy for achieving this goal is to build highly efficient primitives (accelerators) to implement computational kernels commonly encountered across applications. Sparse Matrix – Vector Multiplication (SpMV) is one such computational kernel, which is widely encountered in the scientific computation domain and frequently constitutes a bottleneck for such applications [70]. Besides the classical use in numerics, SpMV can be used to implement a variety of graph algorithms by customizing the add and multiply operators [15, 38, 64].

A defining characteristic of the SpMV kernel is the *irregular memory access pattern* caused by the sparse storage formats. A critical part of the kernel depends on accesses to memory addresses that correspond to non-zero element locations of the matrix, which are only known at runtime. These accesses constitute a large problem especially when SpMV is used for analysis of social graphs [15], since the resulting matrices are unstructured and are growing in size as part of the Big Data trend. The kernel is otherwise characterized by little data reuse and large per-iteration data requirements [70], which makes the performance memory bandwidth-bound. Storing the kernel inputs and outputs in high-capacity high-bandwidth DRAM is considered a cost-effective solution [30]; however, the burst-optimized architecture of DRAM contrasts with the fine-grained SpMV random accesses. Engineers and computer architects thus face an ever-growing "irregularity wall" in the quest for enabling efficient SpMV implementations.

Recently, there has been increased interest in FPGA-based acceleration of computational kernels. The primary benefit from FPGA accelerators is the ability to create customized memory systems and datapaths that align well with the requirements of each kernel, enabling stall-free and highly efficient execution. From the perspective of the SpMV kernel, the ability to deliver high external memory bandwidth, embedded SRAM blocks (which we refer to as *on-chip memory (OCM)*) to provide high-bandwidth random accesses and runtime specialization via dynamic reconfiguration are attractive properties.

Several FPGA implementations for the SpMV kernel have been proposed, either directly for SpMV or as part of larger algorithms like iterative solvers [25, 27,

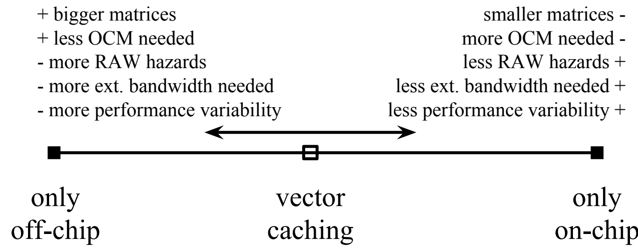


Figure B3.1.: Tradeoffs in SpMV result vector random accesses.

28], some of which present order-of-magnitude better energy efficiency and comparable performance to CPU and GPGPU solutions. These accelerators tackle the irregular access problem by buffering the entire random-access data in OCM. Unfortunately, this strategy is limited to operating on smaller matrices where the random-access data can fit in OCM. An alternative solution is performing random accesses to DRAM and mitigating the high latencies with many in-flight requests and memory-level parallelism. This approach is not limited by FPGA OCM size, but exhibits variable efficiency depending on the matrix structure since DRAM is not optimized for fine-grained random accesses. Finally, our previous work in [61] proposed a specialized vector caching scheme to take advantage of SpMV-specific reuse and access patterns, which showed promising results despite the limited evaluation in simulation and low efficiency for caches under a certain size due to lack of memory-level parallelism.

In essence, these three approaches line up along a tradeoff axis as illustrated in Figure B3.1. In this paper, we adopt a processor-like view of the SpMV reducer hardware for each random-access approach to better understand the tradeoffs and facilitate comparison. To combine the strengths of the on-chip-only and off-chip-only accesses, we extend upon our previous work in [61] to build the *Nonblocking Cooperative Vector Caching Scheme* (*NCVCS*). Afterwards, we compare these approaches by implementing them as part of a fully-functional FPGA SpMV accelerator. Our results indicate that *NCVCS* effectively balances the use of off- and on-chip memory bandwidth, achieving 73% average computational efficiency across a set of large sparse matrices while occupying half of the OCM on a Zynq Z7020. This work makes the following new contributions:

- A processor-like view of SpMV reducer hardware to analyze and compare vector random access schemes.
- A simplified preprocessing algorithm for providing cold miss skip capabilities with low overhead.
- A pipelined, nonblocking vector cache architecture for *NCVCS* that supports cold miss skip as well as exploiting memory-level parallelism.
- Three open-source SpMV accelerator implementations that make use of different random access methods, and their evaluation with respect to performance, efficiency and resource usage on the ZedBoard.

B3.2. Background

B3.2.1. The SpMV Kernel and Sparse Matrix Storage

The SpMV kernel $\vec{y} = \mathbf{A} \cdot \vec{x}$ consists of multiplying an $m \times n$ sparse matrix $\mathbf{A}$ with NZ nonzero elements by a dense vector $\vec{x}$ of size n to obtain a result vector $\vec{y}$ of size m. The sparse matrix is commonly stored in a format which allows storing only the nonzero elements of the matrix. Many storage formats for sparse matrices have been proposed, some of which specialize on particular sparsity patterns, and others suitable for generic sparse matrices. Among the most popular storage formats for generic sparse matrices are Compressed Sparse Row (CSR) and its column-major counterpart Compressed Sparse Column (CSC). Figure B3.2 illustrates a sparse matrix, its representation in the CSC format, and the pseudocode for performing column-major SpMV. We use the `variable` notation to refer to CSC SpMV data in memory (e.g., `values`, `rowind`, `colptr`, `x`, `y`). As highlighted in the figure, the result vector `y` is accessed depending on the `rowind` values, causing the random access patterns that are central to this work.

In this paper, we will focus on column-major sparse matrix traversal (in line with [25, 30, 62]) and the CSC storage format with 4-byte indices and double-precision floating point values. Column-major traversal enables simpler SpMV datapaths by interleaving different rows the adder pipeline (see Section B3.3)

$$\begin{bmatrix} 1.1 & 0 & 0 \\ 0 & 2.2 & 3.3 \\ 4.4 & 0 & 5.5 \end{bmatrix}$$

```
colptr={0 2 3 5}
values={1.1 4.4 2.2 3.3 5.5}
rowind={0 2 1 1 2}
```

```
for(j=0 to n-1)
  for(i=colptr[j] to colptr[j+1])
    y[rowind[i]] += values[j] * x[j]
```

Figure B3.2.: A matrix, its CSC representation and SpMV pseudocode. The clause causing random-access to y is highlighted.

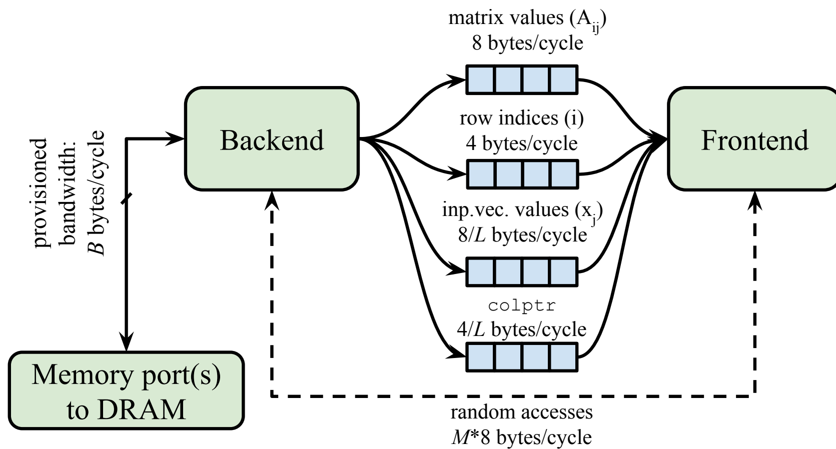


Figure B3.3.: Decoupled SpMV accelerator architecture.

and permits maximum temporal reuse of the input vector and `colptr` values; each of these values gets reused $L = \frac{NZ}{n}$ times.

B3.2.2. Bandwidth-Bound Performance and Efficiency

SpMV has a low computation-to-memory ratio and typically requires double-precision floating point arithmetic when used as part of scientific computation. As such, the inputs can be very large and are stored in external DRAM, which makes the kernel performance bounded by DRAM bandwidth. Our previous

work in [62] proposed the decoupled architecture illustrated in Figure B3.3 to match the computational capability with memory bandwidth and identify sources of inefficiency. Here, the *backend* is provisioned with B bytes per cycle of external memory bandwidth, and is responsible for fetching matrix and input vector data from DRAM. The backends feeds a *frontend* with data, which performs the actual computation. In order to keep a frontend supplied with enough data to perform one multiply and one add per cycle, the accelerator should be provisioned with $B \geq 12 + \frac{12}{L}$ bytes per cycle. In this work, we consider an extended SpMV frontend where some or all result vector accesses may be serviced from DRAM to allow working with bigger matrices, with random accesses indicated with the dashed line in Figure B3.3. If M is the ratio of random vector accesses that go to DRAM, achieving full throughput with $M > 0$ requires more bandwidth, as described in Equation B3.1.

$$B \geq 8 \cdot M + 12 + \frac{12}{L} \tag{B3.1}$$

However, DRAM is not optimized for fine-grained random accesses and may not deliver the desired bandwidth, which causes the frontend to stall and decreases efficiency. The goal of this work is to devise a random access scheme that minimizes these stalls by minimizing M and exploiting memory-level parallelism, thus maximizing efficiency and performance.

B3.2.3. Sparse Matrix Preprocessing

Since the random-access behavior of the SpMV kernel is dependent on the particular sparse matrix used, a preprocessing step is often introduced to optimize performance for a particular matrix. Many different forms of preprocessing for SpMV have been proposed to date. A common form of preprocessing is using the Cuthill–McKee (CMK) algorithm [20] or its reverse (RCMK) to reorder the matrix for smaller bandwidth. Oliker et al. [47] compares the effects of different forms of matrix partitioning and reordering on parallel supercomputers. Kourtis et al. [40] and Wilcock and Lumsdaine [69] propose using preprocessing to compress the matrix and reduce the required memory bandwidth. Pichel et al. [48] investigate the performance effects of reordering techniques on GPUs, and report speedups of up to 2.6x.

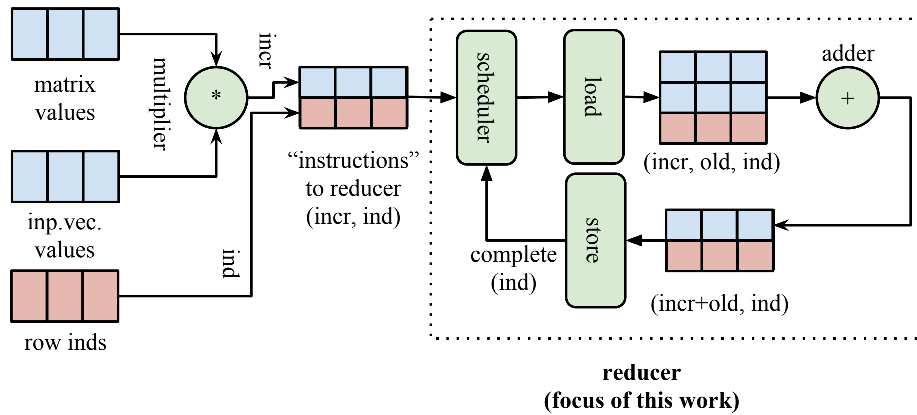


Figure B3.4.: Overview of a column-major SpMV accelerator frontend.

Preprocessing has a certain time cost, as the original matrix must be first read into memory, then the preprocessing performed and the results written back. Fortunately, algorithms that make heavy use of SpMV tend to do so iteratively, i.e., they multiply the same sparse matrix with many different vectors, which enables ameliorating the cost of preprocessing across speed-ups in each SpMV iteration. Toledo et al. [59] reports the cost of preprocessing to be 1–3 times the SpMV cost for simpler reordering techniques (CMK, RCMK), 4–15 times for blocking, and 20–200 times the SpMV cost for a nested dissection-type reordering. They estimate that the speedup from blocking pays its preprocessing cost in approximately 75 iterations. We also adopt a preprocessing step in our scheme to enable optimizing for a given sparse matrix, but unlike previous work, our preprocessing stage produces extra information to enable specialized cache operation instead of changing the matrix structure.

B3.3. Frontend Architectures and Random Access

The frontend of a column-major SpMV accelerator can be viewed as a multiply-accumulator with feedback from a random-access memory, as illustrated in Figure B3.4. The first part of the frontend is the multiplier, which produces the value we refer to as `incr` by multiplying each matrix nonzero with its corresponding input vector element. This does not pose a significant design

problem as the multiplier only needs to support backpressure and can have an arbitrary number of pipeline stages. The pair `(incr, ind)` is passed to the *reducer* to perform `y[ind] += incr`. If we think of the reducer as a very simple processor, `(incr, ind)` can be thought of as an "accumulate" instruction that increments the register `ind` by `incr`. The reducer loads the current value of `old=y[ind]`, compute `incr+old`, and write `y[ind]=incr+old` to complete each instruction. Note that instructions with the same `ind` target the same register, which constitutes a read-after-write (RAW) hazard that prevents these instructions from running in parallel. A *scheduler* prevents this by controlling instruction dispatch, monitoring which `ind` values are in-flight (i.e., not yet completed) and stalling[1] incoming instructions with hazards. The scheduler *issue window* determines the maximum number of in-flight values.

In the following subsections, we will use this processor-like view of the reducer to view the three random-access policies proposed in prior work on FPGA SpMV acceleration in a common frame. Our intent here is not to exhaustively categorize how accesses could be handled, but rather illustrate how response latency and its variability affect the frontend architecture. For instance, random accesses could also be serviced from a large, external SRAM chip, which would have a deterministic random-access latency that is larger compared to OCM but less compared to DRAM.

B3.3.1. All on-chip: ***BufferAll***

Figure B3.5 illustrates a *BufferAll* reducer, which uses FPGA OCM to store and access the entire result vector during SpMV. This offers deterministic and low latency for random reads and writes, which results in highly efficient accelerator implementations. Each instruction is completed after a fixed number of cycles, i.e read/write complete signals can be generated by matching the BRAM read/write delays with a shift register chain and all responses return in-order, which makes the scheduler simple. An issue window of size equal to adder stages plus read and write latency is enough to achieve full throughput, which results in fewer RAW hazard stalls. Keeping all `y` accesses on-chip (i.e., $M = 0$ in Equation B3.1) frees up external memory bandwidth for reading more matrix

[1] Dynamically re-ordering the dispatches can give fewer stalls, though we do not investigate their benefit in this work.

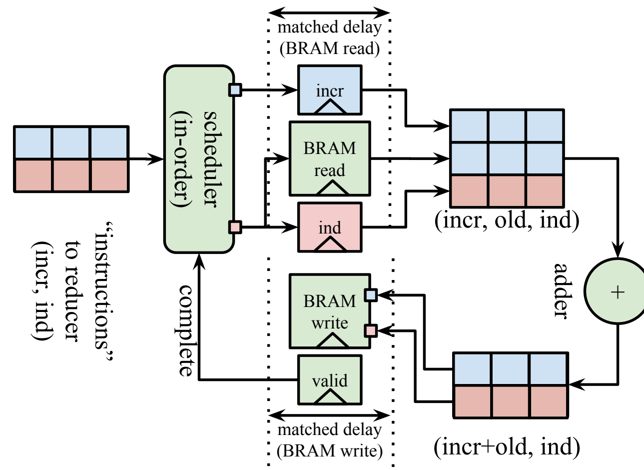


Figure B3.5.: Architecture of a *BufferAll* reducer.

data and enables higher performance. Unfortunately, this is only possible if the entire result vector y can fit within FPGA OCM, which limits the maximum number of rows in the matrix that the accelerator can process.

Much of the prior work on FPGA SpMV acceleration [25, 27, 28, 35, 75] uses OCM for buffering the entire random accessed component and the maximum matrix dimension (rows in column-major, columns in row-major) becomes limited by OCM capacity. The largest sparse matrix dimension in the evaluation is 8127 in the work by Jain-Mendon and Sass [35], 16K in the work by Fowers et al. [27], 17281 in the work by Zhang et al. [75] and 63838 in the work by Chow et al. [28], whereas we consider sparse matrices of up to a million rows in our evaluation. The work by Dorrance et al. [25] illustrates both the benefits and drawbacks of this strategy. While they report a high average E of 92%, the matrix height is limited by OCM (reported maximum 218K rows) and they resort to reduced precision data types to allow larger sparse matrices.

B3.3.2. All off-chip: *BufferNone*

A *BufferNone* reducer, which services the random accesses directly from DRAM, is depicted in Figure B3.6. Although DRAM latency is much higher compared

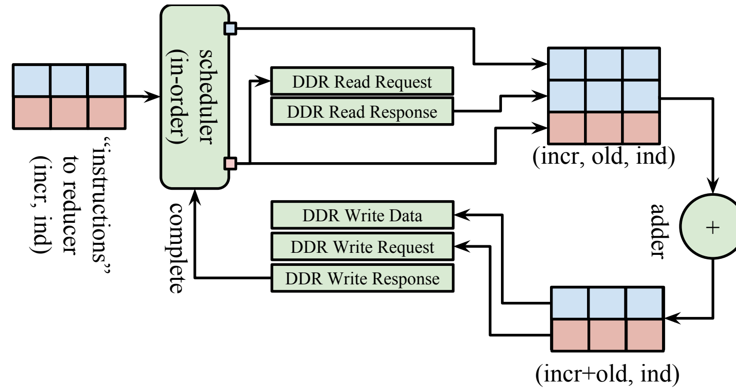


Figure B3.6.: Architecture of a *BufferNone* reducer.

to OCM, DRAM-based memory systems permit multiple outstanding requests to mitigate latency. By using a large issue window, the accelerator can take advantage of memory-level parallelism and achieve high throughput, but a large issue window may result in more RAW hazards. Due to the variability of latencies, read/write-completion must be signaled directly by the memory system instead of delay-matched registers. Responses may also be returned out-of-order to maximize DRAM efficiency, though some systems can handle this internally and return in-order to allow for simpler user logic, which we assume in this work. Overall, the *BufferNone* approach does not require large amounts of OCM and the matrix size is only limited by the DRAM capacity, which is a natural limitation for high-performance systems. The primary disadvantage is the additional consumption of valuable external memory bandwidth ($M = 1$) and performance variability due to DRAM bank conflicts and increased RAW hazards. It should also be noted that DRAM is optimized for large bursts; fine-grained random accesses may not achieve peak bandwidth.

To our knowledge, this strategy has been only used for row-major and mixed-traversal SpMV accelerators. These may handle RAW hazards differently but still use memory-level parallelism to mitigate latency. The work by Halstead and Najjar [31] uses many in-flight requests for high-throughput DRAM random accesses to the input vector. They report efficiency ranging from 10% to 72% depending on the particular sparse matrix used, with an average of 48%. Townsend and Zambreno [60] also use multiple memory requests to access the

input vector in DRAM, although they split the matrix into 16-row chunks to mix column-major and row-major traversal and use data compression to enhance performance. They report performance ranging from 2.1 to 13.6 GFLOPS (average 7.5 GFLOPS), which corresponds to an average efficiency of 40% for the peak performance of 19 GFLOPS.

B3.3.3. Balanced: *BufferCache*

A *BufferCache* reducer uses a chunk of OCM to buffer a portion of the result vector according to some caching policy, and uses DRAM to service the elements that are not in the cache. It has the potential to combine the strengths of *BufferAll* and *BufferNone*, though a "one size fits all" caching policy is difficult to construct due to matrix-dependent memory accesses.

Gregg et al. [30] use a simple, blocking direct-mapped cache to supply y values, backed by DRAM to enable scaling to large matrices. To avoid the major efficiency penalties due to cache misses blocking the entire accelerator, they proposed to split the matrix into cache-sized chunks. They report efficiency ranging from 14% to 72% and mention that splitting can introduce overheads due to increased matrix storage costs and more RAW hazard stalls. Nagar and Bakos [46] include a 8192-element direct mapped cache in their accelerator, with performance of 1.17 – 3.95 GFLOPS (corresponding to 12%–41% of the 9.6 GLOPS peak), but they do not discuss how cache misses influence performance. Finally, our previous work in [61] examined how sparse matrix structure relates to cache misses and proposed a vector caching scheme to combine software preprocessing with special hardware to avoid cold misses. We present an improved version of this vector caching scheme in Section B3.4.

B3.4. The Nonblocking Cooperative Vector Caching Scheme

It is desirable to balance and combine the strengths of the *BufferAll* and *BufferNone* approaches into a *BufferCache*-type scheme and enable high efficiency on large sparse matrices without being constrained by FPGA OCM.

To achieve this, we propose[2] the Nonblocking Cooperative Vector Caching Scheme (*NCVCS*), which uses a combination of software preprocessing and specialized hardware to achieve a high cache hit rate with an OCM footprint smaller than *BufferAll*. Specifically, the *preprocessing step* analyzes the matrix structure to provide a minimal estimate of the required cache capacity and mark the locations where cold cache misses will occur. The capacity estimation can be used to resize (via runtime reconfiguration) the *vector cache hardware*, which can make use of the cold miss markings and memory-level parallelism to offer high random-access performance.

In the following sections, we first describe how matrix structure relates to data reuse and cache misses, then formulate a preprocessing algorithm and describe a nonblocking cache architecture suitable for FPGAs for realizing *NCVCS* in hardware.

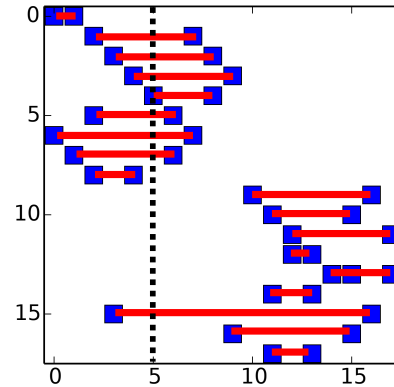
B3.4.1. Row Lifetime Analysis

To relate the vector cache usage to the matrix structure, we start by defining a number of structural properties for sparse matrices. First, we note that each row has a strong correspondence to a single result vector element, i.e., `y[i]` contains the dot product of row `i` with `x`. The period in which `y[i]` is used is solely determined by the period in which row `i` accesses it. This is the key observation that we use to specialize *NCVCS* for a given sparse matrix.

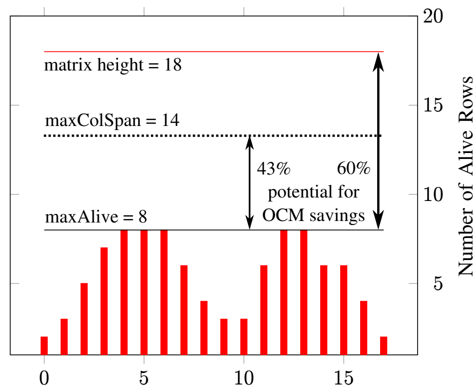
Calculating `maxAlive` and `maxColSpan`

For a matrix with column-major traversal, we define the *aliveness interval* of a row as the column range between (and including) the columns of its first and last nonzero elements, and will refer to the interval length as the *span*. Figure B3.7a illustrates the aliveness intervals as red lines extending between the first and last non-zeroes of each row. For a given column `j`, we define a set of rows to be *simultaneously alive* in this column if all of their aliveness intervals contain `j`. The number of alive rows for a given column is the maximum size of

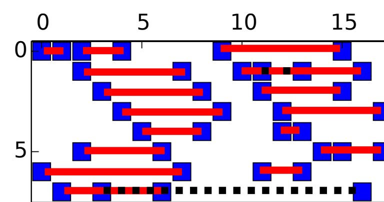
[2] *NCVCS* is an extension of our previous work in [61], and avoids the shortcomings of high LUTRAM consumption and transpose operations in preprocessing, as well as improving performance for small cache sizes by exploiting memory-level parallelism.



(a) Matrix and row lifetimes.



(b) Alive rows in each column.



(c) Conflicts for an 8-line direct-mapped cache.

Figure B3.7.: Matrix Pajek/GD01_b (from [22]) and its vector caching analysis.

such a set. Visually, this can be thought of as the number of aliveness interval lines that intersect the vertical line of a column. For instance, the dotted line corresponding to column 5 in Figure B3.7a intersects 8 intervals, and there are 8 rows alive in column 5. Finally, we define the *maximum simultaneously alive rows* of a sparse matrix, further referred to as `maxAlive`, as the largest number of rows simultaneously alive in any column of the matrix. Incidentally, `maxAlive` is equal to 8 for the matrix given in Figure B3.7a – though the alive rows themselves may be different, no column has more than 8 alive rows in this example.

Calculating `maxAlive` requires preprocessing the matrix. If the accelerator design is not under very tight OCM constraints, it may be desirable to estimate `maxAlive` instead of computing the exact value in order to reduce the preprocessing time. If we define aliveness interval and span for columns as was done for rows, the largest column span of the matrix `maxColSpan` provides an upper bound on `maxAlive`. The column 3 in Figure B3.7a has a span of 14, which is `maxColSpan` for this matrix.

B3.4.2. Avoiding Vector Cache Misses

We now use the canonical cold/capacity/conflict classification to break down cache misses into three categories and explain how accesses to the result vector relate to each category. For each category, we will describe how misses can be related to the matrix structure and avoided where possible.

Cold Misses

Cold (compulsory) misses occur when a vector element is referenced for the first time, at the start of the aliveness interval of each row. For matrices with very few elements per row, cold misses can contribute significantly to the total cache misses. Although this type of cache miss is considered unavoidable in general-purpose caching, a special case exists for SpMV. Consider the column-major SpMV operation $y = Ax$ where the y vector is random-accessed using the vector cache. The initial value of each y element is zero, and is updated by adding partial sums for each nonzero in the corresponding matrix row. If we can distinguish cold misses from the other miss types at runtime, we can avoid

them completely: a cold miss to a y element will return the initial value, which is zero[3]. Recognizing misses as cold misses is critical for this technique to work. We propose to accomplish this by introducing a *start-of-row bit* marked during preprocessing, as described in Section B3.4.3.

Capacity Misses

Capacity misses occur due to the cache capacity being insufficient to hold the SpMV result vector working set. Therefore, the only way of avoiding capacity misses is ensuring that the vector cache is large enough to hold the working set. Caching the entire vector (the *BufferAll* strategy) is straightforward, but is not an accurate working set size estimation due to the sparsity of the matrix. While methods exist to attempt to reduce the working set of the SpMV operation by permuting the matrix rows and columns, they are outside the scope of this paper. Instead, we will concentrate on how the working set size can be estimated. This estimation can be used to reconfigure the FPGA SpMV accelerator to use less OCM, which can be reallocated for other components. In this work, we make the assumption that a memory location is in the working set if it will be reused at least once to reap all the caching benefits. Thus, the cache must have a capacity of at least `maxAlive` to avoid all capacity misses. This requires the computation of `maxAlive` during the preprocessing phase. If OCM constraints are more relaxed, the `maxColSpan` estimation described in Section B3.4.1 can be used instead. Figure B3.7b shows the row lifetime analysis for the matrix in Figure B3.7a and how different estimations of the required capacity yield different OCM savings compared to *BufferAll*.

Conflict Misses

Conflict misses arise when two simultaneously alive vector elements map to the same cache line. This is determined by the nonzero pattern, number of cachelines and the chosen hash function. Assuming that the vector cache has enough capacity to hold the working set, avoiding conflict misses is an associativity problem. Since content-associative memories are expensive in

[3]The more general SpMV form $y = Ax + b$ can be easily implemented by adding the dense vector b after $y = Ax$ is computed.

```
function MarkRowStarts(CSCMatrix A, bool reverse)
    seen[0..n−1] ← 0
    isRowStart[0..NZ−1] ← 0
    for e ← 0..NZ−1 do
        nzind ← (reverse ? NZ−1−e : e)
        rowind ← A.rowind[nzind]
        if seen[rowind] == 0 then
            seen[rowind] ← 1
            isRowStart[nzind] ← 1
        end if
    end for
    return isRowStart
end function
```

Algorithm B3.1: Marking row starts or ends.

```
function GetMaxAlive(CSCMatrix A)
    isRowStart[0..NZ−1] ← MarkRowStarts(A, false)
    isRowEnd[0..NZ−1] ← MarkRowStarts(A, true)
    maxAlive ← 0, currentAlive ← 0
    for e ← 0..NZ−1 do
        currentAlive ← currentAlive + isRowStart[e] − isRowEnd[e]
        maxAlive ← Max(maxAlive, currentAlive)
    end for
    return maxAlive
end function
```

Algorithm B3.2: Finding `maxAlive`.

FPGAs, direct-mapped caches are often preferred. The conflicts arising from this type of mapping can be visualized by "folding" the matrix height to be equal to the cachelines, as shown in Figure B3.7c. Here, the conflicts from the example matrix on a 8-line direct-mapped cache are visible as dotted lines on cachelines 1 and 7. Our results in Section B3.6.4 suggest that conflicts are few for most matrices even with a direct-mapped cache, as long as the cache capacity is sufficient. Jouppi [36] describes how a small fully-associative memory called a *victim cache* can significantly reduce conflict misses in a direct-mapped cache, which could be implemented for cases with many conflict misses.

B3.4.3. Preprocessing

Having established how the matrix structure relates to vector cache misses, we will now formulate the preprocessing step. We assume that the preprocessing

```
function GetMaxColSpan(CSCMatrix A)
    maxColSpan ← 0, currentColSpan ← 0
    for j ← 0..n − 1 do
        firstRowInCol ← A.rowinds[A.colptr[j]]
        lastRowInCol ← A.rowinds[A.colptr[j + 1] − 1]
        currentColSpan ← lastRowInCol − firstRowInCol
        maxColSpan ← Max(currentColSpan, maxColSpan)
    end for
    return maxAlive
end function
```

Algorithm B3.3: Finding `maxColSpan`.

step will be carried out by the general-purpose core prior to copying the SpMV data into the accelerator's memory space. Note that the original preprocessing algorithm in [61] was formulated on the row-major matrix representation (which requires a transpose operation) and required a priority queue. The algorithms we present here operate directly on the column-major (CSC) and with simple operations, which allow for low-overhead preprocessing.

One task that the preprocessing needs to fulfill is marking the start of each row to avoid cold misses. This is accomplished by Algorithm B3.1, which generates a boolean array with one element per matrix nonzero, indicating whether that nonzero is the first element of a row. We reserve the highest bit of each `rowind` value as a flag to indicate the start of a row. Although this decreases the maximum possible matrix that can be represented, it avoids introducing even more data into the already memory-intensive kernel, and can still represent matrices with over 2 billion rows for a 32-bit `rowind`. At the time of writing, this is 16x larger than the largest matrix in the University of Florida sparse matrix collection [22].

To minimize OCM usage of the cache by the methods described in Section B3.4.1, either `maxAlive` or `maxColSpan` needs to be computed. Algorithm B3.2 can be used to compute `maxAlive` by first determining matrix elements where rows start and end, then iterating over these arrays to find the maximum number of rows alive. Alternatively, Algorithm B3.3 can be used to compute `maxColSpan` by iterating over each column of the sparse matrix and finding the one with the greatest span, which has less preprocessing cost but tends to overestimate the needed capacity.

B3.4.4. *NCVCS* Hardware

We now describe a nonblocking cache architecture that forms the hardware component of *NCVCS*, with several improvements over the implementation of our previous work in [61]. For the simple, blocking cache architecture presented in [61], a cache miss will cause the reducer to stall until data is fetched from DRAM. In case the cache capacity is insufficient to contain the working set size, there will be many such stalls, which decreases efficiency significantly. In this work, we address this problem with a *nonblocking* cache that allows other requests to be processed while a cache miss is handled. Additionally, cache tags were kept in combinational-read LUTRAM in [61], which made scaling to larger cache sizes difficult due to the resource bottleneck and frequency penalties. In this design, both result vector data and tag/valid information is kept in synchronous-read OCM and support pipelining, which frees up valuable LUTRAM and exhibits better scalability. Finally, we include a small write buffer to avoid having to wait for completion of all pending DRAM writes, which further improves upon [61]. Otherwise, we keep the design choices from [61] for associativity (direct-mapped) and write policy (write-back), since these allow for an FPGA-suitable implementation that conserves DRAM write bandwidth.

Cache Reads

Figure B3.8 illustrates the read path for the nonblocking cache. Most of the design is built to operate in a continuous data-flow manner, although a state machine is still used to direct the flow of data in the read path when necessary. Each read request to the cache includes the row index (with the most significant bit used as the start-of-row flag) and an operand to the adder. The request-to-response read path is structured as follows: first, tag and data are read from OCM, and stored in the request–response queue together with the original request. Each item in the request–response queue is sent along one of the three response paths, which are the following in order of decreasing priority:

1. **Cache hit:** If the cached tag is equal to the request tag, the data response and request index are directly sent to the read response.

