ARCHITECTURAL SUPPORT FOR ACCELERATING MACHINE LEARNING INFERENCE

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Skand Hurkat

August 2018

ARCHITECTURAL SUPPORT FOR ACCELERATING MACHINE LEARNING INFERENCE

Skand Hurkat, Ph.D.

Cornell University 2018

Machine learning has become ubiquitous over recent years, prompting many studies of architectures for accelerating these algorithms. At the same time, algorithms themselves are rapidly evolving, which means that any accelerators designed have to be efficient not just at the algorithms used today, but also at any future algorithms that may be developed. This thesis explores the question, 'Surely specialised accelerators have their space in this landscape, but is there a case to be made for an architecture that can provide competitive performance at low power, while remaining highly programmable and future-proof?' This work tries to answer that question with a 'Yes', presenting VIP (Versatile Inference Processor), a system employing well-understood concepts of vector-processing and near-data processing with some modest, but key modifications that are critical to its performance. Through detailed microarchitecture simulations, it shows that VIP achieves competitive performance on a number of machine learning algorithms such as belief propagation (BP) on Markov random fields (MRFs), and deep neural networks (DNNs) including convolutional neural networks (CNNs), multi-layer perceptrons (MLPs) and recurrent neural networks (RNNs). Through synthesis of RTL code for a VIP processing engine (PE), it shows that the requirements for VIP are modest – VIP's 128 PEs require 18 mm^2 in area and consume 3.4 W to 4.8 W of power.

BIOGRAPHICAL SKETCH

Skand Hurkat studied electrical engineering at the Indian Institute of Technology, Bombay (IIT-B), working with Dr. Subhasis Chaudhuri on a system to automatically read car numberplates and recognise their models from video streams (a project that would lead to him being awarded an undergraduate research award – URA-01 by IIT-B), and with Dr. Abhay Karandikar on scheduling users in orthogonal frequency-division multiple access (OFDMA) networks. While at IIT-B, he had the pleasure of being instructed by Dr. Dinesh K. Sharma and Dr. Sachin Patkar who instilled in him an interest in digital logic design and computer architecture.

On joining the Computer Systems Lab (CSL) at Cornell University, Skand was privileged to join Dr. José F. Martínez' M3 research group. There, he used his knowledge of computer vision and machine learning, signal processing, and computer architecture to work on developing accelerators for belief propagation in probabilistic graphical models, which eventually morphed into this thesis on developing processors for accelerating inference in machine learning algorithms.

Dedicated to my parents

whose love, support, and encouragement

has carried me through every endeavour

ACKNOWLEDGEMENTS

Working towards my PhD over the last six years has been an experience like no other. Looking back, I feel incredibly grateful to have a number of people who have helped me along the way.

I must start by thanking my parents, who have been instrumental in instilling in me, the sense of curiosity that is so critical to an engineer. They motivated me to pursue the sciences, to work hard towards the IIT-JEE, to take up electrical engineering at IIT-B. They encouraged me to pursue a PhD once I completed undergraduate studies, have been very patient with me over the last six years, and have borne a significant brunt of the frustrations that are sometimes part of the process. On a lighter note, I would be lying if I denied that exceeding my mother's scholastic achievements and her Master of Pharmacy degree wasn't any factor in my decision to pursue a PhD early on.

At the same time, I cordially thank my adviser, Dr. José F. Martínez, who has been instrumental in my development as a researcher. His confidence in my abilities often seemed to exceed my own, and I'm immensely grateful to him for his many words of encouragement, advice, and wisdom throughout these years. I'm also incredibly thankful for the independence he offered me, allowing me to explore my interests and to develop something that I'm truly proud of. I greatly value the many discussions we've had over the years. His motivation spurred me to accomplish things that I did not think myself capable of accomplishing, and I only have to look back at myself six years ago to see how far I've come along in this journey. José has been more than an adviser and a mentor, he has also been a friend. The only thing we haven't agreed upon yet is that PCs are superior to Macs in every conceivable way.

My committee members, Dr. Christopher Batten and Dr. Zhiru Zhang have

been almost as instrumental in helping me succeed. My first serious attempt at developing an accelerator for belief propagation was in collaboration with Zhiru, using high-level synthesis to rapidly prototype accelerators onto an FPGA; this project helped crystallise the initial insights that evolved into this thesis. Chris has likewise been most helpful through these years, and I cannot thank him enough for his guidance on simulation, RTL development, and his readiness to help me debug code even at 11 o'clock on the night before a deadline.

My work has been the result of numerous collaborations with others. I must thank Dr. Eriko Nurvitadhi, who mentored me as an intern, gave me an idea that would develop into an FPGA implementation of hierarchical TRW-S and be published at FPL 2015. Working with Eriko was a pleasure, and I value the lessons learnt during our collaboration. Eriko also introduced me to Dr. Jungwook Choi, then a PhD Candidate working with Dr. Rob Rutenbar. Together with José, we developed the hierarchical TRW-S idea to completion, and I am incredibly thankful for Jungwook for his contributions. I must gratefully thank Dr. Avinash Sodani, Dr. Ulf Hanebutte, Dr. Jim Ballingall and others at Cavium, who not only funded the research on VIP, but also provided useful feedback and suggestions along the way.

I must also thank all my friends at Cornell, and my colleagues at the Computer Systems Lab. I value all the years of productive discussions and fun conversations that we've had, that have made these years seemingly fly by. Finally, I have to thank all the people at Cornell, too many to list in a page, that have made this part of my life so enjoyable.

This thesis was supported in part by the Intel Science and Technology Center for Embedded Computing; by AFOSR grant FA9550-15-1-0311; and by a contract with Cavium.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ALU arithmetic logic unit
ARC array range check
ASIC application specific integrated circuit

BP belief propagation

CGRA coarse-grained reconfigurable array
CMOS complementary metal oxide semiconductor
CPU central processing unit

DIMM dual in-line memory module
DMA direct memory access
DNA deoxyribonucleic acid

ECL emitter-coupled logic

FFT fast Fourier transform
FIFO first in, first out
FPGA field-programmable gate array

GPU graphics processing unit

HBM high-bandwidth memory
HMC hybrid memory cube

IC integrated circuit
ISA instruction set architecture

LSI large-scale integration

MAC multiply accumulate
MAP maximum a posteriori probability
MIMD multiple instruction multiple data
MRF Markov random field

NN neural network
- **CNN** convolutional neural network
- **DNN** deep neural network
- **LSTM** long short-term memory
- **MLP** multi-layer perceptron
- **RNN** recurrent neural network

PE processing engine
PGM probabilistic graphical model
PIM processing in memory

RAM random-access memory
- **BRAM** block RAM
- **DRAM** dynamic RAM
  - **DDR** double data rate
  - **GDDR** graphics double data rate
  - **LPDDR** low power double data rate
- **ReRAM** resistive RAM
- **SRAM** static RAM

ReLU rectified linear unit
RISC reduced instruction set computer
RTL register transfer level

SERDES serial-deserial
SIMD single instruction multiple data
SIMT single instruction multiple thread
SM streaming multiprocessor
SPMD single program multiple data
SVE scalable vector extensions

TPU tensor processing unit
TRW tree-reweighted belief propagation
- **TRW-S** sequential tree-reweighted belief propagation

TSV through-silicon via

VLIW very long instruction word

CHAPTER 1

INTRODUCTION

The landscape of computer architecture is changing rapidly over the past few years. Serious physical limitations such as the end of Dennard scaling [23] and Moore's law [83], along with a changing application landscape have resulted in a significant shift from improving single-thread performance to improving the energy efficiency of these workloads. In this new space, machine learning and artificial intelligence algorithms, particularly deep neural networks (DNNs) have received special attention from computer architects, resulting in a significant number of highly-specialised accelerators being developed for certain types of DNNs, such as convolutional neural networks (CNNs), multi-layer perceptrons (MLPs) and recurrent neural networks (RNNs).

A class of machine learning algorithms that has so far been overlooked is probabilistic graphical models (PGMs), which are fundamentally different from DNNs in the way they approach learning. PGMs are generative algorithms, i.e. they first model the underlying processes that generated the data before labelling the data, whereas DNNs are discriminative algorithms, i.e. they simply seek to label data without modelling the generating process. As a result, we can look into PGMs and understand the decision process, while DNNs remain black-boxes. These algorithms also have different application domains – inference on PGMs is used for computer vision tasks such as image de-noising, depth-from-stereo, and detecting optical flow [33]; analysing DNA sequences [34]; helping pathologists identify disease [48]; etc. DNNs are used in a number of other tasks, such as identifying objects in images [112], machine translation [130], captioning images [124], etc.

Many recent accelerator proposals in the computer architecture and the FPGA communities have focused on individual algorithms or a narrowly defined class of algorithms. For example, Cambricon by Liu et al. [80], Eyeriss by Chen et al. [16], tensor processing unit (TPU) by Jouppi et al. [59], Tetris by Gao et al. [39], and work by Wei et al. [129] are some accelerators for neural networks. Similarly, work by Choi and Rutenbar [19, 20] (later extended [52], parts of the extension are described in chapter 3), and Cheng et al. [17] proposes accelerators for belief propagation (BP) on PGMs. These accelerators take advantage of certain application characteristics to deliver high performance and energy efficiency, e.g.: accelerators for neural networks almost always have ALUs that support little other than the multiply accumulate (MAC) operation, while accelerators for BP have specialised data-paths depending on application characteristics such as the number of labels, size of image, nature of probability distributions, etc. These accelerators often cannot work with other applications or algorithms due to their high specialisation. More general architectures deliver relatively poor performance (case of many general-purpose CPUs) and/or consume a significant amount of power (case of many GPUs). The main question considered in this thesis is this, 'Surely specialised accelerators have their space in the landscape, but is there room for a system that can provide competitive performance at relatively low power?' The answer, it appears, is, 'Yes'. This thesis presents VIP (Versatile Inference Processor), a system initially designed for real-time inference in PGMs used in image and video processing (at full-HD resolution); extended to work on DNNs including CNNs, MLPs, and RNNs just by reprogramming software; delivering competitive performance on all these workloads. VIP's design is relatively well-understood, multiple independent processing engines (PEs) using a vector/SIMD processing paradigm internally.

Table 1.1: Qualitative of various classes of existing architectures and systems, including CPU, GPU, TPU, FPGA, low-power ASICs, and this work (VIP) for inference on PGMs and CNNs.

Platform	**Power**	**Throughput**		**Programmability**
		PGM	**CNN**	
CPU	Med/High	Low	Low	Very High
GPU	High	Med/High	High	Very High
FPGA	Med	Med [52]	Med [129]	Med
Tile-BP [17]	Very Low	Med/High	N/A	Very Low
Eyeriss [16]	Very Low	N/A	Low	Very Low
TPU [59]	High	N/A	Very High	Low
VIP	Low/Med	Very High	Med	High

Table 1.1 shows a rough classification of where VIP falls on the spectra of power, performance, and programmability in the space of CPUs, GPUs, FPGAs, and dedicated accelerators.

1.1 Trends Driving Computer Performance

1.1.1 The Shift to Accelerators

Moore's law [83] and Dennard scaling [23] dictated how computer performance scaled for most of the history of modern computers until the early 2000s. Moore's law was based on an observation that the number of transistors on an integrated circuit doubled every year, later revised to doubling every two years. The performance of smaller transistors was dictated by the scaling law proposed by Dennard et al. [23], which stated that as transistor feature sizes (width and length) scaled by $1/\kappa$, the doping concentration would scale as κ, the voltage, current, capacitance, and switching delay would scale as $1/\kappa$, resulting in the power dissipation of the circuit scaling as $1/\kappa^2$, with the power density (power

per unit area) remaining constant. In other words, circuits would get smaller, faster, allowing more complicated circuits. When coupled with Moore's law, it meant that computers would get faster and more complex, but the total power consumed would remain approximately constant. This led to a so-called golden era of performance scaling, providing cheaper, faster, and more power-efficient transistors operating on ever-decreasing voltages.

Dennard scaling broke down in the early to mid 2000s. As transistors became even smaller, leakage and static power consumption could no longer be approximated away, leading to increased static power consumption, increasing the power density. Increasing temperatures negatively impact the ability of transistors to switch off; this coupled with increasing power densities threatens thermal runaway effects in integrated circuits. As a result, while we could still add more transistors on a given area as the transistor area shrank, keeping these transistors cool meant that they could no longer operate at ever-increasing clock frequencies dictated by Dennard's scaling equations.