Figure B3.8.: Nonblocking cache read path.

2. **Cold miss:** If the start-of-row bit is set, this is a cold miss and a zero is returned as read data. The cache data and tag are updated, which may cause an eviction to DRAM.

3. **Regular miss:** Otherwise, this is a regular miss and a DRAM read request is issued. Upon read response from DRAM, the cache read response is emitted, and the cache data and tag are updated, which may cause an eviction to DRAM.

For cache misses, all information about the miss (from the request–response queue) is put onto the *miss queue*, which prevents misses from blocking the request–response queue. This is the critical component that enables nonblocking cache operation, and the size of the miss queue determines how many outstanding misses the cache can have before blocking. Permitting a larger number of outstanding misses lets the cache exploit more MLP, and can improve performance when cache misses are frequent. When the read request causing the miss is ready to be served (i.e., immediately for a cold miss, or when DRAM read data returns for a regular miss) it is removed from the miss queue and its read response emitted. Due to the out-of-order operation, the original

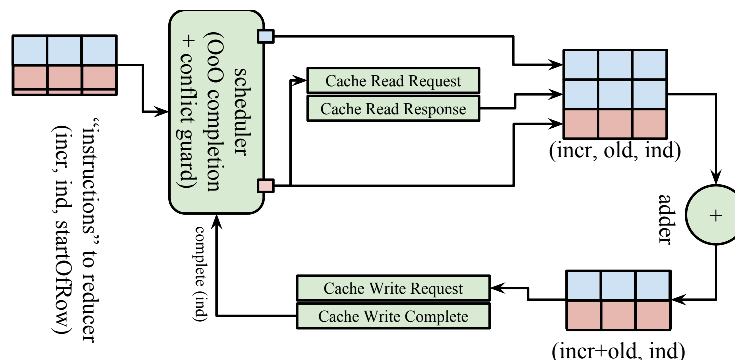


Figure B3.9.: Architecture of the *NCVCS*-reducer.

operand and row index are always emitted as part of the read response along with the returned read data.

Evictions and the Write Buffer

Since the cache uses a write-back policy to conserve DRAM bandwidth, there is a risk for RAW hazards on the DRAM level. Specifically, if a DRAM read request is made to a location with a pending DRAM write, an outdated (incorrect) read response will be returned. In [61] this was avoided by waiting for all pending writes to complete before issuing a DRAM read. This is detrimental for performance, especially for a nonblocking cache. Our design addresses this problem by including a *write buffer*, which is a small (8-entry) associative memory that keeps track of pending DRAM writes. The write buffer contents are checked prior to issuing DRAM reads to prevent RAW hazards. Note that the current implementation does not store the write data itself. Buffering the write data as well would act as a victim cache, which was originally proposed by Jouppi [36] to decrease conflict miss penalties in direct-mapped caches.

The *NCVCS*-reducer and Cache Writes

Figure B3.9 presents a *NCVCS*-reducer that uses the nonblocking vector cache for handling random accesses. As the nonblocking property causes out-of-order

Table B3.1.: Characteristics of the ZedBoard.

System-on-a-Chip	Zynq Z7020
CPU core	Dual ARM Cortex-A9, 666 MHz
CPU cache	32 KB L1D+L1I, 512 KB L2
DRAM and bandwidth	512 MB DDR3, 3.2 GB/s
FPGA logic resources	53200 slice LUTs, 220 DSP slices
FPGA OCM	560 KB (140 BRAMs)

cache responses, we use a scheduler with in-order dispatch and out-of-order retirement. Additionally, we use the scheduler as a *cache conflict guard* to simplify the cache design. Instead of comparing the entire row index of the head instruction, the scheduler compares only the part that corresponds to cache index bits. This prevents instructions with the same cache index (i.e., conflict misses) from entering the reducer; all in-flight instructions map to a different cache index. Since the cache tag is allocated during read response that precedes the write, all incoming cache writes are guaranteed to hit in the cache. This makes the write path of the cache trivial, as data can be written directly to OCM without checking tags. Out-of-order completion requires the row index of the completed instruction to be signaled to the scheduler, which is generated by the cache upon a write complete.

B3.5. Experimental Setup

To evaluate and compare how *BufferAll*, *BufferNone* and *NCVCS* perform as part of a real FPGA SpMV accelerator, we implemented an accelerator system that follows the architectural template in Figure B3.5.. The source code is available from `http://git.io/vsMNJ`. To characterize performance on different sparse matrices, we use the matrix suite from Williams et al. [70] which contains a variety of large sparse matrices from different domains. The properties of each matrix is listed in Table B3.2.

The accelerator system is deployed on the ZedBoard, whose characteristics are shown in Table B3.1. The Zynq chip offers dual ARM Cortex-A9 cores and FPGA

fabric, both of which can access the on-board DRAM. A single ARM core running at 666 MHz was used with bare-metal software for loading matrices from the SD card, executing the preprocessing algorithm and controlling accelerator execution. Most of the SpMV accelerator hardware was built in the hardware description language Chisel [6], except a few components (large BRAMs, large FIFOs and the double-precision floating point operators) which were generated with Xilinx Core Generator and imported as Verilog blackboxes. Vivado 2014.4 was used for synthesis, place and route. The FPGA OCM (BRAM) is used for the FIFOs between the backend and frontend, for storing y in *BufferAll* and for storing cache data and tags in *BufferCache*. The double-precision floating point operators are pipelined with 8 stages for the multiplier and 4 stages for the adder. All accelerator variants are set to operate at 100 MHz, which corresponds to a peak performance of 200 MFLOPS (one add and one multiply per clock cycle). We provision the accelerator with a bandwidth of $B = 16$ bytes per cycle (1.6 GB/s) from two AXI high-performance (HP) ports on the Zynq, which is rate-matched with the peak performance for the matrix suite if M is close to zero (see Section B3.2.2 and Equation B3.1).

We consider only a single SpMV processing element in our performance evaluation. The reason for this is the intricate link between parallel partitioning of sparse matrices and random accesses. Partitioning essentially creates smaller sparse matrices with new access patterns, whose performance can differ significantly. As there are many possible partitionings for each matrix, the experimental space becomes very large and makes it hard to concentrate on the differences between random vector access schemes themselves. Our results indicate that the matrices in the suite contain sufficient irregularity to study different SpMV memory behavior and interactions with the vector access schemes, even in unpartitioned form.

B3.6. Results

We now present and discuss our experimental results, which are organized into four parts. We start with FPGA synthesis results and a best-case performance comparison of the accelerator variants, then provide a more detailed analysis on *BufferNone* and *NCVCS* performance in the last two subsections.

Table B3.2.: Suite with `maxColSpan` and `maxAlive` values for each sparse matrix.

#	Name	Rows (Cols)	Nonzeroes	NZ/col (L)	Problem Type	`maxColSpan`	`maxAlive`
1	cant	62451	4007383	64.17	FEM cantilever	549	549
2	conf5_4-8x8-05	49152	1916928	39	quantum chromodynamics	48392	13836
3	consph	83334	6010480	72.13	FEM concentric spheres	46481	9074
4	cop20k_A	121192	2624331	21.65	accelerator cavity design	121052	99843
5	mac_econ_fwd500	206500	1273389	6.17	macroeconomic model	2481	431
6	mc2depi	525825	2100225	3.99	model of epidemic	770	770
7	pdb1HYS	36417	4344765	119.31	protein database	34475	8499
8	pwtk	217918	11634424	53.39	wind tunnel stiffness matrix	189337	16070
9	rma10	46835	2374001	50.69	3D CFD model	25380	19269
10	scircuit	170998	958936	5.61	circuit simulation	170975	80408
11	shipsec1	140874	7813404	55.46	ship section detail	10145	9797
12	webbase-1M	1000005	3105536	3.10	web connectivity matrix	997552	283024

Table B3.3.: Resources and frequency for best-case configurations.

Resource	*BufferAll*	*NCVCS*	*BufferNone*
Frontend LUTs	2346 (4%)	2978 (6%)	2411 (5%)
Total BRAMs	117 (84%)	77 (55%)	3 (2%)
Total LUTs	6470 (12%)	7443 (14%)	6665 (13%)
F_{max}	133 MHz	131 MHz	145 MHz

B3.6.1. FPGA Resources and Frequency

Table B3.3 compares the LUT and BRAM resource utilization for the best-performing configurations (Section B3.6.2). All frontend variants include the floating-point adder (1110 LUTs) and multiplier (359 LUTs, 10 DSP slices). The logic for the scheduler and random vector access handling uses ~1000 LUTs for *BufferAll* and *BufferNone* and ~1500 LUTs for *NCVCS*, accounting for ~15% of the entire accelerator. The most marked difference for the three variants is in BRAM utilization. Note that the BRAM utilization limits the largest power-of-two sized *BufferAll* y to 64K[4] elements and *NCVCS* to 32K elements (smaller than *BufferAll* due to cache tag overhead).

The maximum clock frequency for each configuration is also reported in Table B3.3. The slightly higher F_{max} for *BufferNone* is likely due to lower BRAM usage compared to *BufferAll* and *NCVCS*. As the goal of our study is to maximize efficiency for a given amount of external memory bandwidth, we did not perform detailed timing analysis or frequency optimizations in this work. If this is desired, the latency-insensitive construction of our accelerator system can accommodate deeper pipelining and retiming to increase the clock frequency.

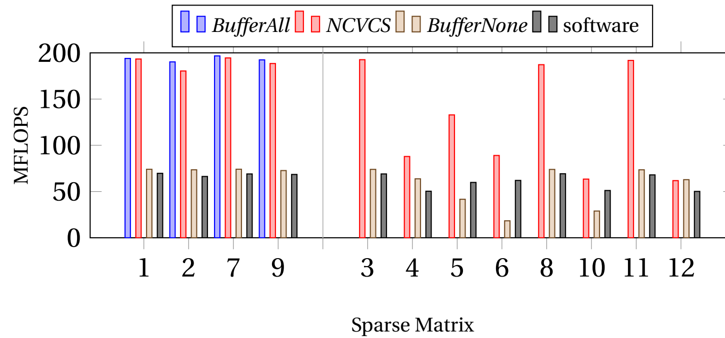


Figure B3.10.: Comparison of best-case SpMV performance.

B3.6.2. Best-Case Performance

We now present a comparison of maximum SpMV accelerator performance with different random access methods. For each access method, we empirically determined the following parameters to maximize performance:

- *NCVCS*: maximum sized cache (32K elements), cold miss skip enabled, nonblocking with 16-element issue window and miss queue.
- *BufferNone*: 32-element issue window.
- *BufferAll*: maximum sized OCM (64K elements), 8-element issue window.
- *software*: software SpMV on the CPU, `-O2` flag.

Figure B3.10 summarizes the SpMV performance from these best-case configurations. Firstly, we note that *BufferAll* cannot be used for 8 of the 12 matrices in the test suite as there is insufficient OCM on the FPGA for these matrices. For the remaining 4 matrices that do fit into OCM, *BufferAll* achieves near-peak (96%) computational efficiency, averaging at 193 MFLOPS. *BufferNone* exhibits performance similar to software SpMV, with an average performance of 61 MFLOPS that corresponds to 30% computational efficiency. Significant deviations from *BufferNone* average performance are visible for matrices 5, 6 and 10. Finally, *NCVCS* outperforms or closely matches the performance of the other methods, with 147 MFLOPS and 73% computational efficiency on

[4] We use NK to refer to $N \cdot 1024$ y elements.

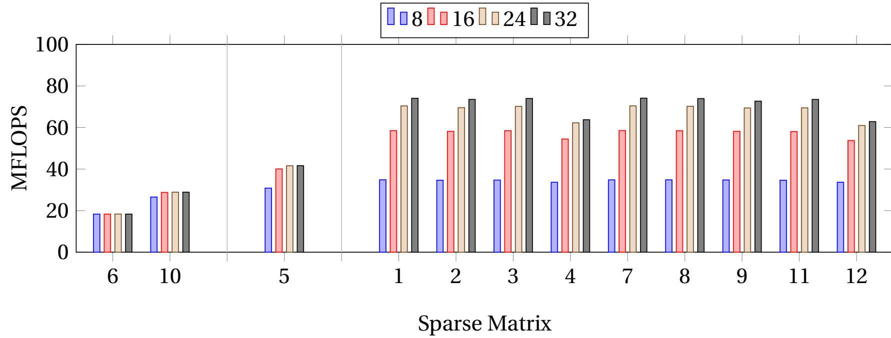


Figure B3.11.: *BufferNone* performance with issue window size.

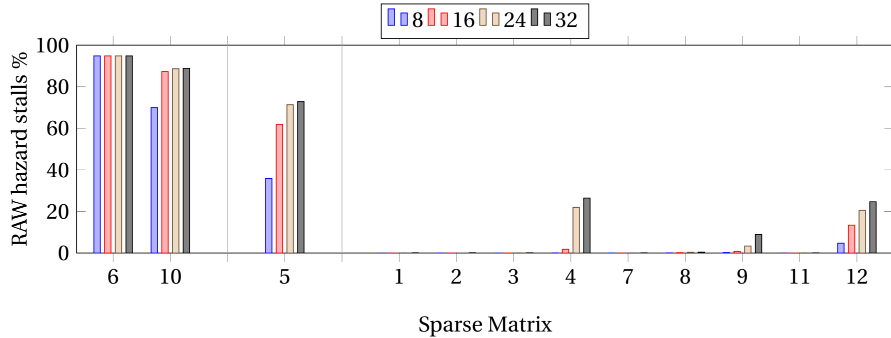


Figure B3.12.: Percentage of RAW hazard stalls with issue window size.

average. Sections B3.6.3 and B3.6.4 provide a deeper analysis of performance and parameters for *BufferNone* and *NCVCS*.

B3.6.3. Analysis of *BufferNone*

As we note in Section B3.3.2, *BufferNone* needs a large issue window to tolerate DRAM's high access latency and enable high performance, but at the risk of increasing RAW hazard stalls. Figure B3.11 plots the performance of *BufferNone* with increasing issue window size. We can classify the matrices into three groups according to their behavior. The first group includes matrices 6 and 10, which get no performance benefit from more memory requests. The

second group includes matrix 5, with limited performance growth going from issue window 8 to 16, and leveling off afterwards. The third and final group includes all other matrices, which experience significant performance growth with increasing window size.

To understand these different behaviours, we plot the percentage of RAW hazard stalls within all frontend stalls in Figure B3.12. A close correspondence between hazard stall growth and lack of performance increase from larger issue window can be observed. Namely, the first group has a high percentage (above 70%) of RAW stalls for even the smallest issue window, while the second group starts out with a moderate percentage of hazard stalls that rapidly increases with a larger issue window. The third group does not experience significant hazard stalls even with large issue window sizes. They experience performance growth up to a window size of 32, after which the memory port becomes saturated and does not deliver more bandwidth for result vector accesses. The expected bandwidth-bound peak performance for *BufferNone* is between 140 to 160 MFLOPS for the matrix suite, whereas the observed RAW hazard-free peak performance is around 70 MFLOPS. This suggests that the DRAM ports are unable to deliver full bandwidth. Permitting out-of-order DRAM operation and prioritizing random vector accesses in shared memory system resources can help increase efficiency, which is left for future work.

B3.6.4. Analysis of *NCVCS*

We now present results and discussion on different aspects of *NCVCS*, including the performance impact of cache size and miss queue size. We also evaluate the cost of preprocessing and show how it compares with the obtained performance and OCM savings. We use a miss queue size of 16 and cold miss skip enabled as baseline parameters for these experiments.

Impact of Cache Size

A key indicator of cache performance is cache miss rate (i.e., the ratio of accesses that do not hit in the cache), which is plotted in Figure B3.13. 9 of the 12 matrices in the suite have `maxAlive` values smaller than the largest cache (32K elements) we can deploy on this FPGA, and together with cold miss skip

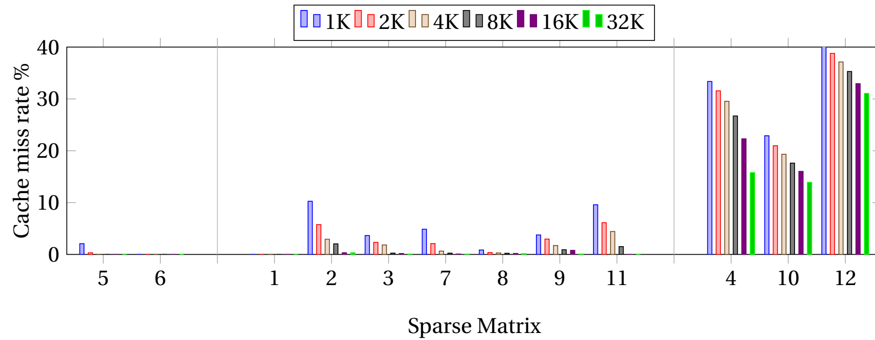


Figure B3.13.: *NCVCS* cache miss rate with cache sizes.

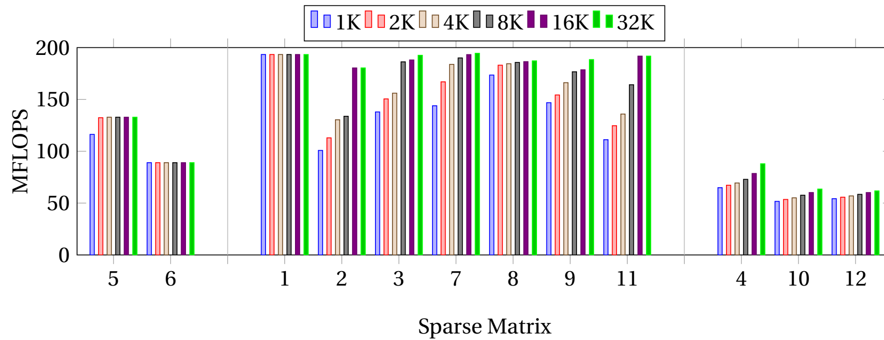


Figure B3.14.: SpMV performance with different cache sizes.

achieve zero or near-zero cache miss rate. This also implies that conflict misses are not a problem for direct-mapped caches with at least `maxAlive` capacity, confirming the results from [61]. The remaining matrices 4, 10 and 12 require cache capacity greater than what we can deploy on this FPGA, and exhibit higher miss rates.

Although cache hit rate is critical to overall accelerator performance, it is not the sole determinant. Figure B3.14 illustrates the SpMV accelerator performance across the matrix suite with different cache sizes, which reveals three groups of behavior. Perhaps most striking are the results for matrices 5 and 6, which have no cache misses but achieve only 50% to 70% of peak performance. This is due to a large number of RAW hazard stalls, which limits the rate at which

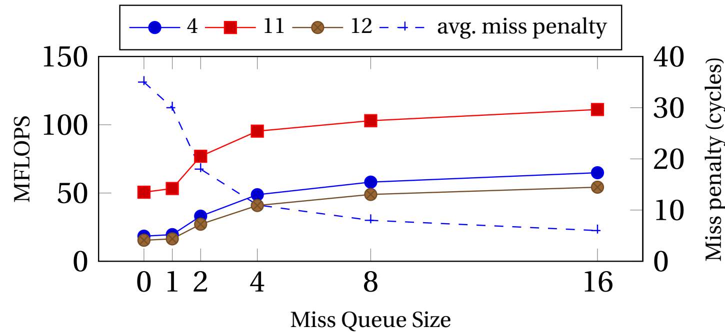


Figure B3.15.: Miss queue size against performance and miss penalty.

instructions can be sent to the reducer and decreases efficiency. Statically or dynamically reordering the matrix can help increase performance in these cases, although we do not investigate their benefit in this work. The three matrices with high cache miss rates (4, 10 and 12) display lower performance averaging around 70 MFLOPS. Although matrices 4 and 10 have similar `maxAlive` values and cache miss rates, matrix 10 benefits less from larger caches due to RAW hazards. Finally, the remaining seven matrices have close to zero in both miss rates and RAW hazard penalties, and achieve on average 95% computational efficiency with a `maxAlive`-element cache or larger.

Impact of Nonblocking Cache

To illustrate how nonblocking helps improve performance with small caches and high miss rates, we plot the performance of matrices 4, 11 and 12 with a 1K-sized vector cache with varying miss queue sizes in Figure B3.15. All of these matrices have `maxAlive` larger than 1K and high cache miss rates. The performance with a miss queue size of 0 (which is a blocking cache) is 18, 51 and 15 MFLOPS for these three matrices, respectively. With a miss queue size of 16, the performance levels rise to 65, 11 and 54 MFLOPS, which correspond to over tripled performance for the larger matrices 4 and 12, and over doubled performance for matrix 11. The increase in performance is directly linked to how the miss penalty decreases with larger miss queue size, which can be examined by dividing the number of frontend stalls (excluding RAW hazards)

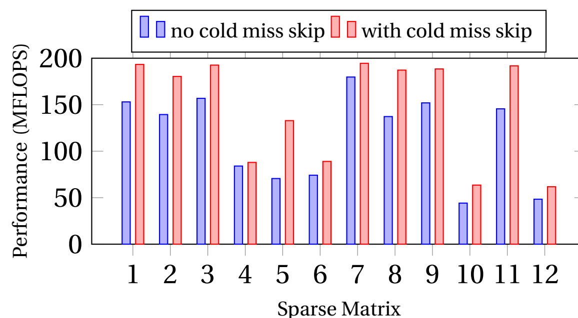


Figure B3.16.: Performance impact of cold miss skip with a 32K-cache.

by the number of cache misses. The average number of stall cycles, which is plotted as a dashed line in Figure B3.15, decreases from 35 for a blocking cache to 6 for a miss queue size of 16.

Impact and Cost of Cold Miss Skip

Figure B3.16 compares the performance of a 32K-element nonblocking cache with and without the cold miss skip optimization across the matrix suite. On average, cold miss skip improves performance by 30%, although the exact benefit varies between 5% to 88%. Due to interactions[5] between matrix structure and how cold miss skip influences latency, the performance benefits cannot be trivially linked to matrix dimensions and sparsity in a nonblocking cache. For instance, matrices 5 and 6 have both zero non-cold cache misses, few elements per row and experience performance degradation due to RAW hazards. However, the performance benefit from cold miss skip is quite different: while matrix 6 performance improves by 20%, matrix 5 almost doubles its performance with 88% improvement.

It is important to remember that the cold miss skip requires marking the start of each row via preprocessing, which has a certain cost. As discussed in Section B3.2.3, the cost of preprocessing can be outweighed by the resulting speedups across multiple SpMV iterations. To characterize the cost of preprocessing

[5]For instance, in a cache without cold miss skip, a nonblocking cold miss followed by hits can mimic the benefits of cold miss skip.

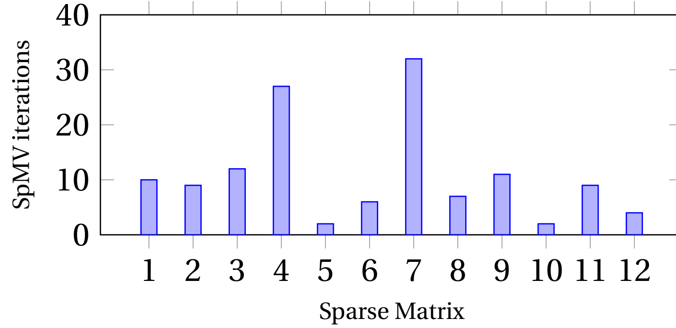


Figure B3.17.: Iterations to break-even for cold miss skip preprocessing.

for cold miss skip, we use the break-even iterations as an indicator. We define S as the SpMV iteration count when cold miss skip peformance improvement matches or exceeds its preprocessing cost, i.e.,:

$$S = \left\lceil \frac{T_{CPU-preproc}}{T_{FPGA-spmv-no-cms} - T_{FPGA-spmv-cms}} \right\rceil$$

Figure B3.17 depicts the number of break-even iterations for a cache with 32K elements, which ranges between 2 to 32 iterations (average 11) for the matrix suite. This compares favorably to the 75 break-even iterations estimated by Toledo [59] for matrix reordering, and can be improved further by using a more powerful CPU or hardware for the preprocessing.

Row Lifetime Analysis and OCM Savings

In Section B3.4.2 we described how the minimum cache size to avoid all capacity misses could be calculated for a given sparse matrix, either using `maxColSpan` or `maxAlive`. The rightmost columns of Table B3.2 list these values for each matrix. However, a vector cache also requires tag and valid bit storage in addition to the cache data storage, which must be taken into consideration. To calculate the OCM savings with vector caching, we use the *BufferAll* OCM requirements as a baseline, which is $64 \cdot m$ bits (one double-precision floating point value per `y` element). For a matrix with m rows, the minimum

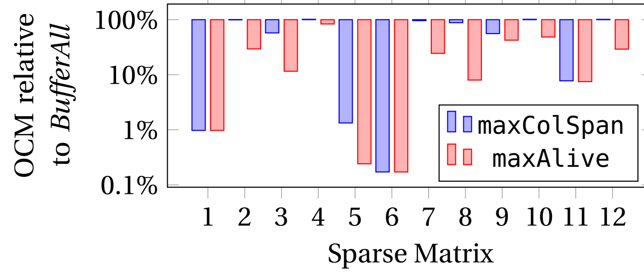


Figure B3.18.: OCM requirements compared to *BufferAll*.

storage requirement in terms of number of bits for a vector cache of W elements, including tag and valid bits, is given by the following equation:

$$(64 + \lceil log_2 m \rceil - \lceil log_2 W \rceil + 1) \cdot W$$

In Figure B3.18 we plot the required OCM of `maxColSpan`- and `maxAlive`-sized caches relative to *BufferAll*. On average, a vector cache offers OCM savings of 76% with `maxAlive` elements and 41% with `maxColSpan` elements. This shows that substantial OCM savings can be achieved by dimensioning the cache based on row lifetime analysis. Matrices 1, 5, 6 and 11, which have most of their elements clustered around the diagonal, already gain significant storage savings with `maxColSpan`. The other matrices require `maxAlive` to shrink the required OCM by at least half, with the exception of matrix 4. For this matrix, even `maxAlive` is only 17.6% smaller than the number of rows, and the OCM benefit from vector caching is limited.

Since preprocessing is required to dimension the vector cache, it is useful to characterize the cost of this preprocessing. If we denote the time cost of a single software SpMV iteration cost as T, our results indicate that the preprocessing costs are on average $0.13T$ for `maxColSpan` and $2.6T$ for `maxAlive`. In comparison, Toledo [59] estimates the cost of Cuthill-McKee matrix reordering to be $1T$ to $3T$ and nested dissection reordering to be $20T$ to $200T$. Furthermore, the preprocessing times will improve by using a better CPU with a more powerful memory system or parallel preprocessing, making it worthwhile to perform row lifetime analysis if a smaller OCM footprint is desired.

B3.7. Conclusion and Future Work

In this paper, we considered the problem of handling random accesses to large volumes of data to enable efficient SpMV implementations. We have proposed a processor-like view of the SpMV reducer, and framed three approaches with the random accesses handled strictly on-chip, strictly off-chip and balanced across off- and on-chip to understand how each approach can achieve high efficiency. To combine the strengths off- and on-chip random accesses, we have proposed *NCVCS*, which combines software preprocessing with a customized nonblocking cache for servicing random accesses.

To compare the *BufferAll*, *NCVCS* and *BufferNone* approaches, we deployed them as part of a FPGA SpMV accelerator, and evaluate their performance and efficiency with a suite of large sparse matrices. Our results indicate that the three methods have similar FPGA logic resource utilization, but significant differences in the required FPGA OCM (BRAMs) and performance. We confirm that *BufferAll* enables highly efficient (96% of peak) accelerators, but the size of the largest sparse matrix it can handle is limited by FPGA OCM. *BufferNone*, which services all random accesses from DRAM, performed the worst among the three schemes with 30% average efficiency and significant RAW hazard penalties. Finally, *NCVCS* either outperforms or performs almost as well as the other schemes for all matrices, with average 73% efficiency. A cache miss rate close to 0% is achieved on matrices with at least `maxAlive`-sized caches. This indicates that `maxAlive` is a good indicator of minimum required capacity and can provide significant OCM savings. Our results also show that the preprocessing necessary for cold miss skip and row lifetime analysis has reasonably low cost, and is well worth the benefits.

As future work, we plan to investigate how well these random access schemes work with parallelization. Our results indicate that the remaining inefficiencies in SpMV acceleration are mostly concentrated around RAW hazard stalls and low bandwidth for DRAM random accesses, which could also be investigated to further improve efficiency.

Paper B4

Hybrid Breadth-First Search on a Single-Chip FPGA-CPU Heterogeneous Platform

Yaman Umuroğlu and Magnus Jahre

Published in
Proceedings of the 2015 International Conference on Field-Programmable Logic and Applications (FPL)

B4. Hybrid Breadth-First Search on a Single-Chip FPGA-CPU Heterogeneous Platform

Abstract. Large and sparse small-world graphs are ubiquitous across many scientific domains from bioinformatics to computer science. As these graphs grow in scale, traversal algorithms such as breadth-first search (BFS), fundamental to many graph processing applications and metrics, become more costly to compute. The cause is attributed to poor temporal and spatial locality due to the inherently irregular memory access patterns of these algorithms. A large body of research has targeted accelerating and parallelizing BFS on a variety of computing platforms, including hybrid CPU-GPU approaches for exploiting the small-world property. In the same spirit, we show how a single-die FPGA-CPU heterogeneous device can be used to leverage the varying degree of parallelism in small-world graphs. Additionally, we demonstrate how dense rather than sparse treatment of the BFS frontier vector yields simpler memory access patterns for BFS, trading redundant computation for DRAM bandwidth utilization and faster graph exploration. On a range of synthetic small-world graphs, our hybrid approach performs 7.8x better than software-only and 2x better than accelerator-only implementations. We achieve an average traversal speed of 172 MTEPS (millions of traversed edges per second) on the ZedBoard platform, which is more than twice as effective as the best previously published FPGA BFS implementation in terms of traversals per bandwidth.

B4.1. Introduction

Graphs as data representations and algorithms that operate on graphs are ubiquitous throughout most scientific domains. Breadth-first search (BFS) is a key building block for exploring graphs, and is fundamental to a variety of graph metrics such as counting connected components, calculating graph diameter and radius [18]. Special cases of BFS such as the independent cascade model (ICS) are used to simulate the spread of information or disease across networks [37].

In order to meet the demand for analysis of ever-larger graphs brought by the Big Data trend, high-performance graph processing is of vital importance. Two key characteristics of BFS (and many other graph algorithms) are *irregular memory accesses* due to data-driven computations on the vertex and edge structure of the graph [13], and *low computation-to-memory ratio*. Thus, BFS performance is commonly memory bandwidth limited. These characteristics make accelerating and parallelizing BFS a major challenge, which has motivated a large body of research on different platforms (Section B4.5).

With the increased focus in recent years on energy efficient computing systems, heterogeneous processing with reconfigurable logic and FPGAs is gaining popularity, including in datacenters [49]. Prior work by Betkaoui et al. [13] and Attia et al. [4] showed that reconfigurable logic for accelerating BFS on large graphs is performance-competitive with multi-core CPUs and GPGPUs. There are three main reasons why FPGAs are suitable for energy efficient BFS. First, the memory architecture can be customized to effectively deal with the irregular memory access patterns. Additionally, FPGAs can be reconfigured to adapt to changing algorithms while consuming less power than CPU and GPGPUs, offering flexible energy efficiency. Finally, BFS performance on large graphs is bound by accesses to high-latency external memory, which is a good fit for achieving high performance on FPGAs via ample parallelism and relatively low clock frequencies.

However, this suitability is dependent on the *availability* of parallel work in BFS to offer high performance, which can lead to significant waste of execution resources. Large real-world graphs often have the small-world property, where the amount of parallelism available changes significantly during BFS (see Section B4.2.2). Our work explores how this change in parallelism can be

exploited in the context of a single-chip FPGA-CPU heterogeneous processor to offer high-performance BFS on large graphs. Through observations on the Boolean matrix-vector representation for BFS, we propose an FPGA BFS accelerator architecture with a stall-free datapath. We describe two architectural variants that treat the BFS frontier as *sparse* or *dense* to show how redundant computations can be traded for bandwidth and increase BFS performance in hybrid execution. Our experimental results on the ZedBoard with synthetic real-world networks indicate that this scheme can utilize up to 78% of the DRAM bandwidth and achieve over twice as many traversals per platform bandwidth compared to previous work.

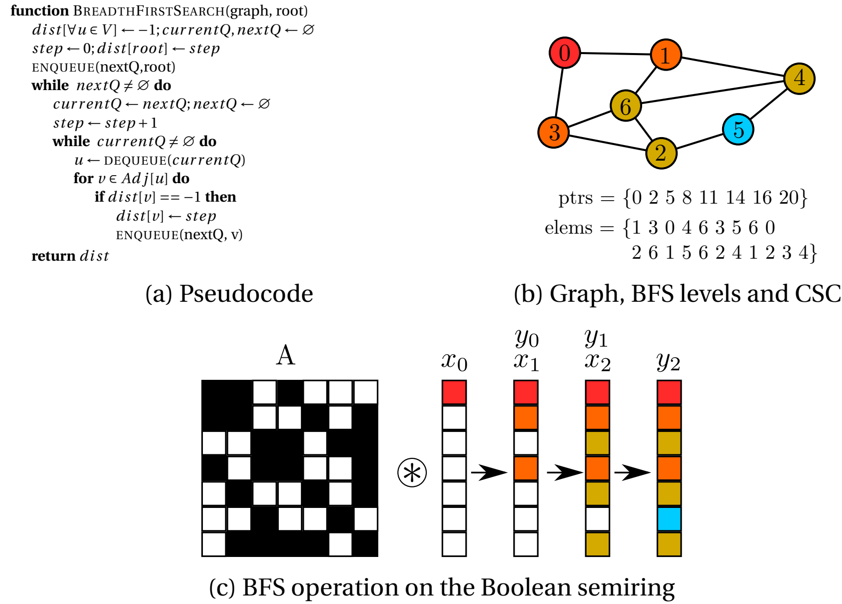
Specifically, our work makes the following contributions:

- A fast FPGA-CPU hybrid BFS architecture with high DRAM bandwidth utilization;
- An analysis of BFS memory request structure and bandwidth utilization for sparse and dense BFS frontier treatment;
- A stall-free BFS datapath using FPGA on-chip RAM to buffer the node visit status;
- A method for decoupling BFS level computation from the traversal to keep the node visit status data small.

B4.2. Background

B4.2.1. Breadth-first search

We consider undirected, unweighted graphs of the form $G = (V, E)$ with sets of $|V|$ vertices (nodes) V and $|E|$ edges E. A breadth-first search begins at a root node v_r contained within the largest connected component $v_r \in V_c, V_c \subset V$ and traverses each edge e_{rj} for every neighbor v_j. As such, the graph is traversed in *levels*, where all nodes at each level are explored before the next level is processed. In line with previous work exploring BFS performance ([4, 13, 33]), we consider the variant of the kernel that produces the distance array (`dist` in



(a) Pseudocode

(b) Graph, BFS levels and CSC

(c) BFS operation on the Boolean semiring

Figure B4.1.: Three representations of the breadth-first search algorithm.

Figure B4.1a), which is the distance (in terms of BFS steps) of each visited node from the root node.

B4.2.2. Sparsity and the Small-World Property

A *small-world* graph is one in which the diameter is small, e.g., “six degrees of separation”, and for social graphs such as Facebook has been shown to be as low as four [7]. Small-world graphs generally exhibit scale-free degree distributions of the form $y = x^a$, i.e., consisting of very few high-degree central “hubs” and very many low-degree nodes that form the periphery [17]. This means that when BFS starts there is a high probability that the root node will only be connected to a few neighboring nodes, and those neighbors connected to a few, and so on. Thus, the first iterations of BFS visit a small percentage of the graph. However, as more and more edges are traversed, the frontier size increases dramatically (see Figure B4.2), constituting a large percentage of the network. As the BFS frontier size is correlated with available parallelism [32],

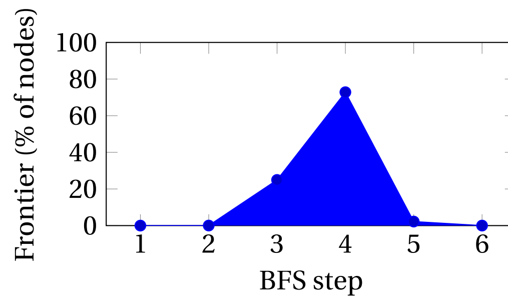


Figure B4.2.: A typical frontier profile for BFS on a small-world graph. Exposed parallelism increases with the frontier size, which approaches the total number of nodes in the graph for the intermediate levels (here 3, 4) of the BFS algorithm.

the same figure is representative for how amenable the different steps are for parallelization.

Large small-world networks are generally sparse (i.e., most nodes are not neighbors). To take advantage of this, the graph is typically stored in sparse adjacency matrix form such as Compressed Sparse Column (CSC)[1], exemplified in Figure B4.1b.

B4.2.3. BFS in the Language of Linear Algebra

Choosing a different representation for an algorithm may expose algorithmic characteristics that can be exploited for accelerator design. Towards this end, we will be using the "matrices over semirings" concept [38] to express BFS as a linear algebra operation. The core idea is to substitute the number data type and the operators for multiplication and addition in linear algebra to express a variety of algorithms as matrix-vector operations.

Specifically, we will make use of the matrix-times-vector operation on the Boolean semiring to perform BFS. In practice, this operation "multiplies" a

[1]For an undirected graph, the Compressed Sparse Row (CSR) representation is equivalent to CSC. Our work assumes CSC and column-major traversal.

binary matrix and a binary vector, with the regular multiply and add operations substituted with the Boolean AND and OR operators, respectively. To disambiguate from regular matrix-vector multiply over real numbers, we will use $\circledast$ to denote this operator. As illustrated in Figure B4.1c, each $y_t = A \circledast x_t$ operation corresponds to a breadth-first step, and each result vector y_t is the representation of the visited nodes in the graph after step t. The matrix A in the operation is the adjacency matrix of the graph, while the initial input vector x_0 is initialized to all zeroes, except a single 1 at the location of the root node. The result vector y_t is used as the input vector x_{t+1} of the next step, which in turn generates more visited nodes in its result vector until the result converges (i.e., no more nodes can be visited).

We note that the properties of the Boolean semiring can be exploited here to perform less work: x_t elements that are zeroes can be simply skipped since AND operation with a 0-input will always return a 0. Furthermore, only a subset of the 1-entries (those that were produced in the previous step) in x_t may actually produce new 1-entries in y_t. From a BFS standpoint, these observations correspond to only the newly-visited (frontier) nodes doing useful work. In terms of linear algebra, we can say that the x_t vector can be treated as *sparse*, though dense treatment is functionally correct.

As we will show in Section B4.3.2, this representation enables us to view BFS in a way that permits trading redundant computations for higher memory bandwidth. Additional advantages of this approach include the potential for easier integration with software that use linear algebra as a building block, as well as the ability to apply memory system optimizations designed for iterative sparse linear algebra (e.g., [61, 63]).

B4.3. Hybrid BFS on an FPGA-CPU Hybrid

Our accelerator system specifically targets in-memory small-world graphs, where breadth-first search exhibits a characteristic profile in terms of the frontier size explained in Section B4.2.2. Since the amount of parallelism available during BFS is closely correlated with the frontier size, the opportunity for heterogeneous mapping onto different types of processing elements presents itself.

A single CPU core can be used for the steps with small frontiers, while a high-throughput accelerator can be used for the steps with large frontiers. Following the example of [33] for a CPU-GPGPU system, we adopt the hybrid approach for a single-chip FPGA-CPU heterogeneous processor. The primary advantage of this platform is the low cost of switching execution modes back and forth, effectively adapting to the amount of parallelism available in small-world BFS. As justified by our results in Section B4.4.2, our strategy is to start the BFS kernel on the CPU, switch to the FPGA accelerator after a few steps to rapidly explore most of the graph, then switch back to the CPU for the last few steps.

In the following sections, we will first develop several ideas around implementing BFS on the Boolean semiring, and afterwards describe the architecture of our accelerator system.

B4.3.1. Decoupled Distance Generation

The Boolean semiring matrix-vector representation of BFS given in Section B4.2.3 is very lean in terms of storage requirements, which makes it suitable for a hardware accelerator implementation. Specifically, the x and y vectors require only one bit of storage per graph node. Since y will be random-accessed due to matrix sparsity, keeping the range and volume of the data to be random-accessed to a minimum is advantageous for performance. Unfortunately, iteratively invoking $\circledast$ is not sufficient[2] for BFS as it only generates the node visited status and not the `dist` array.

We address this shortcoming by introducing a separate *distance generation* (`DistGen`) step after each $\circledast$ invocation. A node i has distance t if it was visited during the BFS step t, and we know that a node cannot go from being visited to unvisited. Thus, we can conclude that the node has distance t if it is unvisited in x_t and visited in y_t, or $dist[i] = t \iff (x_t[i] == 0 \land y_t[i] == 1)$. To generate the distance information, it is sufficient to examine the input and output vectors of each BFS step, after each step is finished. This array compare operation is decoupled from the regular BFS step and can be easily parallelized

[2] Although using the tropical semiring $(R \cup \{\infty\}, min, +)$ would remedy this, each vector element in the tropical semiring is a number and loses the leanness/storage advantages of the Boolean representation.

```
function DistGen(dist[], level, x, y)
    updates ← 0
    for i ← 0..N − 1 do
        if x[i] == 0 & y[i] == 1 then
            dist[i] ← level
            updates ← updates + 1
    return updates

function BFSAsLinearAlgebra(A, root)
    x, y ← [0, 0, ..0];
    dist ← [−1, −1, .. − 1]
    x[root] ← 1
    y[root] ← 1
    dist[root] ← 0
    level ← 1
    converged ← 0
    while !converged do
        y ← A ⊛ x
        converged ← DistGen(dist, level, x, y)
        y ← x;
        level ← level + 1
    return dist
```

Algorithm B4.1: BFS with $\circledast$ and `DistGen`.

or implemented in hardware (Section B4.3.3) to reduce its performance overhead. The complete BFS algorithm expressed with $\circledast$ and `DistGen` is listed in Algorithm B4.1.

B4.3.2. Trading Redundant Computation for Bandwidth

As traversal of sparse graphs involves little actual computation and is known to be a memory-bound problem, delivering graph data with high bandwidth is critical for accelerator performance. Therefore, a careful analysis of memory access patterns is a critical step for designing a BFS hardware accelerator. In our accelerator as with many other systems, the BFS inputs are stored in and accessed from DRAM. Due to the inherent latency and three-dimensional organization of modern DRAM chips, three features are key to achieving a substantial portion of available DRAM bandwidth: high request rate to mitigate

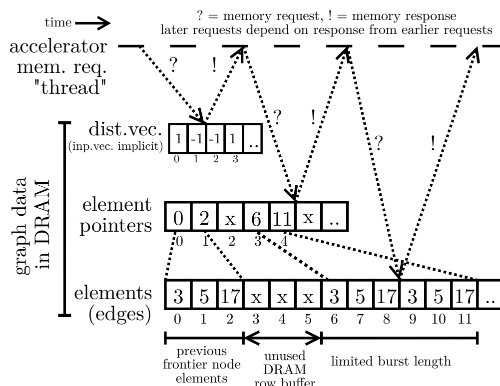


Figure B4.3.: Structure of memory requests for sparse x_t.

latency, large bursts, and a sequential access pattern to maximize row buffer hits.

In Section B4.2.3, we observed how the input vector x_t could be treated as sparse to avoid redundant work, but the linear algebra notation tells us that we can treat x_t as dense and still get a correct BFS result. In a hybrid accelerator operating on small-world networks, this seemingly unorthodox idea of treating the BFS input frontier as dense and performing redundant work (revisiting the entire graph) can be actually beneficial for overall performance due to simpler DRAM access patterns. How we treat the x_t vector influences how the matrix A data will be accessed, and in turn, with how much bandwidth.

Figure B4.3 depicts how treating x_t as sparse influences the memory requests to the matrix data. The accelerator must first obtain a node index that is a member of the frontier by reading `dist`, then obtain this node's start and end pointers, and finally obtain the list of adjacent edges using these pointers. Visible here is the dependency of requests on responses to previous requests; this is typical of applications with indirect accesses, and sparse treatment of x_t leads to two levels of indirection. More concretely, the effects are three-fold. The first is the limitation of the request rate by the response rate. Secondly, the length of read bursts to the `elems` array are limited by the number of edges in the node. Finally, even though the reads to the `elems` array are sequential, there may be large gaps in between the used portions of the array due to frontier nodes being far apart, causing parts of the DRAM row buffer to go unused. This becomes

especially prominent with several accelerators requesting different parts of the edges array in parallel. Although this method avoids doing redundant work, these three effects can dramatically decrease DRAM bandwidth utilization, especially for platforms whose memory systems cannot handle a large number of outstanding requests.

In contrast, if we treat x_t as dense and consider every node of the graph, the access pattern of A becomes significantly simpler. In particular, we can simply read out the entire matrix, which can be done with maximum-length burst read operations and without having to wait for responses from previous requests. This is a much simpler and more suitable access pattern for achieving high DRAM bandwidth. Memory bandwidth is, of course, only half the story; the amount of redundant work performed by treating x_t as dense is nontrivial – the smaller the frontier, the more redundant work will be performed. However, since we are building a hybrid CPU-FPGA system where the accelerator handles the BFS steps with large frontiers, the overhead of redundant work is less significant. In fact, as described in Section B4.4, our experiments on the Zynq platform with scale-free graphs show that the dense x_t treatment always outperforms the sparse treatment in this hybrid approach.

B4.3.3. Processing Element Architecture

Based on the ideas from Sections B4.3.1 and B4.3.2, we now describe a hardware architecture for BFS. To compare the effects of sparse and dense x_t treatment described in Section B4.3.2, we consider two processing element (PE) variants.

Dense x variant

The architectural overview of a dense x_t processing element (PE) is illustrated in Figure B4.4. The architecture is organized in a data-flow manner, and modularized into three main components: a *backend*, which connects to the DRAM via the system interconnect, a *frontend* for performing the $\circledast$ operator, and a *distance generator*. The backend is responsible for all interaction with main memory, which includes reading out the matrix and vector data, and writing updates to the distance vector. The matrix and vector data are requested by the

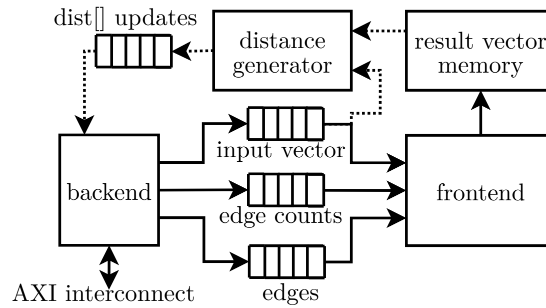


Figure B4.4.: Architecture overview of a dense x_t processing element. Dotted lines are active only during the distance generation operation.

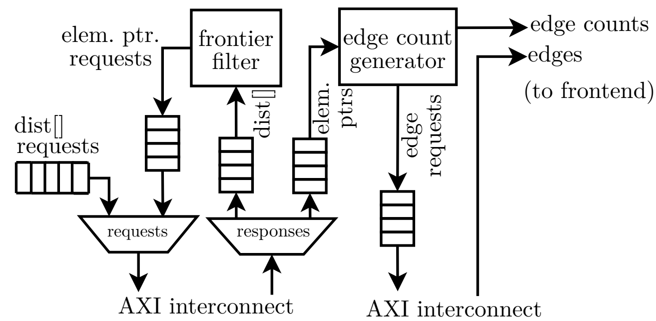


Figure B4.5.: Backend architecture for sparse x_t variant.

backend in large bursts and made available to the frontend, which performs the BFS step operation and updates the result vector memory. Concretely, the frontend simply writes 1s to result memory addresses indicated by the edges whenever the input vector is 1. The edge counts data is used to determine when to read a new input vector element. An input vector value of 0 implies edges from a yet-unvisited node, and this data is simply dropped without further operation. The control and status interface of the accelerator is provided through memory-mapped registers, which are not shown in the figure.

Sparse x variant

In terms of the overall architecture, the sparse x_t variant is identical to the dense one except that it does not need the input vector FIFO (since the sparse treatment implies all x_t values are ones). However, as illustrated in Figure B4.5, the internals of the backend are substantially different from the dense variant. The sparse input vector (or frontier indices) is generated internally by a *frontier filter*, which scans the values of the distance array and emits the indices whole values were written in the previous BFS step. Afterwards, the start- and end-pointers of each generated index are requested. These pointers are used to request the edge data for the node, and also to produce the edge count data for the frontend.

Stall-free y Writes

To keep the accelerator running without stalls, it is important that the frontend is able to consume data as fast as the backend is producing it. The result vector y_t is random-accessed by the frontend during the $\circledast$ operation, since the accessed node locations depend on the visited graph edges. Thus, we can abstract the functionality of the frontend as *handling a stream of writes to random addresses*. If the result vector is stored in DRAM, the write request buffers of the interconnect and memory controller can fill up and stall the entire accelerator. To avoid this, our solution exploits the leanness of vector representations. Since our approach requires us to keep only a single bit per graph node, we can effectively utilize dual-port FPGA on-chip RAM to provide two very fast, fine-grained random accesses per cycle. Although this limits the largest graph size we can process, the on-chip RAM capacity of modern FPGAs is quite large and graphs with millions of nodes can still be processed in this manner. Another option is to explore a single BFS step of a large graph across more than one execution by partitioning the matrix along rows, which we do not explore in this work.

Table B4.1.: Characteristics of the ZedBoard.

CPU core	Dual ARM Cortex-A9, 666 MHz
CPU cache	32 KB L1D+L1I, 512 KB L2
DRAM and bandwidth[3]	512 MB DDR3, ~ 3.2 GB/s
FPGA logic resources	13300 logic slices, 53200 slice LUTs
FPGA on-chip RAM	560 KB (BRAM)

[3] Controller efficiency is ~ 75% of the theoretical max of 4.2 GB/s ([72], Section 22.3.2)

Distance Generator

After each BFS step is finished, the distance generator is invoked to implement `DistGen` as described in Section B4.3.1. This involves comparing the input and result vectors and finding the nodes that went from being unvisited to visited. The indices of these nodes is passed to the backend for actually writing the current BFS distance to the corresponding memory locations, and also for updating the x vector for the next BFS step for the dense variant.

B4.3.4. Accelerator System Implementation

Our accelerator system is built and deployed on the ZedBoard platform with the Xilinx Zynq Z7020 FPGA-CPU hybrid [72], whose characteristics are shown in Table B4.1. The accelerator components were first built in Chisel [6]. Verilog descriptions were then generated using the Verilog backend, and imported into Vivado as IP blocks. Vivado IP integrator (version 2014.4) was used afterwards to build the accelerator system, including Xilinx-provided IP blocks for the result memory block RAM (BRAM) and AXI interconnect. The 64-bit AXI high-performance (HP) slave ports, which are capable of utilizing about 75% (3.2 GB/s) of the DRAM bandwidth, are used to feed the accelerators with data and write back results.

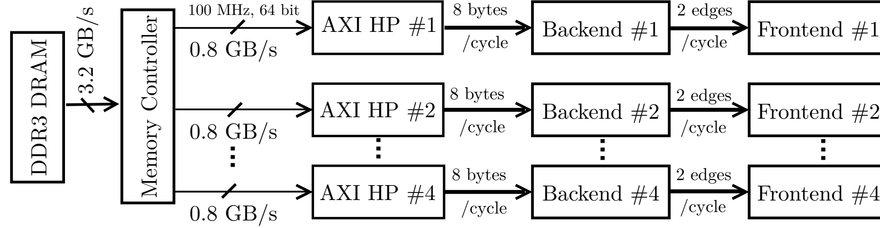


Figure B4.6.: Rate-balanced design for the ZedBoard.

Parallelism and Rate-Balancing

As parallelism is key to achieving high performance with FPGA accelerators, we explore the graph with parallel PEs in our system. We use row-wise partitioning of the input matrix to ensure that the random-access range of each partition fits within the result vector memory of one PE. The entire input vector is read from DRAM by all PEs during a step. After the step, each PE updates a portion of the input vector with its result (during the `DistGen` operation). We use the rate-balancing ideas for FPGA sparse matrix-vector multiplication from [63] to estimate the number of PEs required to consume the available DRAM bandwidth in the platform. Every cycle, each PE backend can fetch a maximum of 8 bytes through the interconnect, and each frontend can process up to two edges of 4 bytes each by issuing writes to dual-port result memory. Assuming $F_{clk} \approx$ 100 MHz, we can obtain a rate-balanced design by attaching one PE to each of the four AXI HP ports, as shown in Figure B4.6.

Software BFS Implementation and FPGA-CPU Switching

The software BFS variant runs on a single Cortex-A9 core inside the Zynq system with caches enabled, and uses the bitmap optimization [2] to track node visit status for better performance. Since the visit status bitmap corresponds to an input vector, it makes switching between CPU and FPGA easier. Switching from CPU to FPGA execution requires updating the PE result memories with the node visit status. The accelerator itself can be used to do this switching by using an identity matrix as the graph, and the visit status bitmap as the input vector. Switching back to the CPU after FPGA execution requires a frontier queue to be

reconstructed from the distance vector, which is also a highly data-parallel task (i.e., search through an array to find indices with a given value). We include a simple hardware accelerator to keep the performance overheads from the switching to a minimum.

Method Switching in Hybrid

After each BFS step is finished, the software uses a simple model to decide which method[3] should be used for the next step. The hybrid BFS starts execution in software, and switches to FPGA execution when the predicted BFS step time for the FPGA is shorter. We exploit the predictability of the dense variant (see Figure B4.7) to model its execution clock cycles with the following formula:

$$T_{\text{step}} = T_{\text{DistGen}} + T_{\circledast} = \frac{1}{\beta} \cdot \frac{nodes}{\#PE} + \frac{1}{\beta}\left(nodes + \frac{edges}{2 \cdot \#PE}\right)$$

where β is the fraction of utilized bandwidth. The FPGA execution continues until the frontier size drops to below θ% of all graph nodes. Afterwards, the software BFS takes over until the search is terminated. β, θ are determined empirically.

B4.4. Results

We now present the results from the experimental evaluation of our accelerator system. For BFS performance testing, we use synthetically generated RMAT graphs with the Graph500 benchmark parameters (A=0.57, B=0.19, C=0.19) in line with previous work [4, 13]. We refer to an RMAT graph with scale S (2^S nodes) and edge factor E ($E \cdot 2^S$ edges) as RMAT-S-E. To avoid reporting results from trivial searches, we only consider nodes which are in the largest connected component in the graph, whose size is $O(N)$ for RMAT graphs. Due to the limited amount of BRAM available on the Zynq, we were unable to evaluate our approach for graphs larger than scale 21 (two million nodes), but our technique can be applied to larger graphs on bigger FPGAs (e.g., up to scale 29 on the largest UltraScale+ Virtex 7).

[3] As our results in Section B4.4.2 indicate that the dense variant outperforms the sparse, we only consider software-dense hybrid BFS.

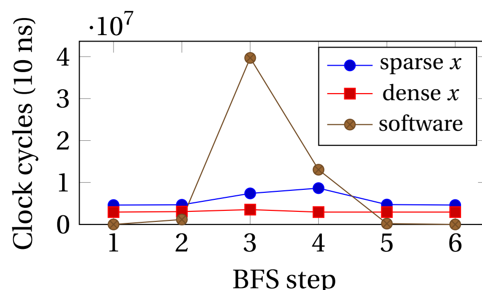


Figure B4.7.: Execution time per step of accelerator and software BFS on an RMAT-19-32.

B4.4.1. FPGA Resource Utilization and Clock Frequency

The area and timing results from Vivado 2014.4 for both the dense and sparse variants are similar. For a 4 PE design, the accelerator system can run at up to 150 MHz and uses about 39% of the FPGA logic resources, and 97% of the FPGA on-chip RAM (BRAM). 82% of the BRAM is used for result vector memory, and about 15% for the FIFOs in the memory system and inside the PEs.

B4.4.2. Comparing Software, Sparse and Dense BFS

To motivate the hybrid BFS solution, we start by comparing the performance of the sparse and dense accelerator variants with software BFS. We perform BFS on RMAT-19-32 with equal number of PEs (4) and clock frequency (100 MHz) for both accelerator variants, and plot the number of clock cycles taken for each BFS step (including distance generation) in Figure B4.7. Our first observation here is that there is no single best method; as expected, the fastest method differs from step to step. The goal of the hybrid scheme would be to choose the fastest method for each step, which can be deduced from this plot. We can see that exploring steps 1 and 2 with the CPU, switching to the dense x accelerator for steps 3 and 4, then switching back to the CPU for the last two steps gives the fastest execution. The dense variant outperforming the sparse implies that *the benefits from the increase in DRAM bandwidth is larger than the cost of redundant data fetches* for the middle steps. The small-world property means

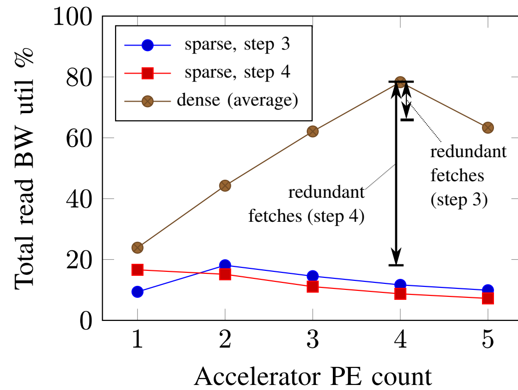


Figure B4.8.: Aggregate read bandwidth utilization for steps 3 and 4 of an RMAT-19-32.

that a large portion of the graph nodes are visited in the middle steps, which means less redundant work for the dense variant.

To better understand why the dense variant outperforms the sparse during the large-frontier steps, we plot the aggregate read bandwidth utilization (compared to the memory link capacity of 32 bytes/cycle) for the BFS steps 3 and 4, for both variants with increasing PE counts. The reason we vary the PE count is to reveal the effects of increased memory pressure from more parallel requests. For the sparse variant, we actually observe that the total utilized bandwidth decreases by adding more PEs. The particularly low utilization in step 4 is likely caused by the frontier being larger (3x) than the step 3 frontier, causing parallel PE requests all across the edges array and leading to many DRAM bank conflicts and row buffer misses. On the other hand, the bandwidth utilization for the dense variant is much better than the sparse and increases almost linearly with PE count, peaking at 78% for 4 PEs, and does not vary between the two steps. Adding more than 4 PEs requires the AXI HP ports to be shared and decreases bandwidth utilization and performance. Even when we account for the significant cost of redundant data fetches in the dense variant (see annotations in Figure B4.8), we can see that the dense variant has better bandwidth utilization than the sparse variant.

It is important to keep in mind that the tradeoffs between these methods will depend on the particular platform, graph and number of PEs being used.

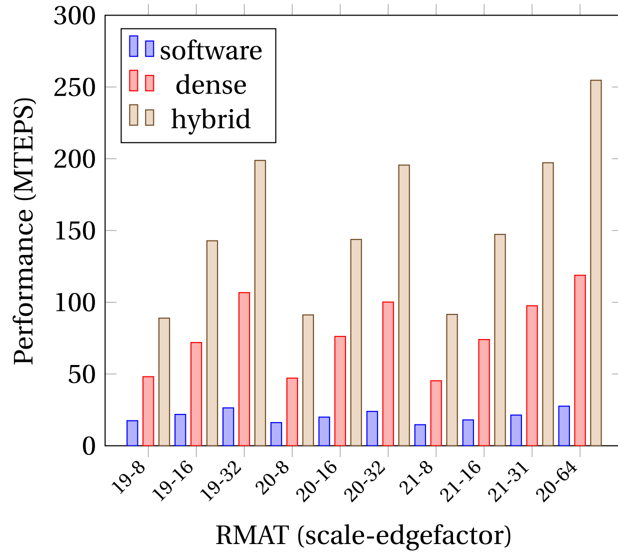


Figure B4.9.: BFS performance on a range of RMAT graphs.

However, the performance depicted in Figure B4.7 is representative of all our experiments on the ZedBoard with RMAT graphs; the software solution is best for the start and the end, and the dense variant always outperforms the sparse variant for the middle steps. We will therefore omit results for the sparse variant from the rest of our discussion.

B4.4.3. Hybrid BFS Performance

We now report results for software-only, dense frontier accelerator-only and hybrid BFS approaches on a range of RMAT graphs. The hybrid BFS works as described in Section B4.3.4. The accelerators are clocked at 150 MHz. We use $\beta = 0.78$ from Figure B4.8 and empirically determine that $\theta = 5\%$ performs close-to-ideal switching. We measure performance in MTEPS (millions of traversed edges per second), which is obtained by dividing the graph edge count by the execution time. The results are averaged over 16 BFS operations started from randomly chosen root nodes within the largest connected component for each graph.

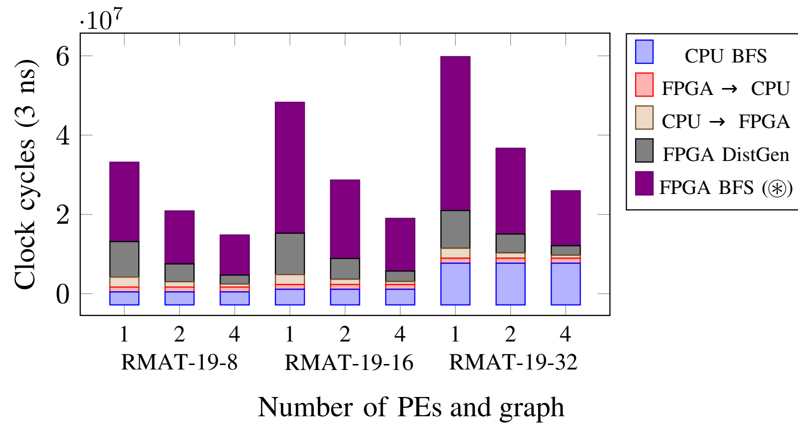


Figure B4.10.: Hybrid execution time breakdown.

Figure B4.9 summarizes the BFS performance for a range of RMAT graphs. The software-only BFS has a performance of 22 MTEPS on average. Since none of these graphs fit into the CPU cache, a large number of data cache misses (~%25 miss rate) degrade the performance. The accelerator is 3.9x as fast on average as the software-only BFS. Given the 4x frequency advantage of the CPU and large amounts of redundant data fetches and work performed by the accelerator, this speedup further supports the claim that slow-clocked but parallel FPGA accelerators are suitable for irregular, memory-bound applications. Finally, the hybrid method combines the “best of both worlds” and outperforms both the accelerator-only and software-only BFS with speedups of 2x and 7.8x, respectively. The performance of the accelerator is correlated with the graph edge factor, with the hybrid BFS achieving a maximum of 255 MTEPS for edge factor 64.

B4.4.4. Hybrid Execution Time Breakdown and Scaling

Figure B4.10 provides a breakdown of the execution time for the hybrid BFS on scale 19 graphs with different edge factors. To show how performance scales with parallel PEs, we provide data points for 1, 2 and 4 PEs. We observe that the execution time of FPGA `DistGen` and ⊛ operations are almost halved by doubling the PE count. This is consistent with the bandwidth scaling in Fig-

ure B4.8. The FPGA ↔ CPU switching overheads account for about 10% of the execution time on average. The overhead of the `DistGen` operation decreases with increasing graph edge factor, and ranges between 9%-15% of the total execution time for 4 PEs. Since the CPU execution time does not vary with increased PE count, we observe an “Amdahl’s Law”-like trend in performance scaling. With 4 PEs on RMAT-19-32, the CPU execution time accounts for 30% of the total, and would eventually become a performance bottleneck on a larger system with more bandwidth and PEs. More performance in the CPU-explored steps can be achieved by using a faster CPU with a better memory system.

B4.4.5. Comparison to Prior Work

As most works targeting high-performance BFS use MTEPS as a metric, comparing raw traversal performance is possible but the available memory bandwidth in the hardware platform sets a hard limit on achievable BFS performance. Our experimental results are from a ZedBoard with much less (about 1/20th of those in Table B4.2) DRAM bandwidth than platforms in prior work and thus is comparatively slow. However, as indicated by our results in Sections B4.4.4 and B4.4.2, the performance of our method scales well as more bandwidth becomes available. Low memory bandwidth is not an inherent problem with single-chip FPGA-CPU solutions and we believe more powerful versions of these devices, such as the Xilinx UltraScale+ Zynq and Altera Stratix 10 SoCs, are likely to be deployed for high-performance computing and graph analysis in the near future.