The breakdown of Dennard scaling led to the development of multi-core processors. Out-of-order superscalar processors typically require N^2 resources for maintaining an instruction window of length N, but could no longer run at faster clock speeds due to power constraints. As a result, performance improvements had to be obtained from explicit parallelism, by running multiple program threads on different processor cores on the same die. Of course, Amdahl's law dictated that this mechanism could only improve performance so much over the parallel regions of code before the serial portion became a bottleneck. Hill and Marty [49] observed that multi-core was not the solution unless single-thread performance could still be improved, even at high cost. They also

advocated for the development of asymmetric multi-core processors, with some cores optimised for single-thread performance at high cost, and other cores optimised for accelerating the parallel region of code at low cost. In a retrospective paper, Hill and Marty [50] noted that their advocacy for both asymmetric multi-cores and for dynamically shifting resources between serial and parallel work at runtime were realised as ARM's big.LITTLE product line and Intel's Turbo mode. They also noted that their original paper underestimated the impact of the demise of Dennard scaling.

Esmaeilzadeh et al. [26] projected the performance of multi-core architectures to future technology generations. Their work predicted that merely scaling multi-cores would provide only limited speedups on many workloads, whether parallelism were increased to very high levels with programmer effort, or if parallelism were kept at existing levels and the system provided an almost unlimited power budget. As technology is scaled down and more cores added on a chip, increasing proportions of silicon area are underutilised, so called 'dark silicon'. It also indicated that microarchitecture innovation and specialisation were key to improving performance efficiently. This makes accelerators attractive, they can occupy the resources that would normally be underutilised anyway, and provide improvements in a direction orthogonal to performance. Hill and Marty [50] note that the ultimate success of accelerator-focussed chips will depend on programmability.

1.1.2 The Memory Wall

Dennard scaling resulted in both processor and DRAM frequencies rising exponentially, but processor frequencies grew at a faster rate than DRAM frequencies. Therefore, the gap between processor and DRAM frequencies grew exponentially. Wulf and McKee [132] noticed this phenomenon and coined the term 'memory wall' to describe it. They observed that while most processors used caches to mitigate the ever-increasing latency (in terms of processor cycles) of accessing DRAM memory, the cache miss rate would ultimately limit the performance of computers. The bandwidth from cache to DRAM is limited, their projections showed that even with a perfect cache, processor speed would be limited by the memory system.

This memory wall prompted a radical shift in the organisation of processors and memory. In the 1990s, a number of proposals were floated for integrating processors and memory onto the same die, thereby increasing the bandwidth to memory, which was predominantly decided by the number of pins available on a chip. Notable efforts in this direction included work by Kogge et al. [67], Gokhale et al. [41], Kang et al. [61], and Patterson et al. [99]. However, difficulties in integrating DRAM and logic silicon technologies on a single die along with the non-standard interfaces introduced as a result led to these efforts fizzling out [120].

Interest in tightly integrating processors and memory was revived in the early 2010s with the introduction of 3D stacking. This process allows different silicon dies to be stacked one over another and connected via through-silicon vias (TSVs). This allows potentially mixing different technologies to get maximum yield on the DRAM dies as well as the logic dies. Modern memory

systems such as Micron's hybrid memory cube (HMC) [53] already integrate some logic along with memory, although most of the logic is for an advanced memory controller and some self-test and array-repair capabilities along with basic atomic operations. This has led to a renewed interest amongst computer architects in proposing lightweight instructions that can be implemented on the logic layer of such a memory system.

Some emerging applications can avoid the memory wall to some degree. For example, CNNs reuse weights across an entire image. More images can be batched together to further increase reuse of data. The computation can be structured as a matrix-matrix multiplication for efficiency. Additionally, the data itself is often small enough to fit within a large on-chip SRAM which can provide more bandwidth than off-chip DRAM. As a result, accelerators for DNNs, particularly CNNs have very high compute to bandwidth ratios. Arithmetic intensity is often used as a metric to quantify the ratio of compute operations performed for every byte of data moved. A ratio of one operation for every byte moved has been historically considered a good ratio. Systems designed for CNNs, however, are optimised for very high arithmetic intensity. Notably, the Nvidia Tesla V100 GPU with its tensor cores provides a peak throughput of 125 TFLOPS, with a memory bandwidth of $900\,\mathrm{GB\,s^{-1}}$ [94], requiring an arithmetic intensity of at least 139 operations for every byte of data moved in order to achieve its peak performance without being limited by memory bandwidth. Accelerating applications with low arithmetic intensity is tricky due to the memory wall. Notably, BP on PGMs is a streaming application with very long data reuse distances. As a result BP will not benefit from caching or any other technique to improve data reuse.

1.2 Thesis Outline and Contributions

This thesis is organised in the following manner. Chapter 2 describes the workloads considered in this thesis, namely BP on PGMs, and DNNs including CNNs, MLPs, and RNNs. Chapter 3 describes an effort to accelerate one form of BP, sequential tree-reweighted belief propagation (TRW-S), through the use of a software technique, and the implementation of this technique on an existing FPGA TRW-S implementation. It also describes the challenges involved in the process and the changes required to the FPGA accelerator design in order to incorporate the software modification. Chapter 4 provides an overview of the proposed system, VIP (Versatile Inference Processor), describing how its ISA was developed to be a good fit on the workloads considered. It also describes how existing vector ISAs fall short of being able to efficiently express the operations involved in these workloads, notably BP on PGMs. Chapter 5 describes VIP's microarchitecture, and the modifications to a HMC based memory system required to achieve high performance. Chapter 6 describes how software is parallelised and vectorised to run on VIP, exploiting both coarse and fine grained parallelism in the workloads. Chapter 7 presents an evaluation of VIP using both a microarchitecture simulator as well as results from the synthesis of a VIP PE in 28 nm TSMC technology. Chapter 8 discusses a section of related work in processing in memory (PIM), near-memory processing, vector execution, and other proposed accelerators. Chapter 9 presents opportunities for future research in this area.