Taking into account the memory-bandwidth-bound nature of BFS on sparse graphs, we use *traversals per unit bandwidth* as a metric to enable fair comparison with prior work, which we obtain by dividing traversal speed in MTEPS by external memory bandwidth in GB/s. Table B4.2 presents a comparison with several related works on reported average BFS performance, available DRAM bandwidth and traversals per bandwidth over RMAT graphs similar to the ones we used. Our method is more than twice as effective in terms of traversals per bandwidth compared to the next-best solution, which is also on an FPGA.

Table B4.2.: Comparison to prior work.

Work	Platform	Avg. MTEPS	BW (GB/s)	MTEPS/BW
[13]	Convey HC-2	~1600	80	20
[4]	Convey HC-2	~1900	80	24.375
[33]	Nehalem+Fermi	~800	128	6.25
This work	ZedBoard	172	3.2	**53.7**

B4.5. Related work

Fast graph traversal has been approached from a range of architectural methods from general-purpose CPU and multicore/supercomputing approaches exposing parallelism [2, 8, 12, 16, 41, 73] to graphics processing units (GPUs) [1, 14, 43, 45], as well as hybrid CPU-GPU methods [33], to more recent methods taking advantage of reconfigurable hardware [4, 13, 23, 67]. Many principles are constant across architectures, for example, the performance hit associated with irregular memory accesses similarly affects GPU systems, single and multi-CPU systems, and FPGAs. For brevity in the following text, we focus primarily on FPGA-based related work.

Early reconfigurable hardware approaches attempted to solve graph traversal problems on clusters of FPGAs [5, 21], but were limited by graph size and synthesis times because the reconfigurable logic was used to model the graph itself. Recent works have implemented optimizations for BFS and other irregular applications on multi-softcore processors in FPGAs, yielding promising results [16, 54, 67]. More closely related research to ours has explored highly parallelized processing elements (PEs) and decoupled computation-memory [4, 13]. Observing that the execution time of BFS on small-world networks is dominated by the intermediate levels, Betkaoui et al. [13] decouple the communication and computation elements in an FPGA to maintain throughput of irregular memory accesses, arguing that on-chip memories in FPGAs are too small for contemporary graphs. In the same vein, the authors of [4] present optimizations to BFS that essentially merge the first two request-response arrows in Figure B4.3 and report increased performance due to fewer memory requests.

Parallel BFS implementations on GPUs are numerous [1, 14, 43, 45], with re-

search typically focusing on level-synchronous [33, 45] or fixed-point [73] methods. CPU and multiprocessor-based approaches attempt to hide memory operation latencies with caches, but for irregular algorithms such as BFS this is not effective. Other techniques, such as using cache-efficient data structures are often employed [1, 41]. A notable approach by Agarwal et al. [2] makes locality optimizations on a quad-socket system in order to reduce memory traffic and proposed a bitmap to keep track of visited nodes in a compact format (1 bit per node). This study is widely cited for the latter aspect [8, 14, 45], and in our work we adopt this optimization as well. Beamer et al. [8] argue for reducing the number of edges traversed through a direction-optimizing approach that switches between parent-child and child-parent traversal depending on frontier heuristics.

The CPU-GPU hybrid method of [33] is similar to our work in that a switching approach is employed: a queue-based method is efficient when the frontier size is small and a read-based method that sequentially reads the adjacency list is more efficient when the frontier size is large (as is typical for small-world graphs). Our approach differs in two main points: we can switch back to executing on the CPU owing to tight CPU-FPGA integration (which is avoided in [33] due to high overhead) and we exploit the frontier density in the middle BFS steps for trading redundant computations for increased bandwidth. Finally, the authors of [53] emphasize the benefits of sequential patterns for achieving high storage device bandwidth in graph algorithms, and propose an edge-centric scatter-gather framework with streaming partitions. Although it also exploits redundant work for increased bandwidth, this approach sacrifices some efficiency (e.g., random accesses to nodes) to achieve general applicability for in- and out-of-memory graphs. It is also intended for cache-based multiprocessors, whose efficiency on BFS are limited. To the best of our knowledge, ours is the first approach to consider redundant work-bandwidth tradeoffs in BFS for a hybrid FPGA-CPU system, or from a hardware-near perspective in general.

B4.6. Conclusion and Future Work

In this work, we have presented a hardware-accelerated BFS architecture for FPGA-CPU hybrid systems that can effectively take advantage of the varying degree of parallelism in small-world graphs. Viewing BFS as a matrix-vector

operation on the Boolean semiring, we showed how the volume of the random-accessed data can be reduced significantly and kept in on-chip RAM for a stall-free BFS datapath. Another revelation from the representation, the idea of treating the input vector as dense instead of sparse, allowed us to trade redundant computations for increased DRAM bandwidth. Our experiments on the ZedBoard platform suggest that the hybrid system performance scales well with increased bandwidth and outperforms previous techniques in terms of traversals per bandwidth.

Future work will include evaluating this technique on more powerful FPGA-CPU platforms and exploring more graph algorithms with the matrices over semirings idea. The source code for our accelerator system can be obtained from `http://git.io/veGTL` for further investigations.

Bibliography

[1] U. A. Acar, A. Charguéraud and M. Rainey. *Fast Parallel Graph-Search with Splittable and Catenable Frontiers*. Tech. rep. 2015.

[2] V. Agarwal, F. Petrini, D. Pasetto and D. A. Bader. 'Scalable graph exploration on multicore processors'. In: *Proc. of the 2010 ACM/IEEE Intl. Conf. for High Perf. Computing, Netw., Storage and Analysis*. 2010.

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams et al. *The landscape of parallel computing research: A view from Berkeley*. Tech. rep. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[4] O. G. Attia, T. Johnson, K. Townsend, P. Jones and J. Zambreno. 'CyGraph: A Reconfigurable Architecture for Parallel Breadth-First Search'. In: *Parallel & Distributed Processing Symp. Workshops (IPDPSW)*. 2014.

[5] J. W. Babb, M. Frank and A. Agarwal. 'Solving graph problems with dynamic computation structures'. In: *Photonics East'96*. 1996.

[6] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek and K. Asanović. 'Chisel: constructing hardware in a Scala embedded language'. In: *Proc. of the Design Automation Conference*. 2012.

[7] L. Backstrom, P. Boldi, M. Rosa, J. Ugander and S. Vigna. 'Four degrees of separation'. In: *Proc. of the 4th Annual ACM Web Science Conf.* 2012.

[8] S. Beamer, K. Asanović and D. Patterson. 'Direction-optimizing breadth-first search'. In: *Scientific Programming* 21.3-4 (2013).

[9] W. Bechtel and A. Abrahamsen. *Connectionism and the mind: Parallel processing, dynamics, and evolution in networks*. Blackwell Publishing, 2002.

[10] N. Bell and M. Garland. *Efficient sparse matrix-vector multiplication on CUDA*. Tech. rep. Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[11] N. Bell and M. Garland. 'Implementing Sparse Matrix-vector Multiplication on Throughput-oriented Processors'. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009. DOI: 10.1145/1654059.1654078. URL: http://doi.acm.org/10.1145/1654059.1654078.

[12] R. Berrendorf and M. Makulla. 'Level-synchronous parallel breadth-first search algorithms for multicore and multiprocessor systems'. In: *FUTURE COMPUTING 2014, The Sixth Intl. Conf. on Future Computational Technologies and Applications*. 2014.

[13] B. Betkaoui, Y. Wang, D. B. Thomas and W. Luk. 'A reconfigurable computing approach for efficient and scalable parallel graph exploration'. In: *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd Intl. Conf. on*. 2012.

[14] M. Bisson, M. Bernaschi and E. Mastrostefano. 'Parallel Distributed Breadth First Search on the Kepler Architecture'. In: *arXiv preprint arXiv:1408.1605* (2014).

[15] D. Buono, J. A. Gunnels, X. Que, F. Checconi, F. Petrini, T.-C. Tuan and C. Long. 'Optimizing Sparse Linear Algebra for Large-Scale Graph Analytics'. In: *Computer* 48.8 (2015).

[16] M. Ceriani, G. Palermo, S. Secchi, A. Tumeo and O. Villa. 'Exploring Manycore Multinode Systems for Irregular Applications with FPGA Prototyping'. In: *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual Intl. Symp. on*. 2013.

[17] D. Chakrabarti, Y. Zhan and C. Faloutsos. 'R-MAT: A Recursive Model for Graph Mining.' In: *SDM*. Vol. 4. 2004.

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein et al. *Introduction to algorithms*. Vol. 2. 2001.

[19] A. Cosoroaba. *Achieving High Performance DDR3 Data Rates*. Aug. 2013. URL: http://www.xilinx.com/support/documentation/white_papers/wp383_Achieving_High_Performance_DDR3.pdf.

[20] E. Cuthill. 'Several strategies for reducing the bandwidth of matrices'. In: *Sparse Matrices and Their Applications*. Springer, 1972, pp. 157–166.

[21] A. Dandalis, A. Mei and V. K. Prasanna. 'Domain specific mapping for solving graph problems on reconfigurable devices'. In: *Parallel and Distributed Processing*. 1999.

[22] T. A. Davis and Y. Hu. 'The University of Florida Sparse Matrix Collection'. In: *ACM Trans. Math. Softw.* 38.1 (Dec. 2011). DOI: `10.1145/2049662.2049663`. URL: `http://doi.acm.org/10.1145/2049662.2049663`.

[23] M. Delorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, T. F. Knight Jr and A. DeHon. 'GraphStep: A system architecture for sparse-graph algorithms'. In: *Field-Programmable Custom Computing Machines, 14th Annual IEEE Symp. on.* 2006.

[24] S. Dolan. *Six degrees of Wikipedia*. 2008. URL: `http://mu.netsoc.ie/wiki/`.

[25] R. Dorrance, F. Ren and D. Marković. 'A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on FPGAs'. In: *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*. ACM. 2014, pp. 161–170.

[26] D. A. Drachman. 'Do we have brain to spare?' In: *Neurology* 64.12 (2005), pp. 2004–2005.

[27] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung and G. Stitt. 'A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication'. In: *Field-Programmable Custom Computing Machines, IEEE Int. Symp. on.* 2014.

[28] C. Gary C.T., P. Grigoras, P. Burovskiy and W. Luk. 'An Efficient Sparse Conjugate Gradient Solver Using a Benes Permutation Network'. In: *Field Prog. Logic and Applications, Int. Conf. on.* 2014.

[29] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis and N. Koziris. 'Performance evaluation of the sparse matrix-vector multiplication on modern architectures'. English. In: *The Journal of Supercomputing* 50.1 (2009). DOI: `10.1007/s11227-008-0251-8`. URL: `http://dx.doi.org/10.1007/s11227-008-0251-8`.

[30] D. Gregg, C. Mc Sweeney, C. McElroy, F. Connor, S. McGettrick, D. Moloney and D. Geraghty. 'FPGA based sparse matrix vector multiplication using commodity DRAM memory'. In: *Field Programmable Logic and Applications, International Conference on.* IEEE. 2007.

[31] R. J. Halstead and W. Najjar. 'Compiled Multithreaded Data Paths on FPGAs for Dynamic Workloads'. In: *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems.* 2013. URL: http://dl.acm.org/citation.cfm?id=2555729.2555732.

[32] M. A. Hassaan, M. Burtscher and K. Pingali. 'Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms'. In: *ACM SIGPLAN Notices.* Vol. 46. 8. 2011.

[33] S. Hong, T. Oguntebi and K. Olukotun. 'Efficient parallel graph exploration on multi-core CPU and GPU'. In: *Parallel Architectures and Compilation Techniques (PACT), 2011 Intl. Conf. on.* 2011.

[34] S. Jain-Mendon and R. Sass. 'A case study of streaming storage format for sparse matrices'. In: *Reconfigurable Computing and FPGAs (ReConFig), International Conference on.* IEEE. 2012.

[35] S. Jain-Mendon and R. Sass. 'A hardware–software co-design approach for implementing sparse matrix vector multiplication on FPGAs'. In: *Microprocessors and Microsystems* 38.8 (2014), pp. 873–888.

[36] N. P. Jouppi. 'Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers'. In: *Computer Architecture, Proc. of the Int. Symp. on.* 1990.

[37] D. Kempe, J. Kleinberg and É. Tardos. 'Maximizing the spread of influence through a social network'. In: *Proc. of the ninth ACM SIGKDD Int. Conf. on Knowledge discovery and data mining.* 2003.

[38] J. Kepner and J. Gilbert. *Graph algorithms in the language of linear algebra.* Vol. 22. SIAM, 2011.

[39] S. Kestur, J. D. Davis and E. S. Chung. 'Towards a Universal FPGA Matrix-Vector Multiplication Architecture'. In: *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on.* IEEE. 2012.

[40] K. Kourtis, V. Karakasis, G. Goumas and N. Koziris. 'CSX: an extended compression format for SpMV on shared memory systems'. In: *ACM SIGPLAN Notices.* Vol. 46. 8. ACM. 2011, pp. 247–256.

[41] C. E. Leiserson and T. B. Schardl. 'A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)'. In: *Proc. of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*. 2010.

[42] B. Liu, M. Wang, H. Foroosh, M. F. Tappen and M. Pensky. 'Sparse Convolutional Neural Networks'. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. (Boston, MA, USA). 2015, pp. 806–814. DOI: `10.1109/CVPR.2015.7298681`. URL: `http://dx.doi.org/10.1109/CVPR.2015.7298681`.

[43] E. Mastrostefano and M. Bernaschi. 'Efficient breadth first search on multi-GPU systems'. In: *Journal of Parallel and Distributed Computing* 73.9 (2013).

[44] K. Matam and V. Prasanna. 'Energy-efficient large-scale matrix multiplication on FPGAs'. In: *Reconfigurable Computing and FPGAs (ReConFig), 2013 International Conference on*. Dec. 2013. DOI: `10.1109/ReConFig.2013.6732284`.

[45] D. Merrill, M. Garland and A. Grimshaw. 'Scalable GPU graph traversal'. In: *ACM SIGPLAN Notices*. Vol. 47. 8. 2012.

[46] K. K. Nagar and J. D. Bakos. 'A Sparse Matrix Personality for the Convey HC-1'. In: *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*. IEEE. 2011, pp. 1–8.

[47] L. Oliker, X. Li, P. Husbands and R. Biswas. 'Effects of ordering strategies and programming paradigms on sparse matrix computations'. In: *Siam Review* 44.3 (2002), pp. 373–393.

[48] J. C. Pichel, F. F. Rivera, M. Fernandez and A. Rodriguez. 'Optimization of sparse matrix–vector multiplication using reordering techniques on GPUs'. In: *Microprocessors and Microsystems* 36.2 (2012), pp. 65–77.

[49] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray et al. 'A reconfigurable fabric for accelerating large-scale datacenter services'. In: *Comp. Arch. (ISCA), 2014 ACM/IEEE 41st Intl. Symp. on*. 2014.

[50] D. A. Reed and J. Dongarra. 'Exascale Computing and Big Data'. In: *Communications of the ACM* 58.7 (2015).

[51] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson and J. D. Owens. 'Memory Access Scheduling'. In: *Proceedings of the 27th Annual Int. Symp. on Computer Architecture*. 2000. DOI: 10.1145/339647.339668. URL: http://doi.acm.org/10.1145/339647.339668.

[52] P. Rosenfeld, E. Cooper-Balis and B. Jacob. 'DRAMSim2: A Cycle Accurate Memory System Simulator.' In: *Computer Architecture Letters* 10.1 (2011). URL: http://dblp.uni-trier.de/db/journals/cal/cal10.html#RosenfeldCJ11.

[53] A. Roy, I. Mihailovic and W. Zwaenepoel. 'X-Stream: Edge-centric graph processing using streaming partitions'. In: *Proc. of the Twenty-Fourth ACM Symp. on Operating Systems Principles*. 2013.

[54] S. Secchi, M. Ceriani, A. Tumeo, O. Villa, G. Palermo and L. Raffo. 'Exploring hardware support for scaling irregular applications on multi-node multi-core architectures'. In: *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th Intl. Conf. on*. 2013.

[55] B. Spinean, A. Geursen and G. Gaydadjiev. 'Improving DRAM performance and energy efficiency'. In: *Proc. IEEE Symposium on Embedded Systems for Real-Time Multimedia*. Tampere, Finland, Oct. 2012.

[56] S. Sun and J. Zambreno. 'A floating-point accumulator for FPGA-based high performance computing applications'. In: *Field-Programmable Technology, International Conference on*. Dec. 2009. DOI: 10.1109/FPT.2009.5377624.

[57] M. B. Taylor. 'Is dark silicon useful?: harnessing the four horsemen of the coming dark silicon apocalypse'. In: *Proceedings of the 49th Annual Design Automation Conference*. ACM. 2012.

[58] O. Temam and W. Jalby. 'Characterizing the behavior of sparse algorithms on caches'. In: *Proc. of the ACM/IEEE Conf. on Supercomputing*. 1992.

[59] S. Toledo. 'Improving the memory-system performance of sparse-matrix vector multiplication'. In: *IBM Journal of research and development* 41.6 (1997), pp. 711–725.

[60] K. Townsend and J. Zambreno. ‘Reduce, Reuse, Recycle (R 3): A design methodology for Sparse Matrix Vector Multiplication on reconfigurable platforms’. In: *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*. IEEE. 2013, pp. 185–191.

[61] Y. Umuroglu and M. Jahre. ‘A Vector Caching Scheme for Streaming FPGA SpMV Accelerators’. In: *Applied Reconfigurable Computing*. Vol. 9040. 2015.

[62] Y. Umuroglu and M. Jahre. ‘An Energy Efficient Column-Major Backend for FPGA SpMV Accelerators’. In: *Computer Design, IEEE Int. Conf. on*. 2014.

[63] Y. Umuroglu and M. Jahre. ‘An energy efficient column-major backend for FPGA SpMV accelerators’. In: *Computer Design (ICCD), 2014 32nd IEEE Intl. Conf. on*. 2014.

[64] Y. Umuroglu, D. Morrison and M. Jahre. ‘Hybrid Breadth-First Search on a Single-Chip FPGA-CPU Heterogeneous Platform’. In: *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*. 2015.

[65] R. W. Vuduc and H.-J. Moon. ‘Fast sparse matrix-vector multiplication by exploiting variable block structure’. In: *International Conference on High Performance Computing and Communications*. Springer. 2005, pp. 807–816.

[66] R. Vuduc, J. W. Demmel and K. A. Yelick. ‘OSKI: A library of automatically tuned sparse matrix kernels’. In: *Journal of Physics: Conference Series*. Vol. 16. 1. IOP Publishing. 2005.

[67] Q. Wang, W. Jiang, Y. Xia and V. Prasanna. ‘A message-passing multi-softcore architecture on FPGA for Breadth-first Search’. In: *Field-Programmable Technology (FPT), 2010 Intl. Conf. on*. 2010.

[68] J. Weinberg, M. O. McCracken, E. Strohmaier and A. Snavely. ‘Quantifying locality in the memory access patterns of hpc applications’. In: *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*. IEEE. 2005, pp. 50–50.

[69] J. Willcock and A. Lumsdaine. 'Accelerating sparse matrix computations via data compression'. In: *Proceedings of the 20th annual international conference on Supercomputing*. ACM. 2006, pp. 307–316.

[70] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick and J. Demmel. 'Optimization of sparse matrix–vector multiplication on emerging multicore platforms'. In: *Parallel Computing* 35.3 (2009).

[71] S. Williams, A. Waterman and D. Patterson. 'Roofline: an insightful visual performance model for multicore architectures'. In: *Communications of the ACM* 52.4 (2009), pp. 65–76.

[72] Xilinx. *Zynq-7000 All Programmable SoC Technical Reference*. Feb. 2015. URL: http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.

[73] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson and U. Catalyurek. 'A scalable distributed parallel breadth-first search algorithm on BlueGene/L'. In: *Supercomputing, 2005. Proc. of the ACM/IEEE SC 2005 Conf.* 2005.

[74] L. Zhang, Z. Fang, M. Parker, B. K. Mathew, L. Schaelicke, J. B. Carter, W. C. Hsieh and S. A. McKee. 'The Impulse memory controller'. In: *Computers, IEEE Transactions on* 50.11 (2001).

[75] Y. Zhang, Y. Shalabi, R. Jain, K. Nagar and J. Bakos. 'FPGA vs. GPU for sparse matrix vector multiply'. In: *Field-Programmable Technology, International Conference on.* Dec. 2009. DOI: 10.1109/FPT.2009.5377620.

[76] H. Zheng, E. Gorbatov, H. David, J. Lin, Z. Zhang and Z. Zhu. 'Mini-rank: Adaptive DRAM architecture for improving memory power efficiency'. In: *Annual IEEE/ACM Int. Symp. on Microarchitecture*. 2008.

[77] L. Zhuo, G. R. Morris and V. K. Prasanna. 'High-Performance Reduction Circuits Using Deeply Pipelined Operators on FPGAs.' In: *IEEE Trans. Parallel Distrib. Syst.* 18.10 (8th Nov. 2007). URL: http://dblp.uni-trier.de/db/journals/tpds/tpds18.html#ZhuoMP07.

Part C.

Deep Neural Networks

1. Introduction

1.1. Inference with Deep Neural Networks

Deep Neural Networks (DNNs) are machine learning algorithms that are able to extract information from complex raw data by using multiple *layers* of linear transformations and nonlinear functions, as illustrated in Figure 1.1. Supported by the availability of high-performance floating point hardware in the form of GPGPUs, huge volumes of data that can be used for training and advances in DNN training algorithms, DNNs are now able to out-perform the average human on certain cognitive computing tasks [47, 55]. Two distinct phases are involved in using a DNN: *training*, where a large, labeled data set is used to learn the parameters, and *inference*, where the learned parameters are kept constant and are used to e.g., recognize images as part of an application. This work focuses on the inference phase.

With the potential to enable a new generation of intelligent applications and services, DNNs have garnered a large amount of interest from both academia and industry in the recent years. Examples of recent successes with deep learning and DNNs include localizing and classifying objects in images, recognizing and synthesizing speech, playing video games, and generating textual descriptions

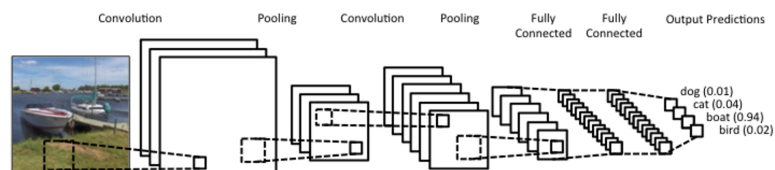


Figure 1.1.: A DNN for image classification (adapted from wildml.com).

Table 1.1.: Rough energy costs for various operations in 45 nm silicon operating at 0.9 V. Adapted from [19].

Operation	Energy Cost	Relative Energy Cost
8-bit integer add	0.03 pJ	1
8-bit integer multiply	0.2 pJ	10
32-bit floating point add	0.9 pJ	10
32-bit floating point multiply	3.7 pJ	100
64-bit access from 8 KB cache	20 pJ	1000
64-bit access from DRAM	1.3–3.6 nJ	100000

of images [32]. However, these impressive feats come at a high computational cost. A modern DNN may require billions of floating point multiply-accumulate operations performed on hundreds of megabytes of parameter data to perform a single inference. This introduces a significant computational challenge, especially when it is desired to deploy DNNs on *edge devices* with highly constrained power and energy budgets. These particular properties of DNN inference justify its selection as one of the chosen acceleration domains according to the criteria set in Part A Section 1.2.

1.2. Quantized Neural Networks

Combined with the fact that DNNs are highly parallelizable and the high market potential, the computational challenges brought by inference have spurred a great deal of interest in accelerators for *efficient deep learning* [51]. Efficient deep learning is concerned with not only the *cognitive performance* (i.e., classification accuracy) of DNNs, but also their *computational performance* (i.e., how much energy and time is required to perform inference). It is a relatively young and interdisciplinary field of research extending into computer architecture, machine learning, approximate computing, high-performance computing and distributed systems.

This thesis focuses on DNNs where the network is forced to operate with a quantized set of integer values, called *Quantized Neural Networks (QNNs)*. This

is a key technique for increasing the efficiency of DNN inference [51], and is studied in more detail in Section 2.2. The primary computational advantage of using QNNs comes from the quantized data types. No matter how efficient the architecture, any accelerator making use of floating point computation will be inherently restricted by the energy and area cost of floating point operations [19] and will rapidly hit the *memory wall* [58] due to the large number of memory bits occupied by floating point parameters. In contrast, quantized values require much fewer resources for computation, as well as fewer bits of memory storage which may allow for keeping the entire working set in on-chip memory. This provides an important computational advantage especially in terms of energy cost, as can be observed in Table 1.1.

2. Background

This chapter presents background concepts required to understand the research articles in this part of the thesis. How the topology is chosen, and how the parameters of the network are trained are outside the scope of this thesis. Instead, given a trained QNN, we will focus on which computations take place when performing inference with a feedforward DNN (without any recurrent connections) that are typically used for visual intelligence tasks.

2.1. The Anatomy of Inference Computations

In general terms, a DNN is a *universal approximator* for some function $y = f(x)$, where x is typically some high-dimensional input (e.g., an image) and y is the output generated by the network (e.g., the classification result of the image). Internally, the function f consists of a series of linear and nonlinear functions $f = g(h(j(\ldots)))$ called *layers*. The *architecture* or *topology* of the network describes how many such layers are present, their types, number of parameters and connections. The parameters (or *weights*) of the DNN are constants at inference time, and the inputs and outputs to each layer are referred to as *activations*.

The networks this thesis focuses on use a variant of the layer structure illustrated in Figure 2.1, and are referred to as *Convolutional Neural Networks (CNNs)*. The types of computation that take place in different layer types can be summarized as follows:

- *Convolutional layers* slide a small window over activations and compute a weighted sum of elements to capture the spatial correlations typically found in images. This operation can also be *lowered* to matrix-matrix multiplication [8].

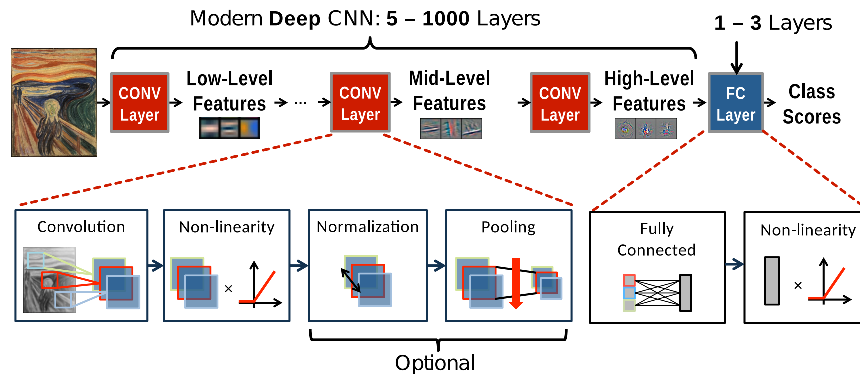


Figure 2.1.: Typical layer structure in a CNN. Adapted from [51].

- *Fully connected layers* multiply a weight matrix with the incoming activations from the previous layer.
- *Activation layers* apply an element-wise nonlinear function to the activations passing through them. Without nonlinearities, the entire neural network could have been collapsed to a single linear transform, which possesses less expressive power to model complex input-output relationships.
- *Normalization layers* perform statistical normalization of the activations that pass through them.
- *Pooling layers* downsample the activations that pass through them to provide a degree of scale invariance and lower the computational load.

2.2. Redundancy and Quantization

Although floating point numbers are a natural choice for handling the small updates that occur during neural network training, the resulting parameters can contain a lot of redundant information [16]. This can be exploited in different ways to reduce the computational cost of inference. Competitive levels of accuracy have been demonstrated using sparsification [34], singular value

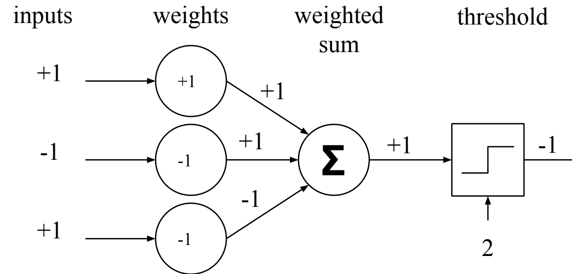


Figure 2.2.: A quantized neuron with binary inputs, weights and activations.

Table 2.1.: Accuracy on several image recognition benchmarks using QNNs.

Dataset	Topology	Float top-1 (top-5)	Quantized top-1 (top-5)	Quantization	Reference
MNIST	MLP	99%	99%	1-bit	[11]
SVHN	VGG-like	98%	99%	1-bit	[11]
CIFAR-10	VGG-like	93.2%	92.5%	2-bit	[7]
ImageNet	AlexNet	58.5% (81.5%)	52.7% (76.3%)	2-bit, 8-bit	[7]
ImageNet	GoogLeNet	71.4% (90.5%)	63.0% (84.9%)	2-bit, 8-bit	[7]
ImageNet	VGG-like	69.8% (89.3%)	64.1% (85.6%)	2-bit, 8-bit	[7]

decomposition [13], changing the architecture [20] or a combination of techniques [17, 23]. For example, Han et al. [17] showed that 92% of synapses in LeNet variants can be pruned without impacting classification accuracy. Unfortunately, this approach results in a sparse, irregular computation that is difficult to parallelize and execute efficiently, as was studied in Part B of this thesis.

Another approach for exploiting redundancy in neural networks is to use *quantization*. This reduces the parameter volume and computational cost of neural networks, while still keeping the computation dense and regular. Empirical results indicate that it is possible to directly quantize trained floating point networks down to 16 or 8 bits with minimal loss of accuracy [26, 44], but there are recent research results indicating that is possible to use far fewer bits as well. Sung et al. [50] study the effects of extreme quantization for both fully-connected and convolutional networks, and report that ternary-precision

networks with enough connections and quantization-aware training perform almost as well as full-precision networks. A number of recent works have demonstrated this in practice for a number of different image recognition benchmarks, which is summarized in Table 2.1. Research by Courbariaux et al. [11] and Rastegari et al. [45] proposed BNNs which use 1-bit weights and activations, as exemplified in Figure 2.2. These *Binarized Neural Networks (BNNs)* can offer accuracy comparable to floating point DNNs on smaller benchmarks. Although there is some accuracy degradation when using BNNs for more challenging benchmarks such as ImageNet, recent research on QNNs variants with higher bitwidth show some promise for closing this accuracy gap. For instance, Half-Wave Gaussian Quantization (HWGQ) by Zhou et al. [7] can reduce the top- 5 accuracy drop to less than 5% using the AlexNet topology with binary weights and two-bit activations.

2.3. Computational Properties

QNN inference mostly consists of dense matrix-matrix and matrix-vector multiplications, whose optimization and parallelization are well studied in literature [6, 25, 27, 56]. Based on prior work, the salient computational properties for QNN inference can be summarized as follows:

1. **High arithmetic intensity.** The key computation in QNN inference is matrix-matrix multiplication, which performs $O(N^3)$ operations on $O(N)$ data. This high arithmetic intensity lessens the impact of the memory wall and can enable high performance on a variety of architectures [56].

2. **Low-precision computations.** Data types used in QNN inference are multiply-add operations between few-bit (typically $\leq$ 4-bit) integers [61]. The bitwidths may differ from layer to layer and between weights and activations [42]. In contrast, 8-bit is the smallest natively supported precision in most typical CPU and GPGPU Instruction Set Architectures (ISAs).

3. **Multiple levels of regular-structured parallelism.** We can identify five levels of parallelism in QNN inference: bit-level, synapse-level, neuron-level and batch-level. This makes it amenable to a high degree of paral-

lelization, with different data reuse patterns that yield different energy efficiency characteristics [51].

4. **Possibilities for static analysis.** A trained QNN has a fixed topology, fixed data dependencies and fixed values for weights. This makes it possible to create customized accelerator architectures on the FPGA that are highly optimized for the particular network.

Paper C1

FINN: A Framework for Fast, Scalable Binarized Neural Network Inference

Yaman Umuroğlu, Nicholas Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre and Kees Vissers

Published in
Proceedings of the 2017 ACM/SIGDA International Symposium on
Field-Programmable Gate Arrays (FPGA)

C1. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference

Abstract. Research has shown that convolutional neural networks contain significant redundancy, and high classification accuracy can be obtained even when weights and activations are reduced from floating point to binary values. In this paper, we present FINN, a framework for building fast and flexible FPGA accelerators using a flexible heterogeneous streaming architecture. By utilizing a novel set of optimizations that enable efficient mapping of binarized neural networks to hardware, we implement fully connected, convolutional and pooling layers, with per-layer compute resources being tailored to user-provided throughput requirements. On a ZC706 embedded FPGA platform drawing less than 25 W total system power, we demonstrate up to 12.3 million image classifications per second with 0.31 µs latency on the MNIST dataset with 95.8% accuracy, and 21906 image classifications per second with 283 µs latency on the CIFAR-10 and SVHN datasets with respectively 80.1% and 94.9% accuracy. To the best of our knowledge, ours are the fastest classification rates reported to date on these benchmarks.

C1.1. Introduction

Convolutional Neural Networks (CNNs) have dramatically improved in recent years, their performance now exceeding that of other visual recognition algorithms [31], and even surpassing human accuracy on certain problems [47, 55]. They are likely to play an important role in enabling ubiquitous machine vision and intelligence on all kinds of devices, but a significant computational

challenge remains. Modern CNNs may contain millions of floating-point parameters and require billions of floating-point operations to recognize a single image. Furthermore, these requirements tend to increase as researchers explore deeper networks. For instance, AlexNet [31] (the winning entry for ImageNet Large Scale Visual Recognition Competition (ILSVRC) [46] in 2012) required 244 MB of parameters and 1.4 billion floating point operations (GFLOP) per image, while VGG-16 [48] from ILSVRC 2014 required 552 MB of parameters and 30.8 GFLOP per image.

While the vast majority of CNNs implementations use floating point parameters, a growing body of research demonstrates this approach incorporates significant redundancy. Recently, it has been shown [11, 29, 45, 50, 61] that neural networks can classify accurately using one- or two-bit quantization for weights and activations. Such a combination of low-precision arithmetic and small memory footprint presents a unique opportunity for fast and energy-efficient image classification using Field Programmable Gate Arrays (FPGAs). FPGAs have *much* higher theoretical peak performance for binary operations compared to floating point, while the small memory footprint *removes* the off-chip memory bottleneck by keeping parameters on-chip, even for large networks. Binarized Neural Networks (BNNs), proposed by Courbariaux et al. [11], are particularly appealing since they can be implemented almost entirely with binary operations, with the potential to attain performance in the teraoperations per second (TOPS) range on FPGAs.

In this work, we propose FINN, a framework for building scalable and fast BNN inference accelerators on FPGAs. FINN-generated accelerators can perform millions of classifications per second with sub-microsecond latency, thereby making them ideal for supporting real-time embedded applications such as augmented reality, autonomous driving and robotics. Compute resources can be scaled to meet a given classification rate requirement. We demonstrate FINN's capabilities with a series of prototypes for classifying the MNIST, SVHN and CIFAR-10 benchmark datasets. Our classification rate results surpass the best previously published results by over 48× for MNIST, 2.2× for CIFAR-10 and 8× for SVHN. To the best of our knowledge, this is the fastest reported neural network inference implementation on these datasets. The novel contributions are:

- Quantification of peak performance for BNNs on

FPGAs using a roofline model.

- A set of novel optimizations for mapping BNNs onto FPGA more efficiently.
- A BNN architecture and accelerator construction tool, permitting customization of throughput.
- A range of prototypes that demonstrate the potential of BNNs on off-the-shelf FPGA platforms.

The rest of this paper is organized as follows: Section 2 provides background on CNNs, BNNs, and their hardware implementations. Section 3 discusses BNNs accuracy and peak performance on FPGAs. Section 4 describes FINN's architecture and optimizations. Section 5 presents the experimental evaluation, and Section 6 concludes the paper.

C1.2. Background

This work is focused on *supervised* learning, in which the goal is to find a function, $g(\mathbf{x}_i)$, which approximates a mapping $\mathbf{x}_i \rightarrow y_i \ \forall \ i$, where $\{\mathbf{x}_i, y_i\}$ is an input/output pair known as a training example. Furthermore, only the *inference* problem is studied, the parameters, w, being assumed to have been learned offline.

C1.2.1. Convolutional Neural Networks

A *multilayer perceptron* is a type of Artificial Neural Network (ANN) which has its neurons arranged in multiple layers, with neurons taking the output of all neurons of the previous layer as inputs. Mathematically, the output, $a_{l,n}$, for the nth neuron in the lth layer of a fully connected network is calculated as follows:

$$a_{l,n} = f_{act}(\sum_{s=0}^{S_l} w_{l,n,s} a_{l-1,s} + b_{l,n}) \ , \qquad \text{(C1.1)}$$

where $w_{l,n,s}$ is weight of the sth synapse connected to the input of the nth neuron in the lth layer, $b_{l,n}$ is a bias term, f_{act} is the activation function, and S_l

is the number of synapses connected to each neuron in the lth layer. Popular activation functions include: the hyperbolic tangent function, $f_{act}(a) = tanh(a)$; and the rectified linear unit (ReLU), $f_{act}(a) = max(0, a)$.

CNNs [33] are a variant of multilayer perceptrons, in which a layer only receives inputs from a small *receptive field* of the previous layer. This approach greatly reduces the number of parameters involved and allows local features (e.g., edges, corners) to be found [33]. A basic 2D convolutional layer in a neural network is similar to a fully connected layer except that: a) each neuron receives an image as inputs and produces an image as its output (instead of a scalar); b) each synapse learns a small array of weights which is the size of the convolutional window; and c) each pixel in the output image is created by the sum of the convolutions between all synapse weights and the corresponding images. The output of the lth convolutional layer, which takes as input S_l images of dimension $R_l \times C_l$, the pixel, $p_{l,n,r,c}$, at location (r, c) of the nth output image is calculated as follows:

$$p_{l,n,r,c} = f_{act}(\sum_{s=0}^{S_l} \sum_{j=0}^{J_l} \sum_{k=0}^{K_l} w_{l,n,s,j,k} p_{l-1,n,r+j,c+k}) \ , \tag{C1.2}$$

where $J_l \times K_l$ are the dimensions of the convolution window. As discussed in Section C1.4, a 2D convolutional layer can be reduced to a matrix multiply followed by an elementwise activation function. CNN topologies are composed from a few common primitives: convolutional layers, *pooling* layers and fully connected layers.

Pooling layers can be considered as simple downsamplers of 2D images. A basic max pooling layer divides an image into small sub-tiles of a given window size and then replaces each sub-tile with its largest element. An average pooling layer is similar but uses the average function instead of max.

C1.2.2. Binary Neural Networks

Although floating point numbers are a natural choice for handling the small updates that occur during neural network training, the resulting parameters can contain a lot of redundant information [16]. One of several possible dimensions possessing redundancy is precision [50]. An extreme case are BNNs in which

some or all the arithmetic involved in computing the outputs are constrained to single-bit values. We consider three aspects of binarization for neural network layers: binary input activations, binary synapse weights and binary output activations. If all three components are binary, we refer to this as *full binarization*, and the cases with one or two components as *partial binarization*.

Kim and Smaragdis [29] consider full binarization with a predetermined portion of the synapses having zero weight, and all other synapses with a weight of one. They report 98.7% accuracy with fully-connected networks on the MNIST dataset, and observe that only XNOR and bitcount operations are necessary for computing with such neural networks. XNOR-Net by Rastegari et al. [45] applies convolutional BNNs on the ImageNet dataset with topologies inspired by AlexNet, ResNet and GoogLeNet, reporting top-1 accuracies of up to 51.2% for full binarization and 65.5% for partial binarization. DoReFa-Net by Zhou et al. [61] explores reduced precision during the forward pass as well as the backward pass, and note that this opens interesting possibilities for training neural networks on FPGAs. Their results includes configurations with partial and full binarization on the SVHN and ImageNet datasets, including best-case ImageNet top-1 accuracies of 43% for full and 53% for partial binarization.

Finally, the work by Courbariaux et al. [11] describes how to train fully connected and convolutional networks with full binarization and batch normalization layers, reporting competitive accuracy on the MNIST, SVHN and CIFAR-10 datasets. Training for this work was performed using their open source implementation. We use the acronym CNN to refer to conventional or non-binarized neural networks for brevity throughout the rest of this paper.

C1.2.3. Neural Networks in Hardware

A great deal of prior work on mapping neural networks to hardware exist both for FPGAs and as ASICs. We refer the reader to the work by Misra and Saha [35] for a comprehensive survey. We cover a recent and representative set of works here, roughly dividing them into four categories based on their basic architecture: 1) a single processing engine [3, 10, 41, 60], usually in the form of a systolic array, which processes each layer sequentially; 2) a streaming architecture [2, 54], consisting of one processing engine per network layer; 3) a vector processor [15] with instructions specific to accelerating the primitives operations

of convolutions; and 4) a neurosynaptic processor [14], which implements many digital neurons and their interconnecting weights.

Systolic arrays: Zhang et al. [60] describes a single processing engine style architecture, using theoretical roofline models tool to design accelerators optimized for the execution of each layer. Ovtcharov et al. [41] implement a similar style architecture, but achieved a 3× speedup over Zhang et al. [60]. Eyeriss by Chen et al. [10] use 16-bit fixed point rather than floating point, and combine several different data reuse strategies. Each 2D convolution is mapped to 1D convolutions across multiple processing engines, allowing for completely regular access patterns for each processing element. The authors report that their data reuse provides 2.5× better energy efficiency over other methods. YodaNN by Andri et al. [3] have a similar design as Zhang et al. [60] but explore binary weights for fixed sized windows.

Streaming architectures: Venieris and Bouganis [54] proposed a synchronous dataflow (SDF) model for mapping CNNs to FPGAs, which is a similar approach to ours. The main difference is that our design is optimized for BNNs while their design targets conventional CNNs. Their designs achieve up to 1.62× the performance density of hand tuned designs. Alemdar et al. [2] implement fully-connected ternary-weight neural networks with streaming and report up to 255K frames per second on the MNIST dataset, but concentrate on the training aspect for those networks.

Vector processors: Farabet et al. [15] describe a programmable ConvNet Processor (CNP), which is a RISC vector processor with specific macro-instructions for CNNs including 2D convolutions, 2D spatial pooling, dot product and an elementwise non-linear mapping function. The authors also created a tool to compile a high level network description into host code which is used to call the CNP.

Neurosynaptic processors: TrueNorth [14] is a low power, parallel ASIC with 4096 neurosynaptic cores, each implementing 256 binary inputs, 256 neurons and a 256 × 256 array of synapses. An internal spiking router can connect any input on any core to any neuron on any core, allowing many network topologies to be implemented on fixed hardware.

The authors are not aware of any publication that demonstrates end-to-end mapping of BNNs onto FPGAs. In comparison to prior art, the binary network

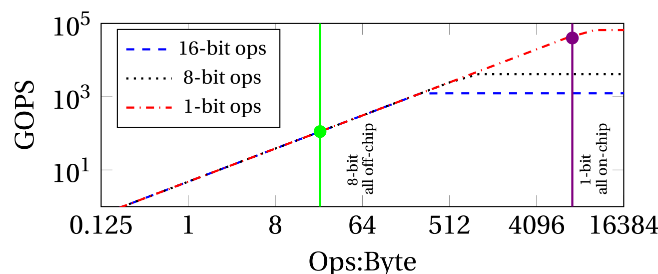


Figure C1.1.: Roofline model for a ZU19EG.

inference engine can significantly increase classification rates, while reducing power consumption and minimizing latency. This currently comes at the cost of a small drop in accuracy for larger networks, however we believe a) there are use cases that do not require the highest level of accuracy, or can be solved with smaller networks (such as classification of playing cards or handwritten digits [33]) and b) that accuracy can be improved by increasing network sizes [50], an ongoing topic in machine learning research.

C1.3. BNN Performance and Accuracy

C1.3.1. Estimating Performance Using Rooflines

To estimate and compare BNN performance with fixed-point CNN, we use a *roofline model* [57] which considers memory bandwidth, peak computational performance and arithmetic intensity (the number of mathematical operations performed for each byte of off-chip memory read or written). The intersection of the roofline curve with a vertical line for a particular arithmetic intensity, gives the theoretical peak performance point, which is either *compute-bound* or *memory-bound*. In particular, we consider the binarized [45, 61] and 8-bit fixed-point [49] implementations of the popular AlexNet [31], both of which require 1.4 billion operations (GOPS) to classify one image.

Using the methodology described in [36], we develop a roofline model for

a Xilinx Zynq UltraScale+ ZU19EG FPGA[1]. The resulting roofline model is depicted in Figure C1.1. We first observe that the FPGA's compute-bound performance is 66 TOPS for binary operations, which is about 16× higher compared to 8-bit and 53× higher compared to 16-bit fixed point operations. However, reaching the compute-bound peak is only possible if the application is not memory-bound. The compact model size of BNNs provides another key benefit. Since the binarized AlexNet requires only 7.4 MB of parameters (compared with 50 MB for 8-bits), the entire neural network model can be kept in on-chip memory. The arithmetic intensities for the binarized and 8-bit fixed point AlexNet variants are shown with vertical lines. Thus, the BNN is almost able to reach the computational peak, while the peak performance of the fixed-point CNN is bound by the memory bandwidth. Based on these observations, with a design that reaches 75% of the peak, we estimate a throughput of $0.75 \cdot \frac{66\text{ TOPS}}{1.4\text{ GOPS}} \approx 35000$ images per second.

Using the same model, it should be possible to extend the comparison to CPUs and GPUs, but little data is available on peak binary synaptic operation performance since BNNs are relatively new. For instance, [11] mentions 6 cycles per 32 synapses (64 binary operations) on recent NVIDIA GPUs, which would yield a computational peak of about 26 TOPS on a Tesla K40 with 2880 cores running at 875 MHz, and 16666 images per second for binarized AlexNet.

C1.3.2. Accuracy–Computation Tradeoffs

A tradeoff between network size, precision and accuracy exists [50] so if one would like to achieve a certain classification accuracy for a particular problem, which approach leads to the most efficient solution? 1) A regular ANN with floating point precision? 2) A larger network, but a BNN? To gain more insight into this issue, we conducted a set of experiments on the MNIST dataset that compare accuracy of floating point and binary precision for the same topology. The binary networks are obtained via replacing regular layers by their binary equivalents, as described by Courbariaux et al. [11]. We also binarize the input images for the BNN as our experiments show that input binarization works well for MNIST. Since the space of possible network topologies that can be

[1] We assume 4.8 GB/s off-chip memory bandwidth, 350 MHz clock and the following operation cost function: 2.5 LUTs for 1-bit, 40 LUTs for 8-bit, 8 LUTs and 0.5 DSPs for 16-bit.

Table C1.1.: Accuracy results - BNN vs floating point NN.

Neurons/layer	Binary Err. (%)	Float Err. (%)	# Params	Ops/frame
128	6.58	2.70	134,794	268,800
256	4.17	1.78	335,114	668,672
512	2.31	1.25	932,362	1,861,632
1024	1.60	1.13	2,913,290	5,820,416
2048	1.32	0.97	10,020,874	20,029,440
4096	1.17	0.91	36,818,954	73,613,312

trained is infinite, we adopted the approach in [50] to simplify the problem. We fix the network topology to a 3 hidden layer, fully connected network while scaling the number of neurons in each layer, and plot the resulting accuracy in Table C1.1 along with the number of parameters and operations per frame. A few trends are apparent for this problem and network configuration space: 1) similar to what was found in by Sung et al. [50], as the network size increases, the difference in accuracy between low precision networks and floating point networks decreases; and 2) in order to achieve the same level of accuracy as floating point networks, BNNs require 2–11× more parameters and operations. Note that we show the accuracy for networks trained using 32-bit floating point numbers, but it is likely that this could be reduced to 8-bit fixed point without a significant change in accuracy [23]. Our BNN performance estimates from Section C1.3.1 suggest a 16× speedup for BNN over 8-bit fixed point, which is greater than the 2–11× increase in parameter and operation size. Thus, we expect that BNNs with comparable accuracy will be faster than fixed-point networks, even though they may require more parameters and operations.

C1.4. BNNs on Reconfigurable Logic

C1.4.1. Architecture

We adopted a *heterogeneous streaming* architecture as shown in Figure C1.2 for this work. We build a custom architecture for a given topology rather than

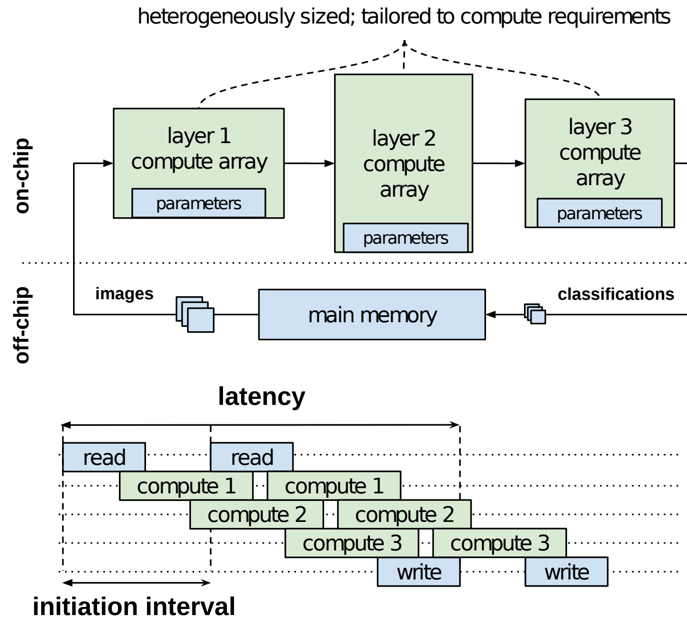


Figure C1.2.: Heterogeneous streaming architecture and schedule.

scheduling a operations on top of a fixed architecture. Separate compute engines are dedicated to each layer, which communicate via on-chip data streams. Each engine starts to compute as soon as the previous engine starts to produce output. Additionally, owing to the compact model size of BNNs, all neural network parameters are kept in on-chip memory. This avoids most accesses to off-chip memory, minimizes the *latency* (the time to finish classifying one image) by overlapping computation and communication, and minimizes the *initiation interval*: a new image can enter the accelerator as soon as the first compute array is finished with the previous image. The separate mapping of layers to compute arrays also enables heterogeneity. By tailoring compute arrays separately for each layer's requirements, we can avoid the "one-size-fits-all" inefficiencies and reap more of the benefits of reconfigurable computing. This requires a different bitfile when the neural network topology is changed but we consider this an acceptable cost for the performance gains obtained.

A BNN accelerator may have various constraints imposed upon it depending on the use case. User-imposed constraints include the choice of FPGA

and platform, desired classification throughput in frames per second (FPS) and clock frequency. Simultaneously, the BNN topology constrains how the compute resources must be allocated to obtain an efficient heterogeneous streaming architecture. FINN offers parameterizable building blocks and a way of controlling the classification throughput, as described in Sections C1.4.3 and C1.4.4. To achieve portability, we chose a commercial high level synthesis tool, Vivado High-Level Synthesis (HLS), for the implementation. The tool enables faster development cycles via high-level abstractions, and provides automated pipelining to meet the clock frequency target.

C1.4.2. BNN-specific Operator Optimizations

BNNs have several properties that enable a more efficient mapping to FPGAs without affecting the network accuracy, which we describe in the following subsections. We assume that the methodology described in [11] is used for training all BNNs in this paper, where all BNN layers have the following properties (unless otherwise stated):

- Using 1-bit values for all input activations, weights and output activations (full binarization), where an unset bit represents -1 and a set bit represents +1.
- Batch normalization prior to the activation function.
- Using the following activation function:
  $\text{Sign}(x) = \{+1 \text{ if } x \geq 0, -1 \text{ if } x < 0\}$

Popcount for Accumulation

The regular and value-constrained nature of BNN computations enable computing binary dot products with fewer hardware resources. Let Y be the number of input synapses (or *fan-in*) for a given neuron, with the number of +1-valued synapse inputs denoted as Y_1 and -1-valued synapses as Y_0. As there are only two possible values (-1 and +1) for any synapse input, $Y = Y_0 + Y_1$. Therefore, by counting the number of synapses for only one value, it is possible to infer the summed response for the entire neuron.

The practical consequence for hardware is that the summation of a binary dot product can be implemented by a *popcount* operation that counts the number of set bits instead of accumulation with signed arithmetic. Our experiments with Vivado HLS indicate that popcount-accumulate requires approximately half the number of LUT and FF resources to implement compared to signed-accumulate. For instance, with a target $F_{\text{clk}} = 200$ MHz, a 128-bit popcount-accumulate requires 376 LUTs and 29 FFs, while a 128-bit bipolar-accumulate requires 759 LUTs and 84 FFs.

Batchnorm-activation as Threshold

All BNN layers use batch normalization [24] on convolutional or fully connected layer outputs, then apply the sign function to determine the output activation. We show how the same output can be computed via thresholding.

Let a_k be the dot product (pre-activation) output of neuron k, and $\Theta_k = (\gamma_k, \mu_k, i_k, B_k)$ be the batch normalization parameters learned during training for this neuron. The output a_k^b is computed as $a_k^b = \text{Sign}(\text{BatchNorm}(a_k, \Theta_k))$, with $\text{BatchNorm}(a_k, \Theta_k) = \gamma_k \cdot (a_k - \mu_k) \cdot i_k + B_k$. Figure C1.3 shows the dot product input vs output activation for three example neurons. Depending on parameter values, the plot may be shifted towards the left or right, or be flipped horizontally, but a threshold τ_k for a change in the output activation is always present. Solving $\text{BatchNorm}(\tau_k, \Theta_k) = 0$ we can deduce that $\tau_k = \mu_k - (B_k / (\gamma_k \cdot i_k))$.

To make the thresholds compatible with the positive-only operations in Section C1.4.2), the computed threshold is averaged with the neuron fan-in S to obtain $\tau_k^+ = (\tau_k + S)/2$. Observing how neuron C activates with an opposite sign threshold to neurons A and B in Figure C1.3, all neurons can be made to activate using a greater-than threshold by flipping the signs of a neuron's weights if $\gamma_k \cdot i_k < 0$.

Using these techniques, we can compute the output activation using an unsigned comparison and avoid computing the batch normalized value altogether during inference. τ_k^+ itself is fixed for a trained network and can be computed from the batchnorm parameters at compile time. Synthesis reports from Vivado

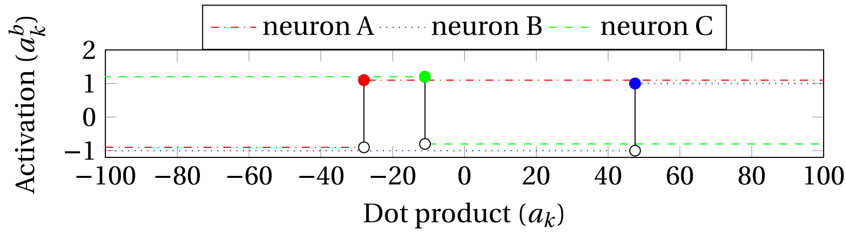


Figure C1.3.: Three examples of binary neuron activations with batch normalization. A slight vertical offset is added for clarity.

HLS for 16-bit dot product output values indicate that regular batchnorm-and-sign activation requires 2 DSPs, 55 FFs and 40 LUTs, whereas the threshold activation we describe here only requires 6 LUTs.

Boolean OR for Max-pooling

The networks described in [11] perform pooling prior to activations, i.e., pooling is performed on non-binarized numbers, which are then batch normalized and fed into the activation function. We show that the same layer outputs can be derived by max pooling *after* the activations without having to re-train the network. Let $a_1, a_2, \ldots a_Y$ be the positive dot product outputs that will be processed by max-pooling. In accordance with Section C1.4.2, the output would be computed as $a^b = (\text{Max}(a_1, a_2, \ldots a_Y) > \tau^+)$. Due to the distributivity of Max, the output will be $true$ if *any* of $a_1, a_2, \ldots a_S$ are greater than τ^+. Therefore, the same result can be computed as $a^b = (a_1 > \tau^+) \vee (a_2 > \tau^+) \ldots \vee (a_Y > \tau^+)$. As the threshold comparisons are already computed for the activations, max-pooling can be effectively implemented with the Boolean OR-operator. We note that similar principles apply for min-pooling (as Boolean AND) and average-pooling (as Boolean majority function) as well.

C1.4.3. FINN Design Flow and Hardware Library

Figure C1.4 illustrates the design flow for converting a trained BNN into an FPGA accelerator. The user supplies a FPS target alongside a Theano-trained BNN to the FINN synthesizer. The synthesizer first determines the folding

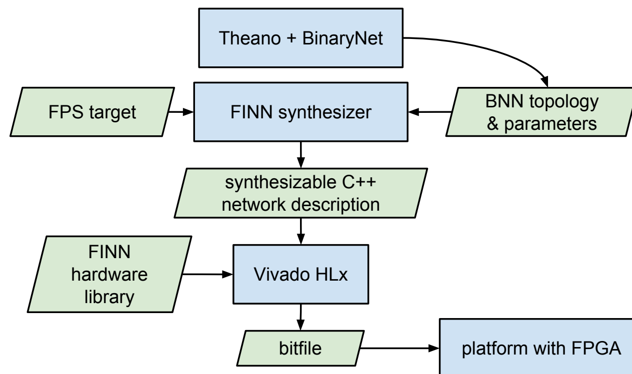


Figure C1.4.: Generating an FPGA accelerator from a trained BNN.

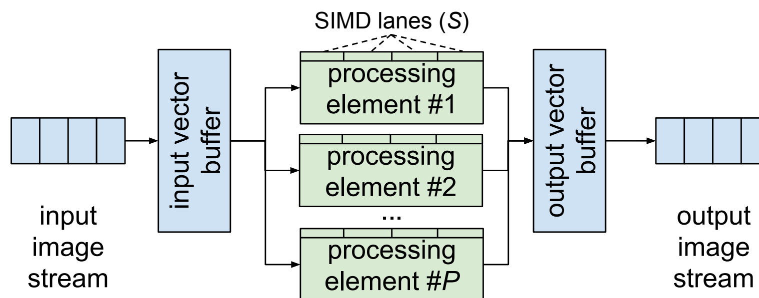


Figure C1.5.: Overview of the MVTU.

parameters (Section C1.4.4) to meet the FPS target and applies the optimizations from Section C1.4.2, then produces a synthesizable C++ description of a heterogeneous streaming architecture. The architecture is composed of building blocks from the FINN hardware library described in the following subsections.

The Matrix–Vector–Threshold Unit

The Matrix–Vector–Threshold Unit (MVTU) forms the computational core for our accelerator designs. The vast majority of compute operations in a BNN can be expressed as matrix–vector operations followed by thresholding. For

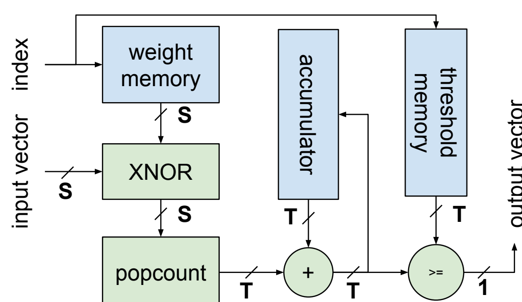


Figure C1.6.: MVTU PE datapath. **Bold** indicates bitwidth.

instance, the pre-activation output $\mathbf{a_N}$ of the fully connected neural network layer at index N is given by matrix-vector product $\mathbf{a_N} = \mathbf{A} \cdot \mathbf{a^b_{N-1}}$ where $\mathbf{A}$ is the synaptic weight matrix and $\mathbf{a^b_{N-1}}$ are the activations from the previous layer. The post-activation output can then be computed by $\mathbf{a^b_N} = \mathbf{a^N} > \tau^+_{\mathbf{N}}$, where the thresholds $\tau^+_{\mathbf{N}}$ are determined as described in Section C1.4.2. Convolutions can also be implemented as matrix–vector products, as will be described in Section C1.4.3. As such, the MVTU implements fully-connected layers as a standalone component, and is also used as part of the convolutional layers.

The overall organization of the MVTU is shown in Figure C1.5. Internally, the MVTU consists of an input and output buffer, and an array of Processing Elements (PEs) each with a number of SIMD lanes. The number of PEs (P) and SIMD lanes (S) are configurable to control the throughput as discussed in Section C1.4.4. The synapse weight matrix to be used is kept in On-Chip Memory (OCM) distributed between PEs, and the input images stream through the MVTU as each one is multiplied with the matrix. Each PE receives exactly the same control signals and input vector data, but multiply-accumulates the input with a different part of the matrix. In terms of the taxonomy described in [10], this architecture is both *weight stationary* (since each weight remains local to the PE) and *output stationary* (since each popcount computation remains local to the PE).

Figure C1.6 shows the datapath of an MVTU PE. It computes the dot product between the input vector and a row of the synaptic weight matrix and compares the result to a threshold, producing a single-bit output. The dot product computation itself consists of an XNOR of the vectors, after which the number of set

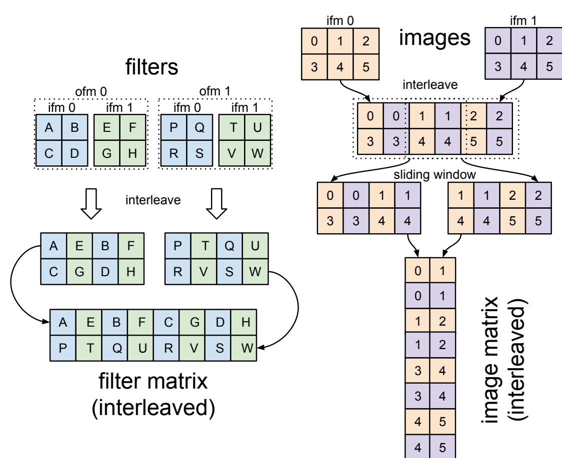
bits in the result is counted (see Section C1.4.2) and added to the accumulator register. Once the entire dot product is accumulated, it is thresholded. The accumulator, adder and threshold memory bitwidth can be scaled down to $T = 1 + \log_2(Y)$ for additional resource savings.

Finally, it is worth pointing out that the MVTU architectural template can also support partial binarization for non-binarized outputs and inputs. Removing the thresolding stage provides non-binarized outputs, while using regular multiply-add instead of XNOR-popcount can handle non-binarized inputs These features are used in the first and last layers of networks that process non-binary input images or do not output a one-hot classification vector.

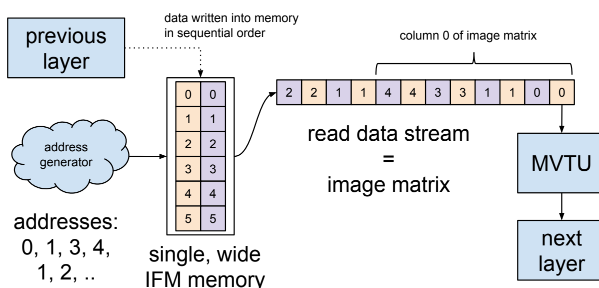
Convolution: The Sliding Window Unit

Convolutions can be *lowered* to matrix-matrix multiplications [8], which is the approach followed in this work. The weights from the convolution filters are packed into a *filter matrix*, while a sliding window is moved across input images to form an *image matrix*. These matrices are then multiplied to generate the output images.

The convolutional layer consists of a Sliding Window Unit (SWU), which generates the image matrix from incoming feature maps, and a MVTU that actually computes the matrix–matrix product using a different column vector from the image matrix each time. In order to better cater for the SIMD parallelism of the MVTU and minimize buffering requirements, we *interleave* the feature maps such that each pixel contains all the Input Feature Map (IFM) channel data for that position, as illustrated in Figure C1.7a. Since the dot product to compute a Output Feature Map (OFM) pixel includes all IFMs pixels at a certain sliding window location, those IFM pixels can be processed in any order owing to the commutative property of addition. Note that interleaving the filter matrix has no additional cost since it is done offline, and interleaving the input image can be done on-the-fly in the FPGA. Storing the pixels in this fashion allows us to implement the SWU with a single wide OCM instead of multiple narrow OCMs, and also enables the output of the MVTU to be directly fed to the next layer without any transposition. As illustrated in Figure C1.7b, the incoming IFM data is simply stored at sequential addresses in a buffer, then the memory



(a) Lowering with interleaved channels.



(b) SWU operation.

Figure C1.7.: Convolution using interleaved channels.

locations corresponding to each sliding window are read out to produce the image matrix.