The main contributions of this thesis are as follows:

1. It presents an analysis of the workloads considered, including BP on PGMs, DNNs including CNNs, MLPs, and RNNs. It shows the similarities and

differences between these workloads, and presents a case for a single architecture to accelerate these workloads instead of the present trend of developing highly-specialised accelerators for each workload.

2. It presents the architecture of a system, VIP, that is capable of accelerating the workloads considered. VIP is developed around the well-understood paradigms of MIMD and SIMD execution, and near-memory processing. It presents novel instructions in the ISA that can efficiently describe the computation patterns involved.

3. It describes how software can be written in parallel fashion on VIP. Through detailed, microarchitecture simulations, it shows that VIP can achieve very close to its peak design performance on the workloads considered. It also presents the area and power requirements of VIP, and shows that these are modest.

1.3 Collaboration, Publications, and Funding

The work presented in this thesis consists of two parts. The first part (presented at the 22nd International Conference on Field-programmable Logic and Applications [52]) was done as a collaboration between Dr. Eriko Nurvitadhi, Dr. Jungwook Choi, Dr. Rob Rutenbar, my adviser (Dr. José F. Marínez), and myself. The initial idea for the development of hierarchical TRW-S was developed between Eriko and myself, we turned to Jungwook and Rob for their expertise on developing a FPGA prototype and evaluating it against their prior work [19, 20]. The second part, VIP, was developed by myself in collaboration with my adviser. We discussed VIP's development and evolution with with Dr. Avinash

Sodani, Dr. Ulf Hanebutte, Dr. Jim Ballingall, and others at Cavium.

This thesis was supported in part by the Intel Science and Technology Center for Embedded Computing; by AFOSR grant FA9550-15-1-0311; and by a contract with Cavium.

CHAPTER 2

BACKGROUND

2.1 Probabilistic Graphical Models

Probabilistic graphical models (PGMs) are a way of representing probabilistic relationships between variables using graphs. Given a graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$, the set of vertices $\mathcal{V}$ represents the variables under consideration, and the edges $\mathcal{E}$ represent the conditional or joint probability distribution between these variables. Probabilistic graphical models may use directed or undirected graphs, edges in directed graphs represent conditional probability distributions while edges in undirected graphs represent joint probability distributions [69]. In this thesis, I will primarily consider undirected PGMs, however, the analysis for directed PGMs is similar.

2.1.1 Message Passing and Variable Elimination

Consider the PGM shown in fig. 2.1. Let potentials $\phi_{i,j}(x_i, x_j)$ be defined between nodes x_i and x_j, where the potentials are proportional to the joint probability distributions, $\phi_{i,j}(x_i, x_j) \propto P(x_i, x_j)$. The joint probability of the variables may be defined as

$$P(x_1, \ldots, x_N) = \frac{1}{Z} \prod_{i=1}^{N-1} \phi_{i,i+1}(x_i, x_{i+1}) \tag{2.1}$$

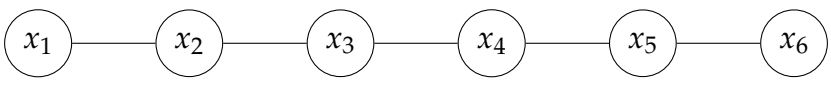


Figure 2.1: An example PGM

where Z is a normalization factor.

The probability of any one variable, x_i, may be obtained by marginalising or summing over all other variables.

$$\begin{aligned}
P(x_i) &= \sum_{x_j|j\neq i} P(x_1,\ldots,x_N) \\
&= \sum_{x_j|j\neq i} \frac{1}{Z} \prod_{k=1}^{N-1} \phi_{k,k+1}(x_k, x_{k+1}) \\
&= \frac{1}{Z} \sum_{x_j|j\neq i} \prod_{k=1}^{N-1} \phi_{k,k+1}(x_k, x_{k+1}) \\
&= \frac{1}{Z} \sum_{x_j|j\in[1,i-1]} \prod_{k=1}^{i-1} \phi_{k,k+1}(x_k, x_{k+1}) \sum_{x_j|j\in[i+1,N]} \prod_{k=i}^{N-1} \phi_{k,k+1}(x_k, x_{k+1}) \\
&= \frac{1}{Z} \sum_{x_{i-1}} \phi_{i-1,i}(x_{i-1}, x_i) \sum_{x_{i-2}} \phi_{i-2,i-1}(x_{i-2}, x_{i-1}) \ldots \sum_{x_1} \phi_{1,2}(x_1, x_2) \\
&\quad \sum_{x_{i+1}} \phi_{i,i+1}(x_i, x_{i+1}) \sum_{x_{i+2}} \phi_{i+1,i+2}(x_{i+1}, x_{i+2}) \ldots \sum_{x_N} \phi_{N-1,N}(x_{N-1}, x_N) \\
&= \frac{1}{Z} \sum_{x_{i-1}} \phi_{i-1,i}(x_{i-1}, x_i) \delta_{i-2\to i-1}(x_{i-1}) \sum_{x_{i+1}} \phi_{i,i+1}(x_i, x_{i+1}) \delta_{i+2\to i+1}(x_{i+1})
\end{aligned} \tag{2.2}$$

where

$$\delta_{i-1\to i}(x_i) = \sum_{x_{i-1}} \phi_{i-1,i}(x_{i-1}, x_i) \delta_{i-2\to i-1}(x_{i-1}) \tag{2.3}$$

$$\delta_{i+1\to i}(x_i) = \sum_{x_{i+1}} \phi_{i,i+1}(x_i, x_{i+1}) \delta_{i+2\to i+1}(x_{i+1}) \tag{2.4}$$

are messages sent from vertices $i-1$ to i and from $i+1$ to i respectively. Messages from a node contain the probability information of all nodes beyond that node.