Although not required by any of the networks described in this work, the SWU also pads the images if necessary. One interesting observation is that with the bipolar number representation used in this work, there is no number corresponding to zero. Therefore, in order to maintain a true binary datapath for activations, images must be padded with our representation or either a 1 or a -1. Future work will look into what impact this has on the accuracy of trained networks, but early experiments suggest that there is very little difference in accuracy, with respect to [11].

The Pooling Unit

The Pooling Unit (PU) implements max-pooling as described in Section C1.4.2. To implement $k \times k$ max-pooling on a $D_H \times D_W$ binary image of C channels, the PU contains $C \cdot k$ line buffers of D_W bits each. As with the rest of our component library, the PU operates in a streaming fashion. The input image is gradually streamed into the line buffers. When at least k rows of the image have arrived, each k consecutive bits of the line buffer are OR'ed together to produce horizontal subsampling for each channel. These are then OR'ed together with the other line buffers to produce vertical subsampling, the results are streamed out, and the oldest line buffers are refilled with the next row of pixels.

C1.4.4. Folding

In terms of the MVTU description given in Section C1.4.3, each PE corresponds to a *hardware neuron*, while each SIMD lane acts as a *hardware synapse*. If we were to dimension each MVTU in a network with a number of hardware neurons and synapses equal to the number of neurons and synapses in a BNN layer, this would result in a fully parallel neural network that could classify images at the clock rate. However, the amount of hardware resources on an FPGA is limited, and it is necessary to time-multiplex (or *fold*) the BNN onto fewer hardware synapses and neurons. We now describe how the folding is performed subject to user constraints.

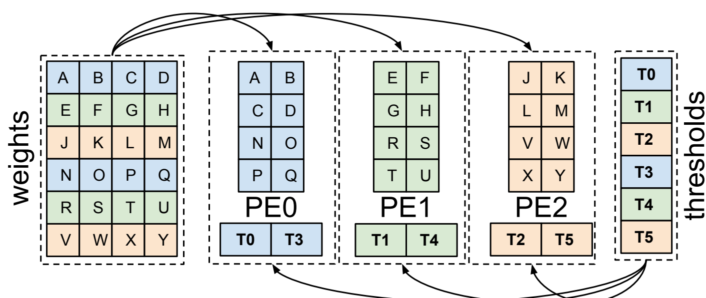


Figure C1.8.: Neuron and synapse folding for MVTU.

The work by Venieris et al. [54] describes a method for folding neural networks expressed as streaming dataflow graphs, with focus on formalizing the folding and design space exploration. In this work, we consider a simpler variant that only controls the folding of matrix–vector products to achieve a given FPS requirement set by the user, and focus on *how* the folding is implemented in terms of the workload mapping. As almost all computations in BNNs are expressed as matrix–vector multiplications, implementing folding for matrix–vector multiplication already enables a great degree of control over the system throughput. Folding directly affects the resource and power consumption of the final system as well, which we explore in Section C1.5.

Folding Matrix–Vector Products

Folding matrix–vector products is achieved by controlling two parameters of the MVTU: P the number of PEs, and S the number of SIMD lanes per PE. These determine how the matrix is partitioned between the PEs. A P-high, S-wide tile of the matrix is processed at a time, with each row in the tile mapped to a different PE, and each column to a different SIMD lane. For a $X \times Y$ matrix, we refer to $F^n = X/P$ as the *neuron fold* and $F^s = Y/S$ as the *synapse fold*. The *total fold* F is then obtained as $F = F^n \cdot F^s$, which is also the number of cycles required to complete one matrix–vector multiply. Note that F^n and F^s should be integers to avoid padding the weight matrix. As an example, Figure C1.8 shows how a 6×4 weight matrix is partitioned between three PEs with two SIMD lanes each. Here, each matrix-vector multiply will take $F^n \cdot F^s = (6/3) \cdot (4/2) = 4$

cycles.

The same principle applies for convolutional layers, but these always have an inherent amount of folding due to our current matrix–matrix product as multiple matrix–vector products implementation. For convolutional layers, the total fold is $F = F^m \cdot F^n \cdot F^s$, where F^m is a network-dependent constant due to multiple matrix-vector products, and is equal to the number of output pixels from the convolution.

Determining F^n and F^s

Avoiding the "one-size-fits-all" inefficiencies requires tailoring each MVTU's compute resources to layer requirements. The guiding principle here is *rate-balancing* the heterogeneous streaming architecture: the slowest layer (with $II_{\max}$) will determine the overall throughput, so each layer should use a roughly equal number of cycles to process one image. As this is a streaming system, the classification throughput FPS will be approximately $\frac{F_{\text{clk}}}{II_{\max}}$, where F_{clk} is the clock frequency. For a fully-connected layer, the total fold F is equal to the initiation interval (II). Therefore, balancing a fully-connected BNN can be achieved by using F^n and F^s such that $F^n \cdot F^s = \frac{F_{\text{clk}}}{\text{FPS}}$ for each layer. Depending on the BNN and the FPS requirements, the number of memory channels or sliding window generation may constitute bottlenecks. For such cases, we match the throughput of all other layers to the bottleneck in order not to waste resources.

C1.5. Evaluation

C1.5.1. Experimental Setup

To evaluate FINN, we created a number of prototypes that accelerate BNNs inference on the MNIST [33] (28×28 handwritten digits), CIFAR-10 [30] (32×32 color images in 10 categories) and cropped SVHN [37] (32×32 images of Street View House Numbers) datasets. Each prototype combines a BNN topology with a different use case scenario. We consider three different BNN topologies for classifying the datasets as follows:

- **SFC** and **LFC** are three-layer fully connected network topologies for classifying the MNIST dataset, with different numbers of neurons to demonstrate accuracy-computation tradeoffs (Section C1.3.2). SFC contains 256 neurons per layer and achieves 95.83% accuracy, while LFC has 1024 neurons per layer and achieves 98.4% accuracy. These networks accept 28x28 binary images and output a 10-bit one-hot vector indicating the digit.

- **CNV** is a convolutional network topology inspired by BinaryNet [11] and VGG-16 [48]. It contains a succession of (3x3 convolution, 3x3 convolution, 2x2 maxpool) layers repeated three times with 64-128-256 channels, followed by two fully connected layers of 512 neurons each. We use this topology for classifying both the CIFAR-10 (with 80.1% accuracy) and SVHN (with 94.9% accuracy) datasets, with different weights and thresholds. Note that the inputs to the first layer and the outputs from the last layer are not binarized; CNV accepts 32x32 images with 24 bits/pixel, and returns a 10-element vector of 16-bit values as the result.

To further demonstrate the flexibility of the framework, we consider two usage scenarios for each BNN topology to guide the choice of parametrization:

- **max** is the maximum performance scenario where it is desirable to reach the peak FPS permitted by the platform, topology and FINN's architecture.

- **fix** represents a scenario with a fixed FPS requirement, which is often determined by an I/O device for real life applications. For instance, consider a 640×480 video stream at 30 FPS, which is to be chopped up into 32×32 tiles for neural network inference. Handling this task with real-time performance would require a BNN inference rate of 9000 FPS, which we set as the requirement for this usage scenario.

We use shortened names to refer to the prototypes, e.g., CNV-fix refers to the prototype that implements the **CNV** topology for the **fix** usage scenario. For each prototype, the folding factors (Section C1.4.4) were determined to meet the requirements of its usage scenario, and the FINN design flow (Section C1.4.3) was followed to generate the hardware accelerator. Vivado HLS and Vivado version 2016.3 were used for the bitfile synthesis. A target clock

Table C1.2.: Summary of workloads.

Topology	Params (Mbits)	Ops (M)	Off-chip I/O (B)	Op.Int. (Ops/B)
SFC	0.3	0.6	112	5970
LFC	2.9	5.8	112	51968
CNV	1.5	112.5	3092	36400
Prototype	**Per-Layer Total Fold (F)**			
SFC-max	13, 16, 16, 16			
SFC-fix	12544, 16384, 16384, 2560			
LFC-max	104, 128, 128, 128			
LFC-fix	13312, 16384, 16384, 10240			
CNV-max	8100, 7056, 5184, 7200, 5184, 4608, 8192, 8192, 1280			
CNV-fix	16200, 14112, 10368, 14400, 10368, 9216, 16384, 16384, 1280			

frequency of 200 MHz was used for both Vivado HLS and Vivado, and to run the resulting accelerator unless otherwise stated. The salient properties of the topologies and folding factors for the prototypes are summarized in Table C1.2.

All prototypes were implemented on a Xilinx Zynq-7000 All Programmable SoC ZC706 Evaluation Kit running Ubuntu 15.04. The board contains a Zynq Z7045 SoC with dual ARM Cortex-A9 cores and FPGA fabric with 218600 LUTs and 545 BRAMs. The host code runs on the Cortex-A9 cores of the Zynq. It initializes 10000 images with test data in the Zynq's shared DRAM, launches and times the accelerator execution to measure classification throughput, then measures accuracy by comparing against the correct classifications. Two power measurements P_{chip} and P_{wall} are provided for each experiment; P_{chip} using the PMBus interface to monitor the FPGA power supply rails, and P_{wall} using a wall power meter for the total board power consumption. The measurements are averaged over a period of 10 seconds while the accelerator is running.

Table C1.3.: Summary of results from FINN 200 MHz prototypes.

Name	Throughput (FPS)	Latency (µs)	LUT	BRAM	P_{chip} (W)	P_{wall} (W)
SFC-max	12361 k	0.31	91131	4.5	7.3	21.2
LFC-max	1561 k	2.44	82988	396	8.8	22.6
CNV-max	21.9 k	283	46253	186	3.6	11.7
SFC-fix	12.2 k	240	5155	16	0.4	8.1
LFC-fix	12.2 k	282	5636	114.5	0.8	7.9
CNV-fix	11.6 k	550	29274	152.5	2.3	10

C1.5.2. Results

Table C1.3 provides an overview of the experimental results, in terms of classification throughput, latency to classify one image, FPGA resource usage and power. The **max** scenario results are perhaps the best summary of the potential of BNNs on FPGAs, with SFC-max achieving 12.3 million classifications per second at 0.31 µs latency while drawing less than 22 W total power. All **fix** results meet and exceed the 9000 FPS requirement by 30% due to folding factors being integers, though lower throughput and power could have been achieved by using a slower clock. We focus on particular aspects of the results in the following subsections.

Maximum Throughput and Bottlenecks

To assess the quality of results for the **max** scenarios, we compare the achieved performance (XNOR–popcount operations per second) with the peak throughput in TOPS indicated by the roofline model. Figure C1.9 presents a roofline model (Section C1.3.1) for the ZC706, assuming 90% LUT utilization, 200 MHz clock frequency and 1.6 GB/s of DRAM bandwidth. The vertical lines show the arithmetic intensities for the topologies, and the actual operations per second values from corresponding prototypes with **max** usage scenarios are indicated as points on those lines. All **max** prototypes achieve performance in the TOPS range, but are bottlenecked due to different factors. CNV-max achieves 2.5

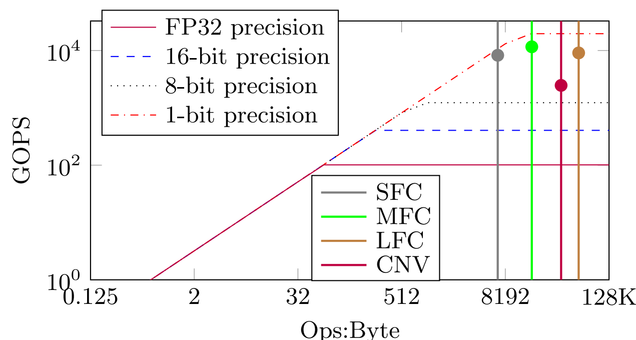


Figure C1.9.: ZC706 roofline with topologies and **max**-datapoints.

TOPS and is *architecture-bound*. The current SWU design does not scale as well as the MVTU and constitutes a bottleneck, which will be addressed in future work. Despite its higher complexity, observe that CNV-max actually requires ~2× fewer LUTs than SFC-max since the folding parameters for CNV-max are chosen in accordance with the maximum performance dictated by the bottleneck. SFC-max achieves 8.2 TOPS and is *memory-bound*. Observe that the SFC arithmetic intensity line intersects the memory-bound (sloped) part of the roofline, thus the performance cannot be scaled up without adding more DRAM memory bandwidth. LFC-max achieves 9.1 TOPS, which is 46% of the roofline, and is *resource-bound*. As folding factors are integers, the smallest increment is 2× which roughly doubles the resource cost. The FPGA has enough LUTs but not enough BRAMs to accommodate doubled resource cost, thus leaving ~30% of BRAMs unused. A 3x512-neuron fully connected topology, labeled MFC in Figure C1.9, was able to achieve 11.6 TOPS and 6238 kFPS with 95% of the device BRAMs.

Energy Efficiency

It is desirable to minimize the energy spent per image classification, which corresponds to maximizing FPS per Watt when many images are to be classified. To help evaluate the energy efficiency, Figure C1.10 plots the achieved FPS per Watt for the prototypes for both the wall power and FPGA power readings. In general, we see that the higher FPS prototypes have better energy efficiency,

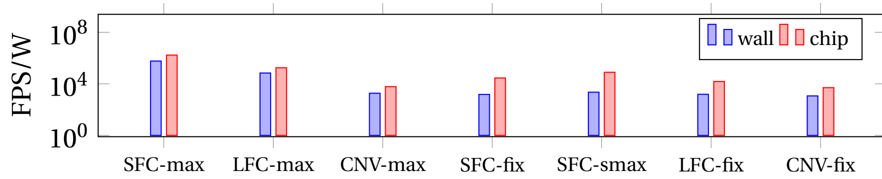


Figure C1.10.: Prototype energy efficiency.

with SFC-max offering 583066 FPS per W of total power and outperforming all other prototypes by at least an order of magnitude. It is also worth noting that the board's idle power consumption is about 7 W, which forms a lower bound on all wall power measurements, and could be improved by e.g., using LPDDR memory.

To maximize energy efficiency with a fixed target FPS, is it better to use a highly parallel design at low clock frequency, or a less parallel design at high clock frequency? We ran an additional experiment to investigate this question by slowing down the SFC-max prototype to meet the **fix** FPS requirement of 9000 FPS. By clocking it at 250 kHz, we obtained a classification throughput of 15731 FPS with 0.2 W of FPGA power. The result is labeled SFC-smax in Figure C1.10, and is over 2× more energy efficient than SFC-fix. This suggests that a high degree of parallelism benefits energy efficiency as long as the FPGA resources are available.

Resource Efficiency

We consider two aspects of resource efficiency for FINN: how efficiently the compute units are used during runtime (*runtime efficiency*), and how efficiently FPGA resources are turned into compute units (*mapping efficiency*).

To assess runtime efficiency, we divide the FPS-based (measured) operations per cycle ($\text{FPS} \cdot \text{Ops}/F_{\text{clk}}$) by the (peak) number of synaptic operations per cycle from the design ($\sum 2 \cdot P \cdot S$). The prototypes exhibit good runtime efficiency, with ~70% for **CNV**, ~80% for **SFC** and ~90% for **LFC**. The efficiency can be increased further by fine-tuning the folding factors between different layers.

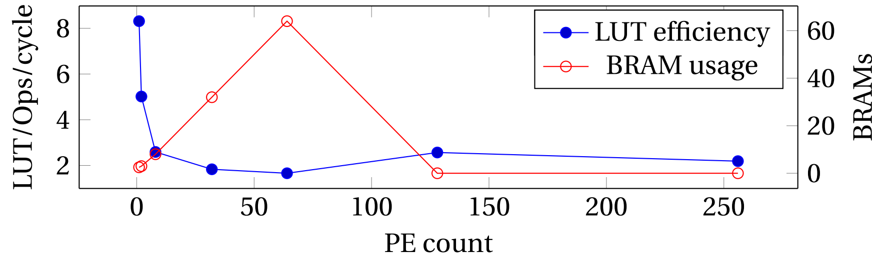


Figure C1.11.: Mapping resource efficiency.

Evaluating the mapping efficiency directly on the prototypes loses some insight, since **CNV** uses LUTs on SWU and PU, while fully-connected topologies do not. Instead, for a single 256×256 fully-connected layer, we fix $S = 64$ and vary P, and plot the LUTs per synaptic operation in Figure C1.11, which should be minimized to maximize efficiency. The LUTs per operation decreases with higher P since the fixed-size control logic is amortized between more PEs and reaches a minimum of 1.83 for $P = 64$, but increases again for $P > 64$. To understand why, we also plot the number of BRAMs used in the same figure. Although all designs have the same number of BNN parameters, the number of BRAMs increases with P since each PE needs its own weight and threshold memories. This also means a significant part of the BRAM storage capacity is unused for $1 < P \leq 64$, since the same amount of network parameters is divided between a greater number of memories. This is also visible for SFC-fix and SFC-max, which use the same network parameters, but have almost $10\times$ difference in the number of BRAMs used (15.5 vs 130.5) since SFC-max has more compute elements working in parallel. Here, with $P > 64$, so little of each BRAM is used that Vivado HLS implements the weight and threshold memories using LUTs, which causes the LUTs per operation to increase. Thus, the depth and number of BRAMs, and the LUT-to-BRAM ratio of the FPGA plays a key role in determining how well the resources will be utilized by a BNN. For instance, on another FPGA with the same amount of LUTs but twice the number of half-depth BRAMs, LFC-max could achieve $2\times$ throughput.

C1.5.3. Comparison to prior work

From an application perspective, we suggest that the current best way to compare different platforms is to simply compare their accuracy, FPS and power consumption when working on the same benchmark datasets (MNIST, CIFAR-10 and SVHN). This comparison is provided in Table C1.4, and is divided into three sections: our results, prior work on low-precision (< 4 bits) networks, and prior work with higher-precision (> 4 bits) networks.

When it comes to pure image throughput, our designs outperform all others. For the MNIST dataset, we achieve an FPS which is over 48/6× over the nearest highest throughput design [2] for our SFC-max/LFC-max designs respectively. While our SFC-max design has lower accuracy than the networks implemented by Alemdar et al. [2] our LFC-max design outperforms their nearest accuracy design by over 6/1.9× for throughput and FPS/W respectively. For other datasets, our CNV-*max* design outperforms TrueNorth [14] for FPS by over 17/8× for CIFAR-10 / SVHN datasets respectively, while achieving 9.44× higher throughput than the design by Ovtcharov et al. [41], and 2.2× over the fastest results reported by Hegde et al. [18]. Our prototypes have classification accuracy within 3% of the other low-precision works, and could have been improved by using larger BNNs.

A recent work by Nurvitadhi et al. [38] compares binary matrix-vector operation performance and efficiency on FPGA, ASIC, GPU and CPU. Their results indicate that CPU and GPUs are severely underutilized for binary synaptic operations, and that FPGAs are only ~8× less energy efficient than ASICs in this case. As they do not provide results on end-to-end network implementations, we do not include them in Table C1.4. Our 11.6 TOPS MFC prototype (Section C1.5.2) is 20% faster than the 9.6 TOPS reported in their work.

C1.6. Conclusion

This work demonstrates the performance and energy efficiency potential of recently proposed BNNs for image classification. They are particularly well-suited for FPGA implementations as parameters can be fit entirely in OCM and arithmetic is simplified, enabling high computational performance. The novel

Table C1.4.: Comparison to prior work. Metrics not reported by prior work are indicated by dashes (-), and our estimates by ~ .

Name	Dataset	Platform	Bits	Err. (%)	kFPS	P_{chip} (W)	P_{wall} (W)	kFPS/P_{chip}	kFPS/P_{wall}	GOPS
SFC-max	MNIST	ZC706	1	4.17	12,361	7.3	21.2	1693.29	583.07	8,265.45
LFC-max	MNIST	ZC706	1	1.60	1,561	8.8	22.6	177.39	69.07	9,085.67
MFC-max	MNIST	ZC706	1	2.31	6,238	11.3	28.5	552	218.8	11,612.86
CNV-max	CIFAR-10	ZC706	1	19.90	21.9	3.6	11.7	6.08	1.87	2,465.5
CNV-max	SVHN	ZC706	1	5.10	21.9	3.6	11.7	6.08	1.87	2,465.5
Alemdar et al. [2]	MNIST	Kintex-7 160T	2	2.24	255.10	0.32	-	806.45	-	~96.68
Alemdar et al. [2]	MNIST	Kintex-7 160T	2	1.71	255.10	1.84	-	138.50	-	~448.47
Alemdar et al. [2]	MNIST	Kintex-7 160T	2	1.67	255.10	2.76	-	92.59	-	~864.03
Park and Sung [43]	MNIST	ZC706	3	-	70	4.98	-	14.06	-	~210
TrueNorth [14]	CIFAR-10	TrueNorth	1	16.59	1.249	0.2044	-	6.11	-	-
TrueNorth [14]	SVHN	TrueNorth	1	3.34	2.526	0.2565	-	9.85	-	-
CaffePresso [18]	MNIST	Keystone-II	16	-	5	-	14	-	0.357	44.82
CaffePresso [18]	CIFAR-10	Keystone-II	16	-	10	-	14	-	0.714	146.14
CaffePresso [18]	MNIST	Parallella	32	-	0.64	-	5	-	0.129	5.78
CaffePresso [18]	CIFAR-10	Parallella	32	-	0.1	-	5	-	0.019	1.40
Ovtcharov et al. [41]	CIFAR-10	Stratix V D5	32	~11-26	2.32	-	25	-	0.093	-

parameterizable dataflow architecture and optimizations presented enable unprecedented classification rates, minimal power consumption and latency, while offering the flexibility of C++ design entry and the scalability required for accelerating larger and more complex networks. We hence believe that this technology is eminently suitable for embedded applications requiring real-time response, including surveillance, robotics and augmented reality. Future work will focus on providing support for non-binary low precision, implementing larger networks like AlexNet, higher performance convolutions, and a more thorough design space exploration. Finally, FINN assumes that all BNN parameters can fit into the available OCM of a single FPGA. Supporting external memory, multi-FPGAs implementations and reconfiguration [54] could improve the utility of our approach.

Acknowledgments

The authors would like to thank the NTNU HPC lab and colleagues at Xilinx Research Labs for their support. This work was supported under the Australian Research Councils Linkage Projects funding scheme (project number LP130101034).

Paper C2

Scaling Binarized Neural Networks on Reconfigurable Logic

Nicholas Fraser, Yaman Umuroğlu, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre and Kees Vissers

Published in
Proceedings of the 8th Workshop and 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM)

C2. Scaling Binarized Neural Networks on Reconfigurable Logic

Abstract.

BNNs are gaining interest in the deep learning community due to their significantly lower computational and memory cost. They are particularly well suited to reconfigurable logic devices, which contain an abundance of fine-grained compute resources and can result in smaller, lower power implementations, or conversely in higher classification rates. Towards this end, the FINN framework was recently proposed for building fast and flexible FPGA accelerators for BNNs. FINN utilized a novel set of optimizations that enable efficient mapping of BNNs to hardware and implemented fully connected, non-padded convolutional and pooling layers, with per-layer compute resources being tailored to user-provided throughput requirements. However, FINN was not evaluated on larger topologies due to the size of the chosen FPGA, and exhibited decreased accuracy due to lack of padding. In this paper, we improve upon FINN to show how padding can be employed on BNNs while still maintaining a 1-bit datapath and high accuracy. Based on this technique, we demonstrate numerous experiments to illustrate flexibility and scalability of the approach. In particular, we show that a large BNN requiring 1.2 billion operations per frame running on an ADM-PCIE-8K5 platform can classify images at 12 kFPS with 671 μs latency while drawing less than 41 W board power and classifying CIFAR-10 images at 88.7% accuracy. Our implementation of this network achieves 14.8 trillion operations per second. We believe this is the fastest classification rate reported to date on this benchmark at this level of accuracy.

C2.1. Introduction

CNNs provide impressive classification accuracy in a number of application domains, but at the expense of large compute and memory requirements [33]. A significant body of research is investigating compression techniques combining numerous approaches such as: weight and synapse pruning; data compression techniques such as quantization, weight sharing and Huffman coding; and reduced precision with fixed point arithmetic [16, 22, 23]. Recently, an extreme form of reduced precision networks, known as BNNs [11], have gained significant interest as they can be implemented for inference at a much reduced hardware cost. This is due to the fact that multipliers and accumulators become XNORs and popcounts respectively, and both are significantly lighter in regards to resource and power footprint. For example, a KU115 offers 483 billion operations per second (GOPS) compared to 46 TOPS for binary synaptic operations. This is visualized in the roofline models in Figure C2.4 which illustrates theoretical peak performance for numerous reduced precision compute operations.[1] Furthermore, the model size is greatly reduced and typically small enough to fit in OCM, again reducing power, simplifying the implementation and providing much greater bandwidth.

FINN [4] describes a framework for mapping BNNs to reconfigurable logic. However, it focuses on BNNs for embedded applications and as such, the results reported are for smaller network sizes running on an embedded platform. In this work, we briefly summarise FINN and analyse it from the perspective of scaling to larger networks and devices, such as those targeted for data centers. Firstly, we focus on several technical issues that arise when scaling networks on FINN including: BRAM usage, throughput limitations and resource overheads. We also identify several properties of CNN layers which make them map to FINN more efficiently. Our results, measured on an ADM-PCIE-8K5 platform [1], show that indeed very high image classification rates, minimal latency with very high power efficiency can be achieved by mapping BNNs to FPGAs, even though improvements may be made. Secondly, we highlight an issue of padding, a common feature of large CNNs, which may cause significant hardware overheads. We propose an alternative form of padding, which maps

[1] Assuming 70% device utilization, 250 MHz clock frequency and 178 LUTs and 2 DSPs per average floating point operation, and 2.5 LUTs per binary XNOR-popcount operation.

more efficiently to reconfigurable logic. Specifically, the contributions of this work are: 1) measured performance results for large-scale networks on an ADM-PCIE-8K5 board; 2) an analysis of FINN for large-scale problems, highlighting some bottlenecks as well as proposing solutions; and 3) a form of padding, which achieves high accuracy while also maintaining a binary datapath.

C2.2. Background

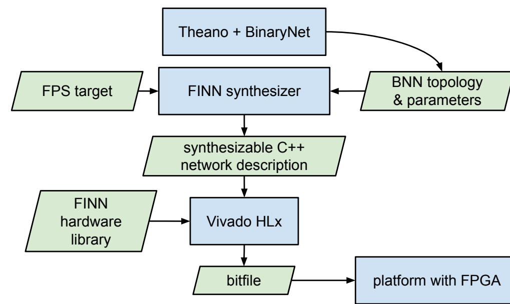
A great deal of prior work on mapping neural networks to hardware exist for FPGAs, GPUs and ASICs to help increase inference rate or improve energy efficiency. We refer the reader to the work by Misra and Saha [35] for a comprehensive survey of prior works. In general we distinguish four basic architectures: 1) a *single processing engine*, usually in the form of a *systolic array*, which processes each layer sequentially [3, 10, 41, 60]; 2) a *streaming architecture* [2, 54], consisting of one processing engine per network layer; 3) a *vector processor* [15] with instructions specific to accelerating the primitives operations of convolutions; and 4) a *neurosynaptic processor* [14], which implements many digital neurons and their interconnecting weights. Significant research investigates binarization of neural networks whereby either input activations, synapse weights or output activations or a combination thereof are binarized. If all three components are binary, we refer to this as *full binarization* [29]. If not all three components are binary, we refer to this as *partial binarization*. The seminal XNOR-Net work by Rastegari et al. [45] applies convolutional BNNs on the ImageNet dataset with topologies inspired by AlexNet, ResNet and GoogLeNet, reporting top-1 accuracies of up to 51.2% for full binarization and 65.5% for partial binarization. DoReFa-Net by Zhou et al. [61] explores reduced precision with partial and full binarization on the SVHN and ImageNet datasets, including best-case ImageNet top-1 accuracies of 43% for full and 53% for partial binarization. Finally, the work by Courbariaux et al. [11] describes how to train fully-connected and convolutional networks with full binarization and batch normalization layers, reporting competitive accuracy on the MNIST, SVHN and CIFAR-10 datasets. All BNNs used in this work are trained by a methodology based on the one described by Courbariaux et al. [11], and unset bits represent a numerical -1 value while set bits represent a +1. The downside to the high performance characteristics of BNNs is a small drop in accuracy, in comparison to

floating point networks. Improving the accuracy for reduced precision CNNs is an active research area in the machine learning community and first evidence shows that accuracy can be improved by increasing network sizes [50].

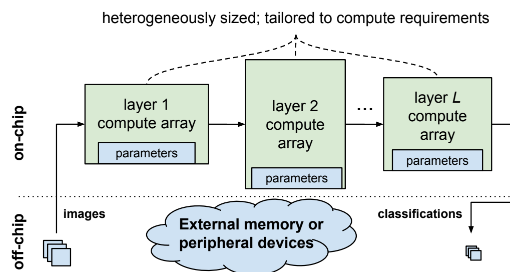
C2.3. BNNs on Reconfigurable Logic

This work builds on top of FINN [4], a framework for building scalable and fast BNN inference accelerators on FPGAs. FINN is motivated by observations on how FPGAs can achieve performance in the TOPS range using XNOR–popcount–threshold datapaths to implement the BNNs described by Courbariaux et al. [11]. Given a trained BNN and target frame rates, FINN follows the workflow in Figure C2.1a to compose a BNN accelerator from hardware building blocks. In more detail, a given network topology and model retrieved through Theano [53], together with design targets in form of resource availability and classifcation rate, is processed by the synthesizer which determines the scaling settings and produces a synthesizable C++ description of a heterogeneous streaming architecture.[2] The top-level architecture is exemplified in Figure C2.1b and has two key differentiators compared to prior work on FPGA CNN accelerators. First, all BNN parameters are kept in OCM, which greatly increases arithmetic intensity, reduces power and simplifies the design. Furthermore, one streaming compute engine is instantiated per layer, with resources tailored to fit each layer's compute requirements and the user-defined frame rate. Compute engines communicate via on-chip data streams and each produces and consumes data in the same order with the aim of minimizing buffer requirements in between layers. Thereby each engine starts to compute as soon as the previous engine starts to produce output. In essence, we build a custom architecture for a given topology rather than scheduling operations on top of a fixed architecture, as would be the case for typical systolic array based architectures, and avoid the "one-size-fits-all" inefficiencies and reap more of the benefits of reconfigurable computing.

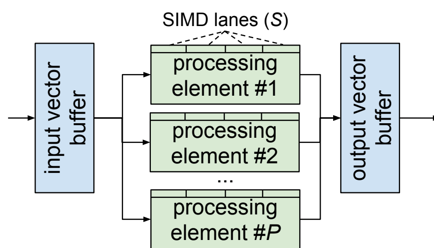
[2] To achieve portability, we chose a commercial high level synthesis tool, Vivado HLS [59], for the implementation. The tool enables faster development cycles via high-level abstractions, and provides automated pipelining to meet the clock frequency target.



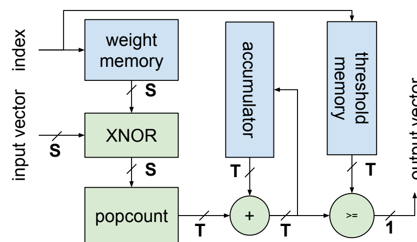
(a) Accelerator generation.



(b) Top-level architecture.



(c) Building block (MVTU).



(d) MVTU datapath.

Figure C2.1.: FINN workflow and architecture, reproduced from [4].

C2.3.1. The Matrix–Vector–Threshold Unit

In more detail, the key processing engine in FINN is the MVTU as illustrated in Figure C2.1c, which computes binarized matrix-vector products and compares against a threshold to generate a binarized activation. Convolutions are *lowered* [8] to matrix–matrix multiplications, using SWUs (described further in Section C2.4.2) to generate the image matrix and the MVTU to carry out the actual arithmetic. The SWU generates the same vectors as those in [8] but with the elements of the vector interleaved to reduce and simplify memory accesses and to avoid the need for data transposition between layers. Internally, the MVTU consists of an input and output buffer, and an array of P PEs, shown in Figure C2.1d, each with a number of SIMD lanes, S. The synapse weight matrix to be used is kept in OCM distributed between PEs, and the input images stream through the MVTU as each one is multiplied with the matrix. Each PE receives exactly the same control signals and input vector data, but multiply-accumulates the input with a different part of the matrix. A PE can be thought of as a hardware neuron capable of processing S synapses per clock cycle. Finally, the MVTU architectural template can also support partial binarization for non-binarized outputs and inputs. Removing the thresholding stage provides non-binarized outputs, while using regular multiply-add instead of XNOR-popcount can handle non-binarized inputs. These features are used in the first and last layers of networks that process non-binary input images or do not output a one-hot classification vector.