In a similar way, consider a graph that may be decomposed as a tree of cliques. Assume that potentials ψ_C are associated with clique C. A clique C

is a set of vertices $\{x_i\}$ such that every pair of distinct vertices is connected by a single edge. It can be shown that the sum-product message passing algorithm described in eq. (2.2) may be generalised to messages passed between cliques [69]. Let $S_{i,j} = C_i \cap C_j$ be defined as the set of vertices common to cliques C_i and C_j, and let $\mathcal{N}(C_i)$ refer to the indices of cliques that are neighbours of C_i. The sum-product message passing computation for cliques may now be written as

$$\delta_{i \to j} = \sum_{C_i \setminus S_{i,j}} \psi_{C_i} \prod_{k \in \mathcal{N}(i) \setminus \{j\}} \delta_{k \to i} \tag{2.5}$$

This is to say that in order to compute a message to clique C_j, clique C_i first receives messages from all cliques except C_j, multiplies them with its own clique potential ψ_{C_i}, and sums out all variables that are not shared with clique C_j. In a given clique tree with a specified root, messages are sent from the leaves of the clique tree to the root, and then from the root to the leaves. Pearl [100] showed that the messages generated would converge after these two phases. At the end of these two phases of message passing, each clique can compute its own belief by multiplying all incoming messages with its own initial potential.

$$\beta_{C_i} = \psi_{C_i} \prod_{k \in \mathcal{N}(i)} \delta_{k \to i} \tag{2.6}$$

As the sum-product message passing belief propagation (BP) algorithm multiplies probability distributions, repeated multiplication may quickly cause arithmetic underflow with limited machine precision. To prevent underflow, messages are stored as negative logarithms of probability. Multiplications in the log-domain are additions, however the sum operation used in marginalising variables require exponentiation. A trick used here is to subtract the largest value from the factors before exponentiation, perform the marginalisation, go back to the log-domain and add back the largest value that was subtracted.

Mathematically

$$-\log(\sum_{x_i}\psi(X)) = -\log\left(\sum_{x_i} e^{-(-\log(\psi(X))-c)}\right) + c \tag{2.7}$$

where

$$c = \max\{-\log(\psi(X))\}$$

The sum-product message passing algorithm generates beliefs that are proportional to probability distributions over variables corresponding to the cliques in the clique-tree. Sometimes, however, we require the configuration of variables that maximises the maximum a posteriori probability (MAP). In this scenario, a variation known as the max-product or min-sum belief propagation algorithm is used. In the max-product algorithm, the message update is similar to the one in eq. (2.5), with the summation replaced by a maximum operation, i.e.

$$\delta_{i\to j} = \max_{C_i\setminus S_{i,j}} \psi_{C_i} \prod_{k\in\mathcal{N}(i)\setminus\{j\}} \delta_{k\to i} \tag{2.8}$$

As messages and factors are stored as negative logarithms for numerical stability, the max-product algorithm changes to the min-sum algorithm.

$$\begin{aligned}
-\log\left(\delta_{i\to j}\right) &= -\log\left(\max_{C_i\setminus S_{i,j}} \psi_{C_i} \prod_{k\in\mathcal{N}(i)\setminus\{j\}} \delta_{k\to i}\right) \\
&= \min_{C_i\setminus S_{i,j}}\left\{-\log\left(\psi_{C_i} \prod_{k\in\mathcal{N}(i)\setminus\{j\}} \delta_{k\to i}\right)\right\} \\
&= \min_{C_i\setminus S_{i,j}}\left\{-\log\left(\psi_{C_i}\right) + \sum_{k\in\mathcal{N}(i)\setminus\{j\}} -\log\left(\delta_{k\to i}\right)\right\}
\end{aligned} \tag{2.9}$$

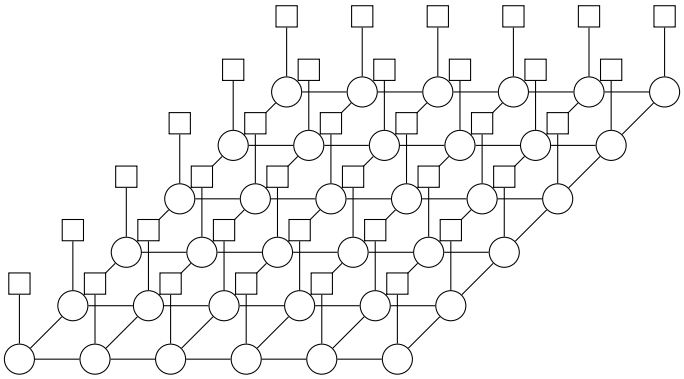


Figure 2.2: An illustration of a Markov random field used in early vision

2.1.2 Belief Propagation in Early Vision

BP is used in a class of computer vision applications (referred to as early vision applications because the computation is performed on data received directly from the sensor without much processing), such as image de-noising, depth from stereo, and image segmentation. These applications manifest as labelling tasks, i.e. the goal is to assign a label (e.g.: original value of pixel after noise is removed, depth or distance of object from camera, object id) to every pixel in the image. These applications use a graph that is a 2D grid, each vertex corresponding to an image pixel, and edges between neighbouring pixels indicating that neighbouring pixels are expected to be correlated. Proxies corresponding to negative logarithms of probability (called costs) are associated with every vertex and edge in this graph. (Figure 2.2 shows an example of a 2D grid used in early vision. The square-shaped nodes represent observed evidence, which is a noisy estimator for the hidden state represented by the circle-shaped nodes.) Vertex costs (messages from the square-shaped vertices in fig. 2.2) are chosen to be loosely related to the negative logarithm of probability of labels being assigned to each pixel. Edge costs (for edges between the circle-shaped nodes in fig. 2.2) are typically chosen to be symmetric, truncated quantities, proportional

to either the L0, L1, or L2 norm of the difference of the labels assigned to each adjacent pixel [33].

The Markov random fields (MRFs) used in early vision tasks are 2D grid graphs initialised with vertex θ_v and edge $\theta_{v,w}$ costs. BP on these graphs works in the following way – each vertex receives messages from its neighbours, and generates messages to its neighbours

$$m_{v\to w}(l_v) = \min_{l_w} \left\{ \theta_{v,w}(l_v, l_w) + \theta_v(l_w) + \sum_{x\in\mathcal{N}(v)\setminus w} m_{x\to v}(l_w) \right\} \tag{2.10}$$

When the messages thus generated converge, each vertex computes its most favourable label

$$l_v = \arg\min_{l_w} \left\{ \theta_v(l_w) + \sum_{x\in\mathcal{N}(v)} m_{x\to v}(l_w) \right\} \tag{2.11}$$