C2.3.2. Folding

Depending on the use case, a neural network inference accelerator may have different throughput requirements in terms of the images classified per second (FPS). In FINN, FPS is controlled by the per-layer parameters P (number of PEs in an MVTU) and S (number of SIMD lanes in each PE). If the number of synapses, Y, connected to a neuron is greater than S, then the computation is *folded* across the PE, with the resulting PE producing an activation every $F^s = Y/S$ clock cycles. Similarly, if the number of neurons, X, in a layer exceeds P, then each PE is responsible for calculating activations for $F^n = X/P$ neurons. In total, it would take the MVTU $F^s \cdot F^n$ clock cycles to compute all its neuron activations. The MVTUs are then rate balanced by adjusting their P and S values

to match the number of clock cycles it takes to calculate all required activations for each layer. As this is a balanced streaming system, the classification throughput FPS will be approximately F_{clk}/II, where F_{clk} is the clock frequency, and the II (Initiation Interval) is equal to the total folding factor $F^{\text{tot}} = F^s \cdot F^n$ cycles for a fully-connected layer. Note that convolutional layers have an extra folding factor, F^m, which is the number of matrix–vector products which need to be computed, i.e., the number of pixels in a single OFM. Therefore, for convolutional layers the total folding factor is: $F^{\text{tot}} = F^s \cdot F^n \cdot F^m$.

C2.3.3. BNN-specific Operator Optimizations

The methodology described in [11] forms the basis for training all BNNs in this paper. Firstly, in regards to arithmetic, we are using 1-bit values for all input activations, weights and output activations (full binarization), where an unset bit represents -1 and a set bit represents +1. Binary dot products result in XNORs with popcounts (which count the number of set bits instead of accumulation with signed arithmetic). Secondly, all BNN layers use batch normalization [24] on convolutional or fully connected layer outputs, then apply the sign function to determine the output activation. In [4] it is shown how the same output can be computed via thresholding, which combines the bias term, batch normalization and activation into a single function. Finally, the networks described in [11] perform pooling prior to activations, i.e., pooling is performed on non-binarized numbers, which are then batch normalized and fed into the activation function. However, as shown in [4], pooling can be equally performed after activation, once binarized, in which case it can be effectively implemented with the Boolean OR-operator.

C2.4. Padding for BNN Convolutions

This section describes the improvements made to FINN in this work.

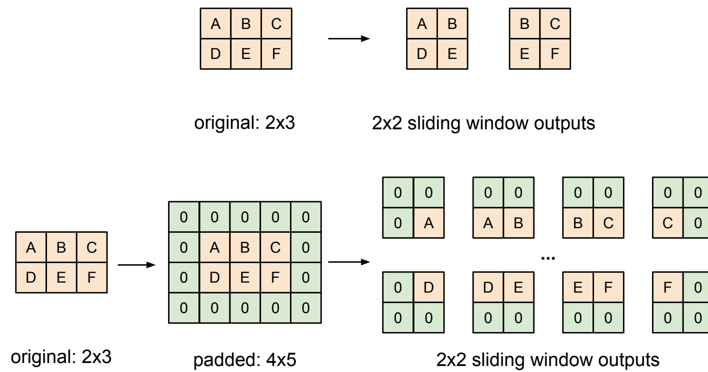


Figure C2.2.: Convolution without (top) and with (bottom) padding.

C2.4.1. Padding using nonzero values

Zero-padding is commonly applied for convolutional layers in deep neural networks, in order to prevent the pixel information on the image borders from being "washed away" too quickly [28]. Figure C2.2 illustrates the sliding window outputs on the same image with and without padding. Observe that the pixels on the border (such as A and F) occur more frequently in the sliding window outputs when padding is used, thus preventing them from being "washed away" too quickly in the next layer.

A challenge arises for zero-padding in the context of BNNs with only $\{-1,+1\}$ arithmetic: there is no zero value defined. In fact, the original BinaryNet [11] paper uses ternary values $\{-1,0,+1\}$ for the forward pass, with zeros used for padding. However, ternary values require two bits of storage, essentially doubling the OCM required to store values and the bitwidth of the datapath. Since FINN focuses on BNNs that fit entirely into on-chip memory of a single FPGA, minimizing the resource footprint is essential. Thus, a padding solution that avoids ternary values is preferable. A straightforward solution would be to use e.g., -1 as the padding value, and expect that the BNN learns weights which compensate for these values. Surprisingly, -1-padding works just as well as 0-padding according to our results, which are presented in Section C2.5.2.

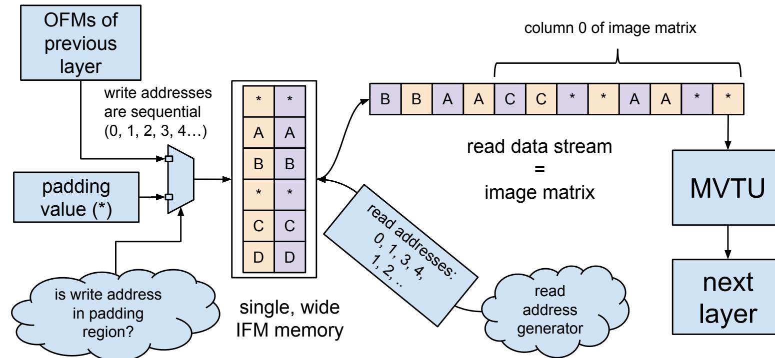


Figure C2.3.: FINN SWU enhanced with streaming padding.

C2.4.2. Streaming padding for FINN

FINN lowers [8] convolutions to matrix-matrix multiplication of the filter weight matrix with the image matrix. The image matrix is generated on-the-fly by the SWU. Figure C2.3 illustrates how the FINN SWU is enhanced to support streaming padding for convolution layers. The key operational principle is the same as in FINN. Namely, a single, wide IFM memory is used to store the feature maps into OCM in the order they arrive, and the addresses that correspond to the sliding window pixels are read out. Padding is achieved by a multiplexer that chooses the data source for writing into the IFM memory. If the current write address falls into the padding region, the padding value (e.g., -1) is written into the memory; otherwise, an element from the output stream of the previous layer is written instead.

C2.5. Evaluation

C2.5.1. Experimental Setup

BNN Topologies

The network topologies used for our experiments are all based on the CNN topology described in [11], which we denote as cnn. This topology is inspired

by the VGG16 network [48], which consists of three groups of (3x3 convolution – 3x3 convolution – 2x2 maxpooling) layers, and two fully-connected layers at the end. To explore how FINN performs on a range of network sizes, we introduce a scaling factor, σ, to scale the width of each layer, and denote the resulting topology as cnn(σ). Note that σ does not influence the number of layers in a network, it merely affects: 1) the number of neurons in each fully connected layer; and 2) the number of filters in each convolutional layer. Specifically, cnn(0.5) has half as many filters in each convolutional layer and half as many neurons in each fully connected layer, compared to the CNN described in [11]. In terms of convolutional networks, [4] only evaluated a single non-padded BNN topology ($\text{cnn}_{\text{NoPad}}(^1/_2)$). In this work, we consider $\text{cnn}(^1/_2)$ as well as smaller ($\text{cnn}(^1/_4)$) and bigger (cnn(1)) padded convolutional topologies to investigate how FINN scales.

In order to simulate a realistic use case, we consider an application with a fixed FPS requirement, i.e., real-time object recognition of a video stream. If one considers an 800 × 600 video stream at 25 FPS, which partitioned into tiles of 32 × 32 for classification. In order to classify the tiles in real-time, a classification rate of approximately 12 kFPS would be required. We use this image rate as our target for all experiments and adjust the number of PEs and SIMD accordingly in each layer of each design.

The Platform

The target board is an Alpha Data ADM-PCIE-8K5 which features a Xilinx Kintex UltraScale XCKU115-2-FLVA1517E FPGA (KU115). The KU115 offers 663k LUTs, 2160 BRAMs (36k) and 5520 DSPs and is running at 125 MHz for our experiments. The host machine is a IBM Power8 8247-21L with 80 cores at 3.69 GHz and 64 GB of RAM and it is running Ubuntu 15.04. In all experiments, all parameters are stored in OCM while the test images and the predicted labels are read from and written to the host memory directly. The provided resource counts include the PCI Express infrastructure used for moving data streams as well as the BNN accelerator. Although we are not able to provide per-experiment power measurements, the maximum power consumption observed for this board was 41 W on a board power dissipation benchmark test, and

Table C2.1.: Accuracy with different padding modes for CIFAR-10.

		Padding Mode		
		no-padding	0-padding	-1-padding
Scale	$\sigma = 1/4$	75.6%	78.2%	79.1%
	$\sigma = 1/2$	80.1%	85.2%	85.2%
	$\sigma = 1$	84.2%	88.6%	88.3%

Table C2.2.: Operations per image with different padding modes for CIFAR-10.

		Padding Mode		
		no-padding	0-padding	-1-padding
Scale	$\sigma = 1/4$	30.4 M	78.5 M	78.5 M
	$\sigma = 1/2$	118.9 M	310.3 M	310.3 M
	$\sigma = 1$	530.1 M	1234.1 M	1234.1 M

we expect that the real power dissipation values for BNN accelerators will be significantly lower than this.

C2.5.2. Effects of Padding

To investigate how different padding modes affect accuracy, we trained a set of convolutional BNNs on the CIFAR-10 dataset with different scaling factors (σ). The convolutions used are 3×3, so one pixel of padding is added on each border. The results are summarized in Table C2.1. As expected, using 0-padding improves accuracy by 4-5% compared to no-padding, indicating that the conventional wisdom on padding increasing accuracy also applies to BNNs. Furthermore, we can see that the accuracy of -1-padded networks are on par with the 0-padded ones of same scale. This suggests that BNNs are able to learn to compensate for the -1 values used for padding by adjusting the weight values and thresholds, and the accuracy benefits can be still obtained with a binary (as opposed to ternary) datapath.

It should also be noted that no-padding results in a significant reduction in

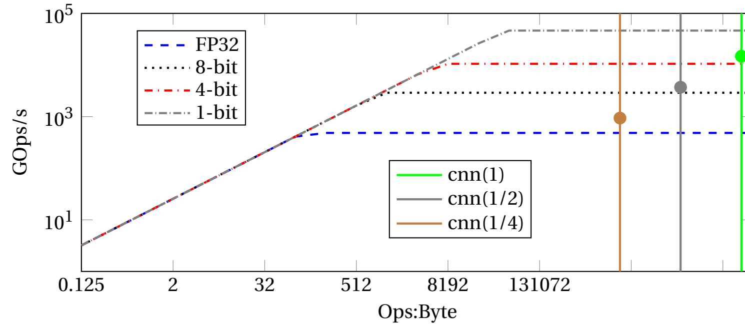


Figure C2.4.: KU115 roofline with different datatypes.

the amount of operations per frame and the number of parameters. Thus, it is worthwhile to examine the computation versus accuracy tradeoffs in the context of padding. Table C2.2 lists the total number of XNOR-popcount operations necessary to classify one image using different padding modes and scaling factors. We can observe that the no-padding topology variant for the same scale factor requires $2-3\times$ less computation. However, this comes at a cost of higher error rate, and a smaller-but-padded network may be advantageous over a larger-but-not-padded network. For instance, cnn($^1/_4$) classifies at 79% accuracy using 78.5 M operations, whereas the $\text{cnn}_{\text{NoPad}}(^1/_2)$ classifies at 80.1% accuracy using 118.9 M operations. Thus, cnn($^1/_4$) may be preferable due to its lower computational cost if a 1% drop in accuracy is acceptable for the use case at hand.

C2.5.3. Scaling to Larger Networks

A results summary is shown in Table C2.3 which also shows the accuracy achieved by the implemented networks on a number of benchmark datasets. The new padded CNN results are provided in the top portion of Table C2.3, while key results from [4] are shown in the lower portion. Note that for comparison, scaled versions of the multilayer perceptrons (MLPs) consisting only of fully-connected layers described in [11] are also shown and denoted as mlp(σ).

We can see that larger networks scale well to larger FPGAs, with our best designs achieving 14.8 TOPS and 671 μs image classification latency. Furthermore, even

Table C2.3.: Key performance and resource utilization results achieved by this work (top) and FINN (bottom) on a number of BNN topologies.

	Network	Device	LUT	BRAM	kFPS	GOps/s
Padded	cnn($^1/_4$)	KU115	35818	144	12.0	938
	cnn($^1/_2$)	KU115	93755	386	12.0	3,711
	cnn(1)	KU115	392947	1814	12.0	14,814
FINN [4]	$\text{cnn}_{\text{NoPad}}$($^1/_2$)	Z7045	54538	192	21.9	2,466
	mlp($^1/_{16}$)	Z7045	86110	130.5	12,361	8,265
	mlp($^1/_8$)	Z7045	104807	516.5	6,238	11,613
	mlp($^1/_4$)	Z7045	79097	398	1,561	9,086

with the largest network tested, all model parameters fit within OCM of the KU115 and thus avoids potential bottlenecks on external memory access. However, if we were to attempt a larger network (such as cnn(2)) the design would no longer fit in OCM without also reducing the frame rate. This is discussed further in Section C2.5.3.

While the results described in Table C2.3 represent state-of-the-art in terms of image classification rates and energy efficiency, it is still work in progress. Our best raw performance number (14.8 TOPS) outperforms that of the smaller FPGA device used in FINN [4] (11.6 TOPS), which is no surprise. However, the MLPs shown in [4] do achieve performance figures closer to the theoretical peak of the device. This is mostly due to the simplicity of MLPs versus CNNs. Figure C2.4 shows the estimated peak performance of the KU115 with vertical lines indicating the arithmetic intensity of the 3 CNN networks and coloured markers indicating actual performance of FINN. We can see that our implementations still fall below the KU115's theoretical peak. We expect that with planned improvements, including those in Section C2.5.3, significant performance gains can still be achieved. However it should be noted, that the largest design cnn(1) shown in Table C2.3 requires 1.2 GOPS per frame, which is similar in computational requirements to the popular AlexNet [31] which requires 1.45 GOPS per frame. In comparison the GPUs, the NVidia Titan X can achieve 3.2 kFPS at 227 W for AlexNet inference, compared to 12 kFPS at less than 41 W on

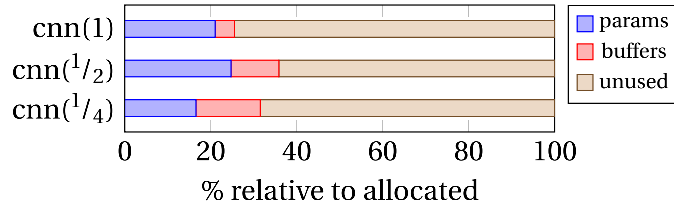


Figure C2.5.: Utilization of allocated BRAM storage space.

the KU115 FPGA [40]. It should be noted that these figures are in terms of 32-bit floating point operations, as opposed to the binarized ones discussed in this work. However, high accuracy has been achieved by fully binarized [21] and partially binarized [61] versions of AlexNet and we expect to be able to achieve high performance on such networks.

BRAM Efficiency

Since FINN currently focuses on BNNs that fit entirely onto the on-chip memory of a single FPGA, making the most out of the available on-chip memory is essential. Figure C2.5 illustrates how much of the allocated BRAM space (as reported by Vivado) is actually utilized by the accelerator. The two largest contributors to BRAM usage in FINN are the network parameters (BNN weights and thresholds), and stream buffers (such as FIFOs and input-output buffers), which are shown with different colors in the bar chart. As can be expected, the majority of the utilized storage is for weights, although the streaming buffers occupy roughly equal storage for cnn($^1/_4$) since there are not as many parameters.

A bigger concern is that on average only ~22% of the storage space in the allocated BRAMs is actually used. For scaling to even larger networks, this under–utilization could constitute a problem as synthesis will fail trying to allocate more BRAMs than is available in the FPGA. Further analysis into this issue revealed that this is a consequence of how convolutions are currently handled in FINN. Recall that the total folding factor is $F^{\text{tot}} = F^s \cdot F^n \cdot F^m$ for a convolution layer. The F^m folding factor here arises due to implementing matrix–matrix products as a sequence of matrix–vector products Unlike F^s and F^n, F^m is currently not controllable, since only one matrix–vector product is computed at a time in each MVTU. When high FPS is desired, the initiation

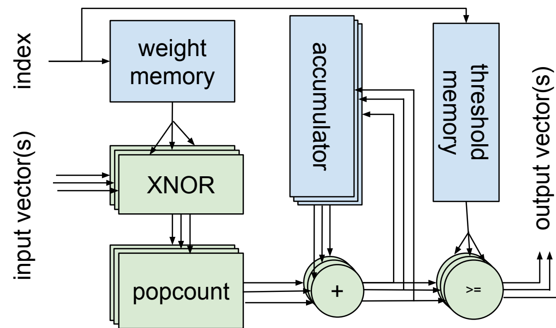


Figure C2.6.: Datapath for matrix–multiple vector product.

interval must be minimized, which can only be achieved by small values F^n and F^s since F^m is constant. This requires creating many PEs and SIMD lanes operating in parallel, each of which have their own weight and threshold memories operating independently. However, this causes the weight matrix to be split and distributed into many small pieces, thus causing the observed storage under–utilization.

One way of addressing this problem would be enabling control over the F^m parameter by enhancing the MVTU to enable multiplying the same matrix by multiple vectors in parallel. In this manner, fewer PEs and SIMD lanes could be instantiated, each working on a larger portion of the weight matrix and utilizing BRAM storage better. Figure C2.6 shows how the MVTU datapath could be enhanced to support multiple vectors, broadcasting the same data from the weight memory to multiple XNOR-popcount-accumulate datapaths. Note that only the datapath is duplicated; the weight and threshold memories have a single copy. We leave further investigation of the matrix–multiple vectors for future work.

C2.6. Conclusion

In this work, we explored the scaling of BNNs on large FPGAs using the FINN framework. We highlight an issue with padding in convolutional layers in BNNs described in [11] which would cause them to require a 2-bit datapath. We show that a small modification to padding (padding with -1 values) improves

accuracy over no-padding and is comparable to 0-padding, while still allowing networks to maintain a binary datapath. We found that high performance for large networks can be attained, with our highest demonstrated performance achieving 12 kFPS at less than 41 W of board power and 14.8 TOPS of raw computational performance. When scaling to large networks, we also show that the efficiency of BRAM usage in FINN is low, and propose an architectural modification which would allow for better BRAM utilization. Alternatively, if a higher number of smaller BRAMs were available on FPGAs devices, this would allow FINN to better exploit the available resources.

For future work, we will further enhance the FINN framework to support partial binarization, and different kinds of convolutional layers, such as inception layers [52] and fire-modules [23]. The architectural improvements, described in Section C2.5.3 will be implemented to further improve the BRAM usage efficiency of architectures produced by FINN. Further networks which have been trained on larger datasets, i.e., ImageNet, will also be implemented. Finally, better power measurements will be attained rather than using “worst-case” power dissipation values.

Paper C3

Streamlined Deployment for Quantized Neural Networks

Yaman Umuroğlu and Magnus Jahre

Presented at
International Workshop on Highly Efficient Neural Networks Design (HENND),
part of Embedded Systems Week (ESWEEK) 2017

C3. Streamlined Deployment for Quantized Neural Networks

Abstract. Running Deep Neural Network (DNN) models on devices with limited computational capability is a challenge due to large compute and memory requirements. Quantized Neural Networks (QNNs) have emerged as a potential solution to this problem, promising to offer most of the DNN accuracy benefits with much lower computational cost. However, harvesting these benefits on existing mobile CPUs is a challenge since operations on highly quantized datatypes are not natively supported in most instruction set architectures (ISAs). In this work, we first describe a *streamlining* flow to convert all QNN inference operations to integer ones. Afterwards, we provide techniques based on processing one bit position at a time (bit-serial) to show how QNNs can be efficiently deployed using common bitwise operations. We demonstrate the potential of QNNs on mobile CPUs with microbenchmarks and on a quantized AlexNet, which is 3.5× faster than an optimized 8-bit baseline.

C3.1. Introduction

From voice recognition to object detection, *Deep Neural Networks (DNNs)* are steadily getting better at extracting information from complex raw data. Combined with the popularity of mobile computing and the rise of the Internet-of-Things (IoT), there is enormous potential for widespread deployment of intelligent devices, but a computational challenge remains. A modern DNN can require billions of floating point operations to classify a single image, which is far too costly for energy-constrained mobile devices. Offloading DNNs to

powerful servers in the cloud is only a limited solution, as it requires significant energy for data transfer and cannot address applications with real-time or low-latency requirements, such as augmented reality or navigation for autonomous drones.

Quantized Neural Networks (QNNs) have recently emerged as a potential solution to this problem. They contain convolutional, fully-connected, pooling and normalization layers similar to the floating point variants, but use a constrained set of values to represent each weight and activation in the network. We will use the notation $\mathbf{W^w A^a}$ to refer to a QNN with w-bit weights and a-bit activations, and focus on cases where they represent *few-bit integers* $(w, a \leq 4)$. The computational advantages of such QNNs are two-fold:

1. Each parameter and activation can be represented with a few bits. A greater portion of the working set can thus be kept in on-chip memory, enabling greater performance, reducing off-chip memory accesses and the energy cost of data movement.

2. Most QNN operations are on few-bit integers, which are faster and more energy-efficient than floating-point.

While a quantized network will generally have reduced accuracy compared to an equivalent DNN using floating point, recent research has demonstrated significant progress in closing this accuracy gap. Courbariaux and Hubara et al. [11] first demonstrated that Binarized Neural Networks (BNNs), a QNN variant with $\mathbf{W^1 A^1}$, could achieve competitive accuracy on smaller image recognition benchmarks like CIFAR-10 and SVHN. XNOR-Net [45] improved upon this technique by adding scaling factors to better approximate the full-precision operations. Noting that more challenging classification tasks such as ImageNet could benefit from higher-precision activations, DoReFa-Net [62] used multi-bit activations and weights to further improve accuracy. Recently, Cai et al. [7] proposed Half-wave Gaussian Quantization (HWGQ) to take advantage of the Gaussian-like distribution of batch-normalized activations, demonstrating $\mathbf{W^1 A^2}$ networks with less than 5% top-5 accuracy drop compared to floating point DNNs on the challenging ImageNet dataset, as summarized in Table C3.1.

Despite the attractive accuracy and computational properties, there is a challenge in reaping the benefits on QNNs on mobile devices with commodity

Table C3.1.: Accuracy of a state-of-the-art QNN [7].

Dataset	Network	Floating Point top-1 (top-5)	$\mathbf{W^1A^2}$ HWGQ [7] top-1 (top-5)
ImageNet	AlexNet	58.5% (81.5%)	52.7% (76.3%)
ImageNet	GoogLeNet	71.4% (90.5%)	63.0% (84.9%)
ImageNet	VGG-like	69.8% (89.3%)	64.1% (85.6%)
CIFAR-10	VGG-like	93.2%	92.5%

processors. Three outstanding issues limit the benefits of QNN deployment on existing mobile CPUs: floating point parameters inside and between quantized layers, lack of native support for efficient few-bit integer matrix multiplications, and overhead of bit-masking operations for convolution lowering on few-bit activations. In this work, we show how these problems can be addressed by absorbing floating point operations into thresholds (*streamlining*), using a bit-serial formulation for handling few-bit integer matrix multiplications and channel-interleaved lowering.

C3.2. Streamlined QNNs

Even for layers with uniform-quantized input activations and weights, state-of-the-art QNN methods use some floating point computation in the forward pass to improve the accuracy. Although these layers do not typically contain a large amount of computation, they may still incur slowdowns on devices where floating point operations are expensive and increase the memory footprint of the QNN by adding floating point parameters. Three such examples from state-of-the-art QNN methods are:

1. **Batch normalization.** Almost all state-of-the-art QNN techniques, including BinaryNet [11], XNOR-Net [45] and HWGQ [7], use batch normalization to obtain zero mean and unit variance prior to quantizing activations. The normalization parameters μ and i are floating point values obtained during network training.

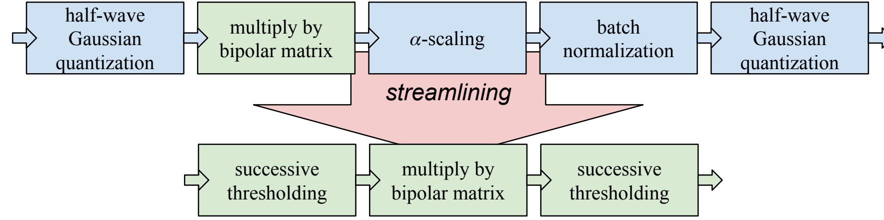


Figure C3.1.: Streamlining an HWGQ network. Blue and green color indicate floating point and integer data, respectively.

2. **α-scaling.** To better approximate the full-precision results using quantized operations, both XNOR-Net [45] and HWGQ [7] use α-scaling. This involves multiplying the quantized matrix multiplication result with α, which is a floating point vector containing the average L1-norm of each row of the weight matrix prior to quantization.
3. **Non-integer quantization levels.** The chosen quantization levels in a QNN may be floating point values to best approximate the underlying value distribution. For instance, the state-of-the-art QNNs produced by HWGQ [7] use the following function for 2-bit uniform quantization:

$$\mathrm{HWGQ}(x) = \left\{ \begin{array}{lr} 0, & \text{for } x \leq t_0 \\ 0.538, & \text{for } 0 < x \leq 0.807 \\ 1.076, & \text{for } 0.807 < x \leq 1.345 \\ 1.614, & \text{for } 1.345 < x \end{array} \right\}$$

C3.2.1. The Streamlining Algorithm

Through a process we call *streamlining*, we show how the forward pass through any QNN layer with uniform-quantized activations and weights can be computed using only integer operations. This consists of the following three steps:

Quantization as successive thresholding

Given a set of threshold values $t = \{t_0, t_1 \ldots t_n\}$, the successive thresholding function $T(x, t)$ maps any real number x to an integer in the interval $[0, n]$,

where the returned integer is the number of thresholds that x is greater than or equal to:

$$T(x,t) = \left\{ \begin{array}{lr} 0, & \text{for } x \le t_0 \\ 1, & \text{for } t_0 < x \le t_1 \\ \dots & \dots \\ n-1, & \text{for } t_{n-2} < x \le t_{n-1} \\ n, & \text{for } t_{n-1} < x \end{array} \right\}$$

Any uniform quantizer $Q(x)$ can be expressed as successive thresholding followed by a linear transformation such that $Q(x) = a \cdot T(x) + b$. As an example, the 2-bit uniform HWGQ quantizer can be expressed as $\text{HWGQ}(x) = 0.538 \cdot T(x,t)$ with $t_0 = 0$, $t_1 = 0.807$, $t_2 = 1.345$. It should be noted that this technique is only economical for few-bit activations, since the number of thresholds grows exponentially with the activation bitwidth.

Moving and collapsing linear transformations

Any sequence of linear transformations can be collapsed into a single linear transformation. We can first move all floating point linear operations to be positioned *between* the quantized matrix operation and the activation quantization, then collapse them into a single linear transformation. For the example in Figure C3.1, the linear transformation $ax + b$ for the previous layer's activation quantization can be moved past the bipolar matrix multiplication, since $W \cdot (ax+b) = a \cdot (Wx) + Wb$, forming a sequence together with the α-scaling and batch normalization. Afterwards, this sequence of three linear transformations can be reduced to a single linear transformation.

Absorbing linear operations into thresholds

The final step in the streamlining process is to update the threshold values as $t_i \leftarrow (t_i - b)/a$ using the parameters a, b of the linear transformation. Observe that in the inequality $t_0 < x \le t_1$ we can substitute $ax + b$ as the variable, and rewrite it as $(t_0 - b)/a < x \le (t_1 - b)/a$. By updating each threshold in this manner, we can remove the floating point linear transformations completely and

feed the result of the quantized matrix operation directly into the successive thresholding layer. Furthermore, if the input to the quantized matrix operation is known to be integer (i.e., the previous layer's activations were also quantized), each threshold can be simply rounded up to the nearest integer without changing the produced results.

C3.3. Inference with Few-Bit Weights and Activations on Mobile CPUs

The dominating computation in QNN inference is convolutions between feature maps and kernels expressed as few-bit integers, which can be *lowered* [9] to matrix-matrix multiplication between few-bit integer matrices. Both the lowering and the matrix multiplications can be carried out by casting all operands to 8-bit integers, which are natively supported by most ISAs today. Libraries such as Google's gemmlowp [5], which has been used to deploy DNNs on mobile devices, offer high-performance 8-bit matrix multiplications. However, using 8-bit operands to carry out few-bit integer operations can be wasteful. For instance, using 8-bit operations to compute a $\mathbf{W^2A^2}$ matrix product would insert six zero bits into each operand, thus unnecessarily increasing the memory footprint by 4×. Here, we provide alternatives that take advantage of few-bit integers for both the lowering and matrix multiplication operations.

C3.3.1. Few-Bit Integer Matrix Multiplication

To perform efficient few-bit integer matrix multiplication, we propose to use commonly supported bitwise operations in *bit-serial* fashion. We will first describe how this is done for the $\mathbf{W^1A^1}$ case, then generalize the method to $\mathbf{W^wA^a}$.

The $\mathbf{W^1A^1}$ case. Binary matrix multiplication, referred to here as BinaryGEMM, can be used for the case where each weight and activations can be represented using a single bit. Previous work [11, 39, 45] discussed how binary dot products can be implemented using bitwise XNOR followed by popcount (counting the number of set bits) operations. Most modern processors provide an instruction for popcount, which enables fast BinaryGEMM implementations even on

function BINARYGEMM(W, A, res, α)
  for $r \leftarrow 0 \ldots rows - 1$ **do**
    for $c \leftarrow 0 \ldots cols - 1$ **do**
      for $d \leftarrow 0 \ldots \lceil depth/wordsize \rceil - 1$ **do**
        $res[r][c] \mathrel{+}= \alpha \cdot$ POPCOUNT($W(r,d)$ & $A(c,d)$)
      end for
    end for
  end for
end function

Algorithm C3.1: $\mathbf{W^1A^1}$ GEMM using AND-popcount.

mobile CPUs. Note that very high performance (in the trillion-operations per second range) on these operations can also be achieved with FPGAs [4, 39] and GPGPUs [11]. Although XNOR-popcount is only applicable for matrices with $\{-1, +1\}$ binary elements, it is possible to extend this idea to $\{0, 1\}$ binary elements by using bitwise AND instead of XNOR as shown in Algorithm C3.1.

The $\mathbf{W^wA^a}$ case. We now leverage BINARYGEMM as a building block for implementing few-bit integer matrix multiplication. We can rewrite the w-bit matrix W as a weighted sum $\sum_{i=0}^{w-1} 2^i \cdot W[i]$, where $W[i]$ is the binary matrix formed by taking bit i of each element of W, also referred to as a *bit plane*. In this manner, the product of two few-bit integer matrices can be written as a weighted sum of the pairwise products of their bit planes, i.e., $W \cdot A = \sum_{i=0}^{w-1} \sum_{j=0}^{a-1} 2^{i+j} \cdot W[i] \cdot A[j]$.

Algorithm C3.2 uses this observation to formulate few-bit integer matrix multiplication between the w-bit weight matrix W and the a-bit activation matrix A. This is a *vectorized bit-serial* operation, since the contributions to each result element are computed between two bit positions at a time, but the bitwise operations inside BINARYGEMM operate on vectors of bits. In this manner, we are able to take advantage of the full width of the processor datapath without introducing a large number of zero bits inside operations regardless of the values of w and a. The time taken by BITSERIALGEMM will be proportional to $w \cdot a$, with $\mathbf{W^1A^1}$ executing fastest.

```
function BitSerialGEMM(W, A, res)
  for i ← 0 ... w − 1 do
    for j ← 0 ... a − 1 do
      sgnW ← (i == w − 1 ? −1 : 1)
      sgnA ← (j == a − 1 ? −1 : 1)
      BinaryGEMM(W[i], A[j], res, sgnW · sgnA · 2^(i+j))
    end for
  end for
end function
```

Algorithm C3.2: Signed $\mathbf{W^w A^a}$ GEMM using BinaryGEMM.

C3.3.2. Lowering with Few-Bit Activations

Lowering convolutions [9] allows taking advantage of optimized matrix-matrix multiplications, which is the approach we take in this work. Memory usage is typically a concern for lowering due to duplicated pixels. In theory, QNNs do not suffer as much from this problem since quantized activations use much fewer bits per pixel, but taking advantage of this on a CPU can be tricky. Namely, the lowering process itself (often called *im2col*) requires accessing the feature map data in a "sliding window" fashion, which may require bit masking and shifting operations that decrease performance. Representing each few-bit activation as an 8-bit value avoids this problem, but introduces many unused zero padding bits.

We propose an alternative, which we refer to as *interleaved lowering*, that uses a bit-serial, channel-interleaved data layout as illustrated in Figure C3.2. Each pixel, which may span one or more CPU words, contains the bits from one bit position across activations from all channels, padded to the nearest word boundary. Afterwards, the lowering can be performed on the granularity of entire CPU words, with one im2col per activation bit.

C3.4. Evaluation

We implemented BitSerialGEMM using ARM NEON intrinsics in C++, with register blocking and L1 cache blocking to achieve higher performance. We

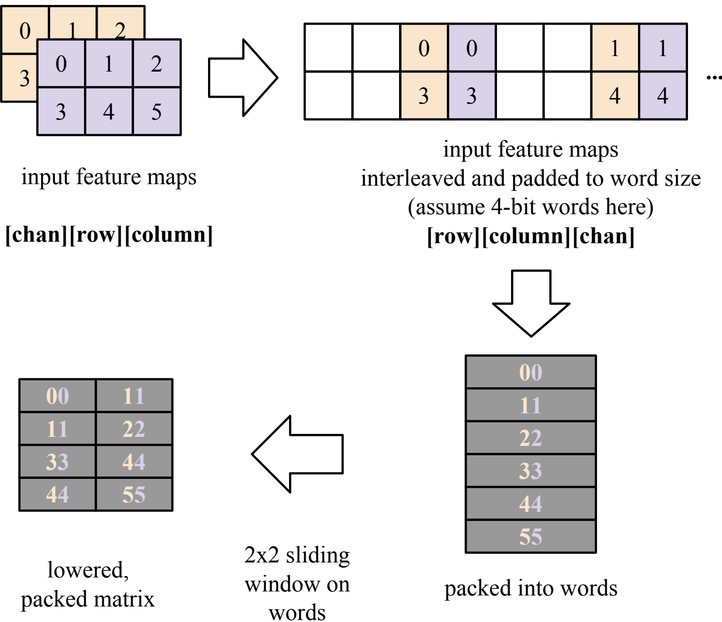


Figure C3.2.: Interleaved lowering.

compare against the gemmlowp library [5], which utilizes hand-optimized inline assembly for 8-bit matrix multiplications. All reported results are measured on a single ARM Cortex-A57 core running at 1.9 GHz on the Nvidia Jetson TX1 board. We use 8-bit native matrix multiplications provided by gemmlowp [5] as the baseline alternative to BITSERIALGEMM.

C3.4.1. Matrix Multiplication Microbenchmarks

As matrix multiplication accounts for the majority of time in neural network inference, we start by evaluating BITSERIALGEMM on matrix multiplication microbenchmarks. For a (rows, depth, cols) operation that takes T nanoseconds, we report the performance in integer giga-operations per second (GOPS) measurement as $(2 \cdot \text{rows} \cdot \text{depth} \cdot \text{cols}) \ / \ T$ by averaging over a runtime of 10 s.

Compute-Bound Performance

To measure the maximum achievable performance with our implementation, we use the largest matrices that still fit into the L1 cache. For gemmlowp, we observed a peak performance of 22 GOPS. For BITSERIALGEMM on $\mathbf{W^1A^1}$ (binary matrices), we observed a peak performance of 150 GOPS, which is 6.8× faster than using 8-bit operands. As expected, the performance linearly decreases with more bits of precision: 77 GOPS for $\mathbf{W^1A^2}$, 50 GOPS for $\mathbf{W^1A^3}$, 34 GOPS for $\mathbf{W^2A^2}$ and 23 GOPS for $\mathbf{W^2A^3}$. Thus, for this particular platform, BITSERIALGEMM is faster than using 8-bit operations for $\mathbf{W^wA^a}$ with $w \cdot a \leq 6.8$ when working with in-cache matrices.

Performance vs Matrix Size

To investigate how performance is influenced by the dimensions of a $M \times N \times K$ matrix multiplication, we performed a sweep of different sizes between 2^6 and 2^{12} in each dimension using both gemmlowp and BITSERIALGEMM with $\mathbf{W^1A^1}$. Figure C3.3 presents a scatter plot of the performance with increasing depth (K). We observe that BITSERIALGEMM performance is sensitive to the depth (K) dimension. For small matrix sizes, there is little or no performance advantage

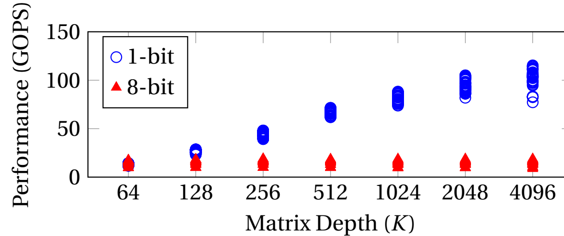


Figure C3.3.: Log-linear plot of performance versus depth (K).

over gemmlowp, which should be taken into consideration when choosing the execution method for each layer. BITSERIALGEMM quickly becomes faster with increasing depth and becomes advantageous over gemmlowp, up to 6.6× faster than gemmlowp for a $64 \times 1024 \times 4096$ multiplication. With $K \geq 2048$, we observe decreased performance for larger M and N values in BITSERIALGEMM due to increased cache misses, which can be addressed by adding more levels of blocking to the implementation.

C3.4.2. Quantized AlexNet

To assess the benefits of the techniques discussed for QNN deployment, we developed a version of Caffe with support for quantized layers. Each quantized layer can be configured individually to use either gemmlowp or BITSERIALGEMM as the execution engine. We use a quantized AlexNet from [7] with batch size 1 as a benchmark, with $\mathbf{W^8A^8}$ for the first layer, $\mathbf{W^8A^2}$ for the last layer, and $\mathbf{W^1A^2}$ for all other matrix layers. The first and last layer are computed using gemmlowp in 8-bit precision to preserve accuracy. For non-matrix layers such as thresholding and max pooling which constitute a tiny portion of the total compute, we use regular floating point operations. We evaluated the performance for the following combinations of techniques:

- **baseline:** No streamlining, all layers using gemmlowp.
- **bsgemm:** Streamlining, all but the first and last layer using BITSERIALGEMM, first and last layer in gemmlowp.

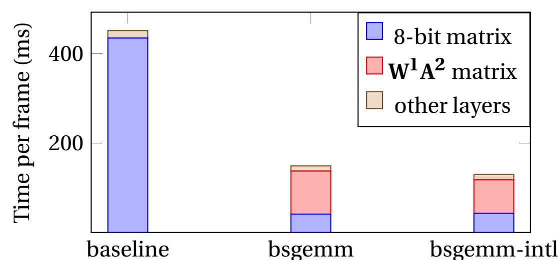


Figure C3.4.: Overall performance for quantized AlexNet.

- **bsgemm-intl:** Streamlining and interleaving, all but the first and last layer using BitSerialGEMM .

Overall performance

Figure C3.4 compares the time per frame with each of the three techniques and presents a basic breakdown of time cost. Overall, bsgemm achieves a 3× speedup over the baseline, and bsgemm-intl further improves this to 3.5×. Speedups from bsgemm are limited by the presence of 8-bit first/last layers, which account for 33% of the execution time in bsgemm-intl. Also quantizing those layers further would bring further performance benefits. The current throughput is 2.2, 6.7 and 7.7 frames per second respectively for **baseline**, **bsgemm** and **bsgemm-intl**, and the performance can be further improved by multi-core parallelism and code optimization.

Detailed comparison.

Table C3.2 presents a detailed breakdown of time spent in lowering and matrix multiplication operations across AlexNet convolutional and fully-connected layers with different optimizations. All matrix multiplications are indicated in parantheses as (rows, columns, depth), with the midline separating convolutional and fully-connected layers. The advantage of using quantized operations for fully-connected layers is especially prominent, with speedups of up to 50× owing to increased arithmetic intensity in matrix-vector multiplications. For matrix-matrix multiplications in convolutional layers, the advantage of using

Table C3.2.: Time cost breakdown for $\mathbf{W^1A^2}$ AlexNet with batch size 1. Best numbers for each row are highlighted.

	Operation	Time (ms)			Speedup
		baseline	bsgemm	bsgemm-intl	
convolutional	lowering	**6.7**	**6.7**	**6.7**	1×
	(96, 363, 3025)	**20.7**	**20.7**	**20.7**	1×
	lowering	8.7	15	**0.9**	10×
	(256, 2400, 729)	90.3	23.7	**23.7**	3×
	lowering	2.4	3.7	**0.2**	12×
	(384, 2304, 169)	32.2	8.3	**8.3**	4×
	lowering	3.5	5.7	**0.3**	10×
	(384, 3456, 169)	48	10.7	**10.7**	5×
	lowering	3.5	5.7	**0.3**	10×
	(256, 3456, 169)	35.8	7.3	**7.3**	5×
FC	(4096, 9216, 1)	114.7	2.3	**2.3**	50×
	(4096, 4096, 1)	52.9	1.1	**1.1**	50×
	(1000, 4096, 1)	**13**	**13**	**13**	1×

$\mathbf{W^1A^2}$ BITSERIALGEMM is around 4×. When the matrix multiplies become faster, the overhead of lowering and bit packing costs become substantial, almost as much as the matrix-matrix multiplication time for earlier layers with large filters. Fortunately, this can be remedied by interleaving, as indicated by the results in the **bsgemm-intl** column. By taking advantage of packing bits prior to lowering, interleaved lowering can offer a 10× speedup over the baseline.

C3.5. Conclusion and Future Work

We have presented methods for efficient processing of QNNs on mobile CPUs via absorbing scaling factors into thresholds, channel-interleaved lowering, and bit-serial matrix multiplication. Our results indicate that these methods can take better advantage of few-bit operations in QNNs, offering significant speedups over native 8-bit operations on mobile CPUs. As future work, we note that this approach can enable approximate computing by only considering

the contributions from higher-order bits and taking advantage of bit-level sparsity. Another use for this technique would be on-device training DNNs using low-precision gradients [12], which also requires low-precision matrix operations.

Bibliography

[1] *ADM-PCIE-8K5 Datasheet*. 1.3. Alpha Data. Sept. 2016.

[2] H. Alemdar, N. Caldwell, V. Leroy, A. Prost-Boucle and F. Pétrot. 'Ternary Neural Networks for Resource-Efficient AI Applications'. In: *CoRR* abs/1609.00222 (2016).

[3] R. Andri, L. Cavigelli, D. Rossi and L. Benini. 'YodaNN: An Ultra-Low Power Convolutional Neural Network Accelerator Based on Binary Weights'. In: *CoRR* abs/1606.05487 (2016). URL: `http://arxiv.org/abs/1606.05487`.

[4] Anonymous. 'FINN: A Framework for Fast, Scalable Binarized Neural Network Inference'. In: *Under Review* (2016).

[5] B. J. et al. *gemmlowp: a small self-contained low-precision GEMM library*. `https://github.com/google/gemmlowp`. 2017.

[6] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry et al. 'An updated set of basic linear algebra subprograms (BLAS)'. In: *ACM Transactions on Mathematical Software* 28.2 (2002), pp. 135–151.

[7] Z. Cai, X. He, J. Sun and N. Vasconcelos. 'Deep Learning with Low Precision by Half-wave Gaussian Quantization'. In: *CVPR*. 2017.

[8] K. Chellapilla, S. Puri and P. Simard. 'High performance convolutional neural networks for document processing'. In: *Proceedings of the International Workshop on Frontiers in Handwriting Recognition*. Suvisoft. 2006.

[9] K. Chellapilla, S. Puri and P. Simard. 'High performance convolutional neural networks for document processing'. In: *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft. 2006.

[10] Y.-H. Chen, J. Emer and V. Sze. 'Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks'. In: *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture*. IEEE, 2016.

[11] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv and Y. Bengio. 'Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1'. In: *CoRR* abs/1602.02830 (2016). URL: http://arxiv.org/abs/1602.02830.

[12] C. De Sa, M. Feldman, K. Olukotun and C. Re. 'Understanding and Optimizing Asynchronous Low-Precision Stochastic Gradient Descent'. In: *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE. 2017.

[13] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun and R. Fergus. 'Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation'. In: *Advances in Neural Information Processing Systems*. (Montreal, Quebec, Canada). 2014, pp. 1269–1277. URL: http://papers.nips.cc/paper/5544-exploiting-linear-structure-within-convolutional-networks-for-efficient-evaluation.

[14] S. K. Esser, P. A. Merolla, J. V. Arthur, A. S. Cassidy, R. Appuswamy, A. Andreopoulos, D. J. Berg, J. L. McKinstry, T. Melano, D. R. Barch et al. 'Convolutional Networks for Fast, Energy-Efficient Neuromorphic Computing'. In: *CoRR* abs/1603.08270 (2016).

[15] C. Farabet, C. Poulet, J. Y. Han and Y. LeCun. 'CNP: An FPGA-based processor for convolutional networks'. In: *Proceedings of the International Conference on Field Programmable Logic and Applications*. IEEE. 2009, pp. 32–37.

[16] S. Han, H. Mao and W. J. Dally. 'Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman coding'. In: *CoRR* abs/1510.00149 (2015).

[17] S. Han, J. Pool, J. Tran and W. J. Dally. 'Learning both Weights and Connections for Efficient Neural Networks'. In: *CoRR* abs/1506.02626 (2015). URL: http://arxiv.org/abs/1506.02626.

[18] G. Hegde, Siddhartha, N. Ramasamy and N. Kapre. 'CaffePresso: An Optimized Library for Deep Learning on Embedded Accelerator-based platforms'. In: *Proc. CASES*. 2016.

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

[20] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto and H. Adam. 'MobileNets: Efficient convolutional neural networks for mobile vision applications'. In: *arXiv preprint arXiv:1704.04861* (2017).

[21] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv and Y. Bengio. 'Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations'. In: *CoRR* abs/1609.07061 (2016).

[22] F. N. Iandola, K. Ashraf, M. W. Moskewicz and K. Keutzer. 'FireCaffe: near-linear acceleration of deep neural network training on compute clusters'. In: *CoRR* abs/1511.00175 (2015).

[23] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally and K. Keutzer. 'SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 1MB model size'. In: *CoRR* abs/1602.07630 (2016).

[24] S. Ioffe and C. Szegedy. 'Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift'. In: *Proceedings of the International Conference on Machine Learning, ICML*. Lille, France, 2015, pp. 448–456. URL: `http://jmlr.org/proceedings/papers/v37/ioffe15.html`.

[25] J.-w. Jang, S. Choi and V. Prasanna. 'Area and time efficient implementations of matrix multiplication on FPGAs'. In: *Field-Programmable Technology, 2002.(FPT). Proceedings. 2002 IEEE International Conference on*. IEEE. 2002, pp. 93–100.

[26] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers et al. 'In-datacenter performance analysis of a tensor processing unit'. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM. 2017, pp. 1–12.

[27] B. Kågström, P. Ling and C. Van Loan. 'GEMM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark'. In: *ACM Transactions on Mathematical Software (TOMS)* 24.3 (1998), pp. 268–302.

[28] A. Karpathy. *CS231n: Convolutional Neural Networks for Visual Recognition*. URL: `http://cs231n.github.io/convolutional-networks/`.

[29] M. Kim and P. Smaragdis. 'Bitwise Neural Networks'. In: *CoRR* 1601.06071 (2016). URL: `http://arxiv.org/abs/1601.06071`.

[30] A. Krizhevsky and G. Hinton. 'Learning multiple layers of features from tiny images'. In: *Technical Report* (2009).

[31] A. Krizhevsky, I. Sutskever and G. E. Hinton. 'Imagenet classification with deep convolutional neural networks'. In: *Advances in Neural Information Processing Systems*. 2012, pp. 1097–1105.

[32] Y. LeCun, Y. Bengio and G. Hinton. 'Deep learning'. In: *Nature* 521.7553 (2015), pp. 436–444.

[33] Y. LeCun, L. Bottou, Y. Bengio and P. Haffner. 'Gradient-based learning applied to document recognition'. In: *Proceedings of the of the IEEE* 86.11 (1998), pp. 2278–2324.

[34] B. Liu, M. Wang, H. Foroosh, M. F. Tappen and M. Pensky. 'Sparse Convolutional Neural Networks'. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. (Boston, MA, USA). 2015, pp. 806–814. DOI: `10.1109/CVPR.2015.7298681`. URL: `http://dx.doi.org/10.1109/CVPR.2015.7298681`.

[35] J. Misra and I. Saha. 'Artificial neural networks in hardware: A survey of two decades of progress'. In: *Neurocomputing* 74.1–3 (2010), pp. 239–255.

[36] S. Muralidharan, K. O'Brien and C. Lalanne. 'A Semi-Automated Tool Flow for Roofline Anaylsis of OpenCL Kernels on Accelerators'. In: *Proceedings of the Workshop on Heterogeneous High-performance Reconfigurable Computing* (2015).

[37] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu and A. Y. Ng. 'Reading digits in natural images with unsupervised feature learning'. In: *NIPS Workshop on Deep Learning and Unsupervised Feature Learning* (2011).

[38] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh and D. Marr. 'Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC'. In: *Proc. ICFPT*. 2016.

[39] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh and D. Marr. 'Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC'. In: *Field-Programmable Technology (FPT), 2016 International Conference on*. IEEE. 2016.

[40] NVIDIA. 'GPU-Based Deep Learning Inference: A Performance and Power Analysis'. In: *White Paper* (2015).

[41] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss and E. Chung. *Accelerating Deep Convolutional Neural Networks Using Specialized Hardware*. Feb. 2015. URL: `https://www.microsoft.com/en-us/research/publication/accelerating-deep-convolutional-neural-networks-using-specialized-hardware/`.

[42] E. Park, J. Ahn and S. Yoo. 'Weighted-Entropy-Based Quantization for Deep Neural Networks'. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 5456–5464.

[43] J. Park and W. Sung. 'FPGA based implementation of deep neural networks using on-chip memory only'. In: *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2016, pp. 1011–1015.

[44] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang and H. Yang. 'Going Deeper with Embedded FPGA Platform for Convolutional Neural Network'. In: (Monterey, CA, USA). 2016, pp. 26–35. DOI: `10.1145/2847263.2847265`. URL: `http://doi.acm.org/10.1145/2847263.2847265`.

[45] M. Rastegari, V. Ordonez, J. Redmon and A. Farhadi. 'XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks'. In: *Proceedings of the European Conference on Computer Vision*. 2016.

[46] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg and L. Fei-Fei. 'ImageNet Large Scale Visual Recognition Challenge'. In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: `10.1007/s11263-015-0816-y`.

[47] J. Schmidhuber. 'Deep learning in neural networks: An overview'. In: *Neural Networks* 61 (2015), pp. 85–117. DOI: 10.1016/j.neunet.2014.09.003. URL: http://dx.doi.org/10.1016/j.neunet.2014.09.003.

[48] K. Simonyan and A. Zisserman. 'Very deep convolutional networks for large-scale image recognition'. In: *CoRR* abs/1409.1556 (2014).

[49] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. B. K. Vrudhula, J. Seo and Y. Cao. 'Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks'. In: (Monterey, CA, USA). 2016, pp. 16–25. DOI: 10.1145/2847263.2847276. URL: http://doi.acm.org/10.1145/2847263.2847276.

[50] W. Sung, S. Shin and K. Hwang. 'Resiliency of Deep Neural Networks under Quantization'. In: *CoRR* abs/1511.06488 (2015). URL: http://arxiv.org/abs/1511.06488.

[51] V. Sze, Y.-H. Chen, T.-J. Yang and J. Emer. 'Efficient processing of deep neural networks: A tutorial and survey'. In: *arXiv preprint arXiv:1703.09039* (2017).

[52] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke and A. Rabinovich. 'Going deeper with convolutions'. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 1–9.

[53] Theano Development Team. 'Theano: A Python framework for fast computation of mathematical expressions'. In: *CoRR* abs/1605.02688 (May 2016). URL: http://arxiv.org/abs/1605.02688.

[54] S. I. Venieris and C.-S. Bouganis. 'fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs'. In: *Proceedings of the IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE. 2016, pp. 40–47.

[55] T. Weyand, I. Kostrikov and J. Philbin. 'PlaNet - Photo Geolocation with Convolutional Neural Networks'. In: *CoRR* abs/1602.05314 (2016). URL: http://arxiv.org/abs/1602.05314.

[56] S. Williams, A. Waterman and D. Patterson. 'Roofline: an insightful visual performance model for multicore architectures'. In: *Communications of the ACM* 52.4 (2009), pp. 65–76.

[57] S. Williams, A. Waterman and D. A. Patterson. 'Roofline: an insightful visual performance model for multicore architectures'. In: *Commun. ACM* 52.4 (2009), pp. 65–76. DOI: `10.1145/1498765.1498785`. URL: `http://doi.acm.org/10.1145/1498765.1498785`.

[58] W. A. Wulf and S. A. McKee. 'Hitting the memory wall: implications of the obvious'. In: *ACM SIGARCH computer architecture news* 23.1 (1995), pp. 20–24.

[59] Xilinx Inc. 'Vivado Design Suite User Guide: High-Level Synthesis'. In: *White Paper* (2016).

[60] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao and J. Cong. 'Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks'. In: *Proc. of the 2015 ACM/SIGDA Intl. Symp. on Field-Programmable Gate Arrays*. 2015.

[61] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu and Y. Zou. 'DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients'. In: *CoRR* abs/1606.06160 (2016). URL: `http://arxiv.org/abs/1606.06160`.

[62] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu and Y. Zou. 'DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients'. In: *CoRR* abs/1606.06160 (2016).

Part D.

Concluding Remarks

1. Conclusion

In this thesis, in order to gain insight into how the energy efficiency of computer systems can be improved via specialization, we have studied how sparse linear algebra and deep neural network inference can benefit from FPGA acceleration. This chapter will provide a summary of the contributions from the scientific articles presented in Parts B and C, organized around the research questions that were posed in Part A Section 1.3.

1.1. Research Question 1

How can sparse linear algebra benefit from FPGA acceleration?

By utilizing more off-chip memory bandwidth. Utilizing the available off-chip memory bandwidth is key to achieving high performance for sparse problems. This applies to both the sequential access patterns generated by multiple backends, which can be made up to 20% more bandwidth-efficient with the interleaved data layout described in Paper B1, and the random access patterns, which are explored in Paper B3. Modern DRAM controllers support multiple outstanding requests, and it is essential to keep the memory system busy with requests to achieve higher utilization with random access patterns. Paper B3 adopts non-blocking caches, a technique commonly used in out-of-order CPUs, to increase the MLP and achieve higher bandwidth utilization, yielding speedups of up to 3× compared to blocking caches.

By preprocessing and hardware-software caching. The random memory access component is frequently the bottleneck in sparse problems, and although random accesses can be handled rapidly by OCM this is a limited and valuable resource for all chips including FPGAs. In Papers B2 and B3, we showed how

lightweight preprocessing of the sparsity structure can yield a minimalistic upper bound on the range of random accesses. When combined with a specialized hardware cache that makes use of this information, in addition to decreasing the amount of OCM needed to service the random access component by 70% on average, this also enables avoiding cold miss penalties which can be quite significant for some sparse matrices.

By shrinking the random access volume. Sparse computation can experience a large speed-up if the random memory access component fits into on-chip memory. Thus, algorithmic reformulations that lead to a more compact random access component can be beneficial for performance. Paper B4 provides an example of this for BFS, decoupling the traversal phase from the distance-to-root-node generation phase. This reduces the random-access component to a binary vector that is more likely to fit into OCM, which enables very efficient traversal of large graphs, offering up to twice as many traversals per unit of memory bandwidth compared to prior work. This is similar to the benefits of quantization for deep neural networks studied in Part C, which alludes to the more general conclusion that reducing the memory footprint via algorithmic reformulations or approximate computing approaches can be hugely beneficial for performance and efficiency.

By balancing memory and computation resources. Sparse computation has typically very low arithmetic intensity and a problematic indirect access pattern. For instance, in Paper B4, the computation involved for sparse matrices over the Boolean semiring is no more than AND-OR operations. In terms of the decoupled design strategy in Paper B1, FPGA designs can better cater to this imbalance by allocating more resources to the memory system part and less to the compute part. In contrast, typical CPU and GPGPU architectures are designed with applications with much higher arithmetic intensity, resulting in severe underutilization of the allocated compute resources.

1.2. Research Question 2

How can deep neural network inference benefit from FPGA acceleration?

By using quantization. The key technique for reaping benefits of FPGA acceleration in DNN inference is quantization. As revealed by both the roofline analysis and the results in Papers C1 and C2, quantization has a two-fold effect on performance and energy efficiency. First and foremost, the memory footprint of a QNN is up to 32× smaller, potentially allowing the entire network to reside in OCM for even large, complex networks. This avoids most off-chip memory accesses, offering tremendous performance and energy efficiency benefits. The second benefit is a much better fit to the FPGA computing resources, since quantized operations offer much higher compute density. The combined effect is demonstrated by the FINN-generated accelerators in Paper C1, which can perform ten-thousands to millions of image classifications per second while using less than 12 W of FPGA power. Without using highly quantized arithmetic, FPGAs are at a clear disadvantage compared to GPGPUs, since most existing FPGAs (with the exception of the recent Arria and Stratix-10 models from Intel) do not contain hard blocks for floating point operations, and must implement floating point using LUTs.

By tailoring compute resources and parallelism to requirements. The fine-grained compute resources and vast parallelism of the FPGA can be specifically tailored to the particular topology of the network and user throughput requirements, as demonstrated by the FINN framework in Paper C1. This avoids the "one-size-fits-all" penalties associated with fixed-size compute arrays, achieving up to 90% utilization of instantiated resources at runtime.

By utilizing OCM efficiently. Modern FPGAs contain many blocks of SRAM memory tightly coupled with the compute fabric. Each of these memories can independently read and write data every cycle. This yields high on-chip memory bandwidth, which can take advantage of the arithmetic intensity of the QNN inference computation. Papers C1 and C2 take advantage of the OCM for both storing the QNN parameters and for carrying the activations between layers via on-chip FIFOs. Keeping the computation on-chip and parallelizing across layers also offers significant latency benefits. FINN-generated accelerators are able to classify smaller datasets such as MNIST and CIFAR-10 with sub-millisecond latency.

By taking advantage of static analysis. In terms of being able to analyze the computation statically, accelerating inference with a trained DNN is relatively straightforward, unlike the sparse linear algebra computations studied in Part

B which exhibit dynamic runtime behavior. By using the mathematical simplifications described in Papers C1 and C3, most or all floating point parameters and operations can be removed from the computation, freeing up valuable FPGA logic resources and DSP blocks. Static analysis is also what makes the tailoring of compute resources to requirements possible in Paper C1.

2. Future Work

A PhD thesis often raises more questions than it answers, and this one is no exception. This chapter presents several possible directions for future work in the context of sparse linear algebra and deep learning on FPGAs, either to address the shortcomings of particular techniques proposed in this thesis, or to explore new directions that emerged throughout the research process.

2.1. Sparse Linear Algebra

Sparse matrix partitioning for parallel nonblocking caches. The nonblocking cache scheme proposed in Paper B3 is only studied for a single SpMV processing element with a single non-blocking cache. As more bandwidth and resources become available, instantiating several processing elements with each their own cache will become essential. This creates different access patterns depending on how the sparse matrix is partitioned for parallel execution. Additionally, allowing more random access streams to the same DDR memory is likely to degrade the bandwidth, further complicating the tradeoffs and warranting further study in this direction.

Exploring further preprocessing. Lightweight analysis of the sparsity structures as exemplified in Papers B2 and B3 contributed significantly to performance when this information was exposed to hardware. Analysis of the sparsity structure can potentially be used to tackle many of the inherent problems caused by lack of locality and should be explored further, especially in the context of tailoring a reconfigurable accelerator to individual sparsity patterns.

Impact of in-package memory. Despite the techniques proposed in this thesis for increased efficiency in utilizing off-chip memory bandwidth, GPGPUs are still much faster at processing sparse matrix computations due to the sheer

amount of bandwidth they have available. However, next-generation FPGAs will include in-package DRAM stacks delivering up to a terabyte per second of bandwidth, which may change the competitive landscape. How this amount of bandwidth will influence design choices for next-generation FPGA sparse linear algebra accelerators is an interesting direction for further research. It is likely that the proposed data layout techniques in Paper C1 will become more important to be able to parallelize the execution further while keeping the bandwidth utilization high.

Sparse matrices-over-semirings on FPGAs. As effective as FPGAs might be for sparse graphs (Paper B4), the "programmability wall" still remains for non-experts wanting to take advantage of FPGA accelerators. The standardization of interfaces such as BLAS (Basic Linear Algebra Subprograms) enabled decoupling the productivity-oriented (e.g., physicists or meteorologists that need high-performance BLAS implementations) and performance-oriented (e.g., computer scientists maintaining high-performance BLAS implementations) users of dense linear algebra, leading to a cycle of virtuous growth for both sides. GraphBLAS [2] is a similar effort for high-productivity, high-performance graph algorithms, using sparse linear algebra as building blocks based on the matrices-over-semirings concept. An FPGA sparse linear algebra backend for GraphBLAS could go a long way towards making the computational capabilities available for the masses.

Distributed sparse linear algebra with FPGAs. As computational models of nature are often sparse, there is no limit to the maximum size of the sparse models that scientists would like to study. Increasingly large, sparse models will not fit into even the off-chip storage of a single node, thus requiring distributed processing over multiple FPGA nodes. Although we expect that the challenges will be similar to those faced by distributed sparse linear algebra implementations on CPU clusters [3] at a high level, the possibility of tight integration with network interfaces and custom interconnects can bring significant additional benefits for FPGAs.

2.2. Deep Neural Networks

Bit-serial for mixed precision inference. There is recent work [4] presenting evidence that the best way to preserve accuracy in QNNs may be to use different quantization on different layers. The bit-serial approach proposed and evaluated on CPUs in Paper C3 can be adopted on FPGAs towards this end. As bit-serial performance is directly proportional to binary matrix multiplication performance, which Papers C1 and C2 quantified for FPGAs, this approach has the potential to yield an accelerator that is both high performance and flexible enough to use with a wide range of QNNs.

Multi-FPGA streaming acceleration. The FINN framework proposed in Paper C1 imposes the restriction that all parameters of the QNN will fit into the OCM of a single FPGA. To cater for larger QNN deployments in the datacenter, it could be interesting to explore multi-FPGA acceleration scenarios where the layers are distributed between multiple FPGAs, using either regular Ethernet or dedicated low-latency interconnect to carry activations across partitions.

Codesign for compact QNNs. Papers C1 and C2 indicate that adding more neurons can compensate for the accuracy drop due to quantization, which hints at the presence a hardware-software codesign space. Given the logic resource and OCM capacity of a certain FPGA, what is the best performance, energy efficiency and accuracy attainable by an appropriately-sized QNN? Exploring this vast design space will almost certainly require simple models for cost-benefit analysis, but would be beneficial from the perspective of deploying neural networks on limited-capacity devices.

Quantized training for QNNs. The inference problems studied in this thesis aside, training QNNs still takes tremendous floating point computing power and energy. However, there is evidence [1, 5] that the training may be possible to carry out in reduced precision as well. Given their computational capabilities in reduced precision, FPGAs could be an excellent match for quantized training.

Bibliography

[1] C. De Sa, M. Feldman, C. Ré and K. Olukotun. 'Understanding and Optimizing Asynchronous Low-Precision Stochastic Gradient Descent'. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM. 2017, pp. 561–574.

[2] J. Kepner, D. Bader, A. Buluç, J. Gilbert, T. Mattson and H. Meyerhenke. 'Graphs, matrices, and the GraphBLAS: Seven good reasons'. In: *Procedia Computer Science* 51 (2015), pp. 2453–2462.

[3] S. Lee and R. Eigenmann. 'Adaptive runtime tuning of parallel sparse matrix-vector multiplication on distributed memory systems'. In: *Proceedings of the 22nd annual international conference on Supercomputing*. ACM. 2008, pp. 195–204.

[4] E. Park, J. Ahn and S. Yoo. 'Weighted-Entropy-Based Quantization for Deep Neural Networks'. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 5456–5464.

[5] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen and H. Li. 'TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning'. In: *arXiv preprint arXiv:1705.07878* (2017).