The order in which messages are updated affects the rate of convergence of the algorithm. Popular methods include the accelerated BP by Tappen and Freeman [119] (called BP-M in later works), sequential tree-reweighted belief propagation (TRW-S) by Kolmogorov [70], and a version of BP that schedules message updates in a manner similar to TRW-S (referred to as BP-S). All these algorithms are examples of asynchronous BP algorithms, i.e. they use new message values as soon as they are generated. BP-M works by scheduling message updates along rows or columns in a sequential manner, such that each message updated uses the previous message as an input. When all the messages in a particular direction have been updated, the direction of message updates is changed and the process repeated. TRW-S and BP-S work by updating messages in a different manner. Messages in these algorithms are updated along the diagonal. First, messages are updated in the right and down directions in the 2D grid, then the direction of message updates is changed to update the up

```
1 for y in range(IMG_Y):
2   for x in range(IMG_X):
3     for w in range(NUM_LABELS):
4       temp[w] = data_cost[x][y][w] + msg_right[x][y][w] +
           ↪ msg_up[x][y][w] + msg_down[x][y][w]
5     for v in range(NUM_LABELS):
6       msg_right[x+1][y][v] = MAX_INT
7       for w in range(NUM_LABELS):
8         msg_right[x+1][y][v] = min(msg_right[x+1][y][v],
             ↪ temp[w] + smooth_cost[v][w])
```

Figure 2.3: Example code fragment for BP-M in one direction

and left messages. Szeliski et al. [118] present a comparison of the convergence of multiple BP schedules, including BP-M, BP-S, and TRW-S. While all these algorithms have large amounts of parallelism (across rows and columns for BP-M, and along the diagonal for TRW-S and BP-S), BP-M is the easiest to parallelise as it requires the least amount of synchronisation. Figure 2.3 shows example code for BP-M in one direction. We see that while BP-M imposes a sequential order to the loop on line 2, line 1 may be executed in parallel on independent processors with no communication required between processors. TRW-S and BP-S both require very fine synchronisation between different processors, which makes them amenable to hardware consisting of multiple processors operating in lock-step.

2.2 Deep Neural Networks

Deep neural networks (DNNs) are a recent development in neural networks that include more layers than conventional neural networks. Advances in computing as well as in training methods such as dropout [115] and residual networks [47] have made training large networks feasible. Two types of DNNs,

namely convolutional neural networks (CNNs) and multi-layer perceptrons (MLPs), are of particular interest to the computer architecture community; these have very different computation and memory bandwidth requirements. CNNs have high arithmetic intensity, while MLPs have lower arithmetic intensity. Increased computational throughput provided by GPUs has been responsible for the wide adoption of these DNNs, however, their use in computer vision tasks with real-time constraints motivates the development of low-power accelerators that can meet these constraints.

2.2.1 Convolutional Neural Networks

CNNs were initially proposed by Solla and Lecun [113] as a way of generalising the learning in neural networks for image recognition tasks. CNNs use the convolution operator from classical computer vision as a means of reducing the number of weights in a neural network, while also achieving translational symmetry. CNNs consist of a number of convolutional layers. Each convolutional layer consists of a number of filters. Each convolutional filter is applied across the input features to produce a single channel in the output features.

The primary operation in CNNs is the convolution operation, with an added bias term.

$$O(x,y,z) = f\left(b(z) + \sum_i \sum_j \sum_k I(x-i, y-j, k) h^z(i,j,k)\right) \tag{2.12}$$

where O is output feature map, I is the input feature map, h^z is the z^{th} convolution filter, and b is the bias term. The convolution operation is embarrassingly parallel in the x, y, and filter (z) dimensions. Figure 2.4 shows an example code fragment for the convolution operation in CNNs. We see that the code con-

```
1  for y in range(IMG_Y):
2    for x in range(IMG_X):
3      for z in range(OUTPUT_CHANNELS):
4        out[x][y][z] = 0;
5        for i in range(KERNEL_X):
6          for j in range(KERNEL_Y):
7            for k in range(INPUT_CHANNELS):
8              out[x][y][z] += weight[z][i][j][k] *
                   ↪ in_[x-i][y-j][k]
9        out[x][y][z] += bias[z]
10       out[x][y][z] = max(out[x][y][z], 0)
```

Figure 2.4: Example code fragment for CNNs

sists of six nested loops, and these loops may be reordered in order to make the algorithm most amenable to the data layout, or vice versa. Of the six nested loops, three loops, on lines 1 to 3 are embarrassingly parallel. As the same filters are applied across the entire input feature map, CNNs have very good data reuse and are predominantly compute limited. Smaller networks such as AlexNet [74] require more storage for weights than for input features because input features in such networks are typically small, however, the arithmetic intensity of such networks may be improved through batching, i.e. processing multiple independent input images at the same time.

Another operation involved in CNNs is the pooling operation, which collects values from a neighbourhood and combines these values through the maximum or average operator. This reduces the size of feature maps. The final operation in CNNs (which also appears in MLPs) is the activation operation, wherein a non-linear function, e.g.: hyperbolic tangent, sigmoid, or rectified linear unit (ReLU), is applied to all the values produced at the output of a layer. Recent networks such as AlexNet [74], VGG-16 [112], ResNet [47], and Inception [117] all use the ReLU activation function, $f(x) = \max\{x, 0\}$.

2.2.2 Multi-layer Perceptrons

As the name suggests, MLPs, or fully-connected layers consist of multiple layers of neurons where every neuron produces an output as a weighted sum of inputs. A layer in an MLP can be summarized as

$$O(i) = f\left(b(i) + \sum_j W(i,j)I(j)\right) \tag{2.13}$$

where O is the output vector, W is the weight matrix, I is the input, and b is the bias term. An activation function is applied to the outputs of each layer, similar to CNNs. VGG-16 [112] uses ReLU activation functions for its fully connected layers.

Equation (2.13) involves two operations, a matrix-vector multiplication between the weights and inputs, and a vector-vector addition to add biases. As MLPs consist of matrix-vector multiplications, they are dominated by memory bandwidth requirements. For instance, the first fully connected layer in VGG-16 takes in 25088 inputs and produces 4096 outputs. With 16 bit datatypes, this requires 196 MiB of data and 0.1×10^9 multiply accumulates (MACs).

2.2.3 Recurrent Neural Networks

Recurrent neural networks (RNNs), and their subset, long short-term memory (LSTM) networks, are neural networks where the output of the network at a particular time step is provided as an input to the network at the next time step. This allows the network to store information over many time steps, introducing the concept of memory. RNNs are used in a number of applications, notably with speech and language processing, such as machine translation, captioning

images, speech to text, etc.

A popular variant of LSTM networks was introduced by Gers et al. [40], which processes data in the following manner

$$f_t = \sigma\left(W_f[c_{t-1}, y_{t-1}, x_t] + b_f\right) \tag{2.14a}$$

$$i_t = \sigma\left(W_i[c_{t-1}, y_{t-1}, x_t] + b_i\right) \tag{2.14b}$$

$$\tilde{c}_t = \tanh\left(W_{\tilde{c}}[y_{t-1}, x_t] + b_{\tilde{c}}\right) \tag{2.14c}$$

$$c_t = i_t \odot \tilde{c}_t + f_t \odot c_{t-1} \tag{2.14d}$$

$$o_t = \sigma\left(W_o[c_t, y_{t-1}, x_t] + b_o\right) \tag{2.14e}$$

$$y_t = o_t \odot \tanh(c_t) \tag{2.14f}$$

The network has a state c_t that evolves over time. It takes in input x_t, and produces output y_t. Within the LSTM cells, a neural layer f_t updates the state of the cells to adjust whether they remember or forget their state, another layer i_t updates the state of the cells given the present inputs and the outputs from the previous step.

CHAPTER 3

CASE STUDY: FPGA IMPLEMENTATION OF HIERARCHICAL TRW-S

Belief propagation (BP) methods work well in practice, but have their drawbacks – messages may oscillate and not converge, and the large number of iterations required to reach convergence, particularly with synchronous BP algorithms. Tree-reweighted belief propagation (TRW) introduced by Wainwright et al. [125], and its sequential version, sequential tree-reweighted belief propagation (TRW-S) introduced by Kolmogorov [70] provide a concave lower bound on the log likelihood. These methods work by decomposing a graph with loops into a combination of trees, and performing modified BP on the decomposition. A comparative study of energy-minimization techniques by Szeliski et al. [118] showed that the TRW-S algorithm produces one of the best results for a depth-from-stereo task.

The work described in this chapter speeds up the convergence of the TRW-S algorithm by using a hierarchical approach, similar to the one proposed by Felzenszwalb and Huttenlocher [33] for BP. Further, the hierarchical algorithm is generalised to the case when discontinuity (smoothness) costs are varied based on a contrast-based scaling factor, as is common in early vision applications such as a depth-from-stereo task. Work described in this chapter was done in collaboration with Jungwook Choi, Eriko Nurvitadhi, and Rob Rutenbar, and was presented in the 25th International Conference on Field-programmable Logic and Applications (FPL 2015) [52]. The initial idea for hierarchical TRW-S was developed by Eriko Nurvitadhi and myself, scaling contrast-based smoothness costs was developed in conjunction with Jungwook Choi, and the FPGA implementation was developed entirely by Jungwook Choi. This chapter discusses

the FPGA implementation only for the purpose of providing context.

3.1 Description of the Hierarchical TRW-S Algorithm

A major drawback of BP based methods is the large number of iterations required to reach convergence, especially for synchronous BP algorithms. The hierarchical BP algorithm proposed by Felzenszwalb and Huttenlocher [33] reduces the total number of iterations required. Hierarchical BP works in a similar way to image pyramids in computer vision, it uses multiple levels, grouping together $\epsilon \times \epsilon$ vertices from a lower level of the graph into a single vertex on the upper level. BP is performed on the highest level in this hierarchy, and message values are copied over to its lower level to serve as a starting point for the BP algorithm on this level. This considerably reduces the number of message update iterations required to get the message values close to convergence. Additionally, the overheads due to the hierarchy form a convergent geometric series, so the overhead is bounded.

Our proposed hierarchical TRW-S algorithm works in a way very similar to hierarchical BP, grouping $\epsilon \times \epsilon$ vertices at the lower level into a single vertex at the upper level. Reference software provided by Szeliski et al. [118] uses a contrast-based scaling factor for discontinuity (smoothness) costs in order to improve the performance of the depth-from-stereo algorithm. These factors manifest as multiplicative penalties $w_{v,w}$ applied to a global discontinuity cost function θ_V, as shown in eq. (3.1a). Felzenszwalb and Huttenlocher [33] describe a method to scale a fixed discontinuity cost function between levels as in eq. (3.1b), their method does not take into account these multiplicative penalties. We modify their method to incorporate these penalties as shown in eq. (3.1c) (n_v

and n_w) refer to nodes on the upper level of the hierarchy. In the case that we don't use this contrast-based scaling factor, i.e. $w_{v,w} = 1 \quad \forall (v,w) \in \mathcal{E}$, our method reduces to the one proposed by Felzenszwalb and Huttenlocher. It is important to note that these methods for scaling the discontinuity costs work only in the case of a finite-difference model of discontinuity costs, i.e. the costs are proportional to the difference of labels assigned to neighbouring vertices, and that the costs are zero when adjacent vertices are assigned the same label. The reference depth-from-stereo software uses at most two values for these multiplicative penalties, but our method of scaling these penalties in eq. (3.1c) will result in more values with additional levels. Quantising these penalties to one of the two original values does not affect convergence significantly.

$$\theta^L_{v,w}(l_v, l_w) = w^L_{v,w} \times \theta^L_V(l_v - l_w) \tag{3.1a}$$

$$\theta^{L+1}_V(l_{n_v} - l_{n_w}) = \epsilon \times \theta^L_V\left(\frac{l_v - l_w}{\epsilon}\right) \tag{3.1b}$$

$$w^{L+1}_{n_v,n_w} = \frac{1}{\epsilon} \times \sum_{\{v,w \in \mathcal{E}: v \in n_v, w \in n_w\}} w^L_{v,w} \tag{3.1c}$$

As the approach described by Kolmogorov [70] does not depend upon the initial value of messages, we can use the message values obtained from hierarchical TRW-S to serve as a starting point or guess. This will not interfere with the convergence properties of TRW-S, rather will expedite its convergence on the original problem.

3.2 Parameter Exploration for Hierarchical TRW-S

We apply the hierarchical TRW-S algorithm as described in section 3.1 to a number of stereo vision benchmarks – the Tsukuba Head and Lamp dataset from the University of Tsukuba, and Venus [106], Cones, and Teddy [107] from Middlebury College. We run a software implementation of this hierarchical TRW-S algorithm in order to explore the effects of various parameters that can be tuned in the hierarchical approach. As TRW-S is a sequential, asynchronous BP algorithm that converges a lot faster than synchronous BP, choosing a good set of parameters is critical to the success of a hardware implementation of hierarchical TRW-S.

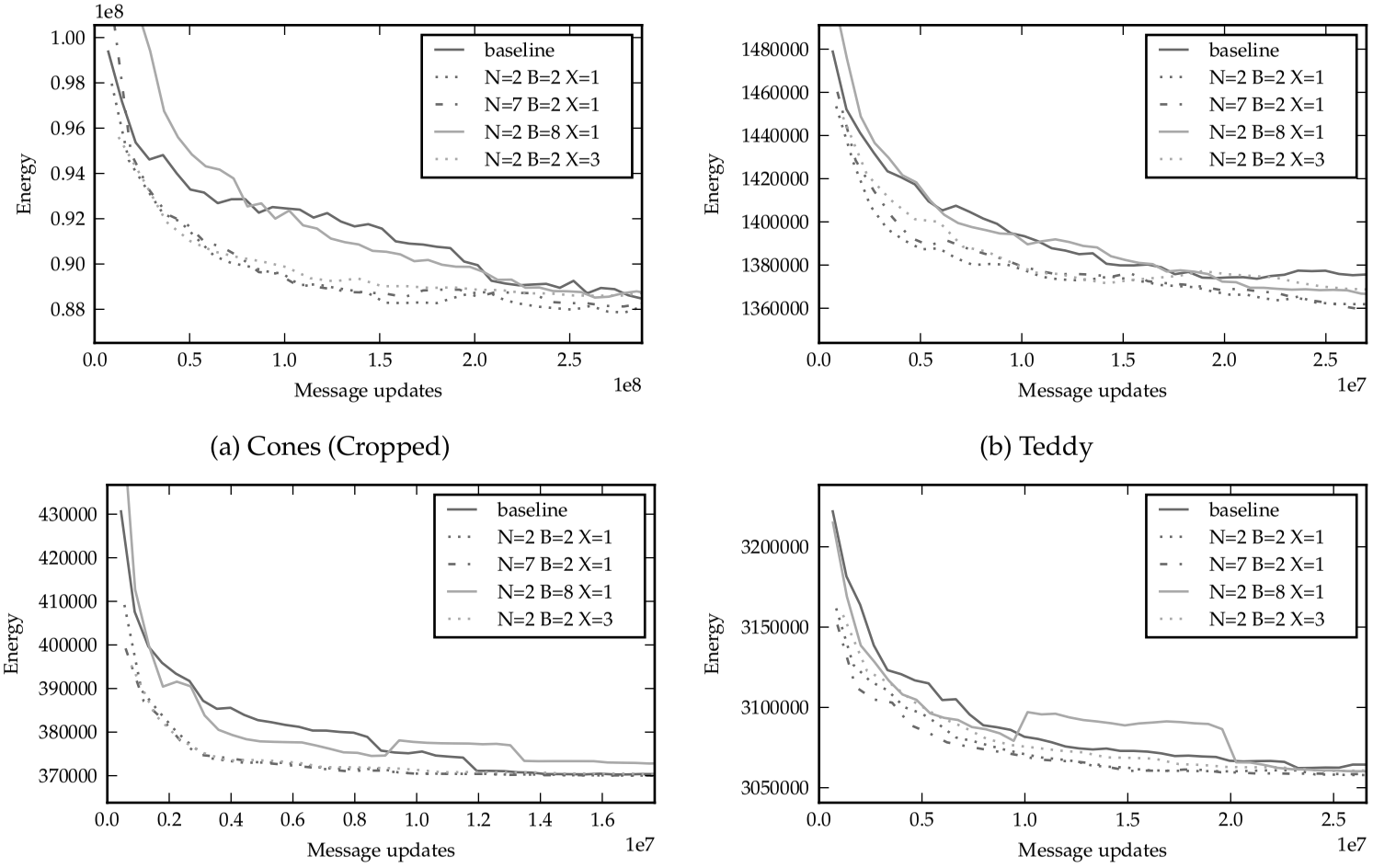
Two parameters that lend to manipulation are the number of levels in the hierarchy as well as the amount of computation performed at each level. Ideally, more levels would lead to faster convergence as adding a level accelerates convergence on the level immediately below it, at the cost of limited overhead. Performing more iterations on higher levels may also be profitable as these iterations require fewer message updates, and may lead to better starting points for lower levels. In order to study the effect of these parameters, we varied the number of levels from two to seven, and allowed each level to perform one, two, or three times the number of iterations performed on the level immediately below it. This gives us 18 different configurations.

Felzenszwalb and Huttenlocher [33] present a case for grouping a block of $\epsilon \times \epsilon$ vertices in a level into a single vertex on the upper level, but limit their evaluation to only the case when $\epsilon = 2$. In order to investigate the effect of increasing the size of blocks used to generate the hierarchy, we evaluated the

convergence rate of the hierarchical TRW-S algorithm when ϵ was set to be two, four, or eight. This, combined with the 18 configurations obtained by varying the number of levels and the computation performed at each level results in a total of 54 different configurations.

Figure 3.1 shows the rate of convergence of the hierarchical TRW-S algorithm under a representative subset of these 54 configurations. From these results, we notice some surprising results. As (asynchronous) TRW-S converges significantly faster than synchronous BP, we notice that the improvements in the performance of TRW-S are not sufficient to offset the overheads of adding more levels in the hierarchy or performing more iterations on upper levels. Further, we observe that larger block sizes used to create the hierarchy actually reduce the rate of convergence of the algorithm, sometimes to worse than the baseline (flat) TRW-S.

In order to study the effect of block sizes further, we created synthetic stereo pairs consisting of a white square on a red background. We studied two particular cases, one when the squares were aligned with an 8×8 grid, and one where they were not aligned with a 2×2 grid. Figure 3.2 shows the difference between messages at the end of the TRW-S algorithm on the upper level (end of the tenth iteration) and the beginning of the last iteration of TRW-S on the lower level (end of the ninth iteration) as L1 norms using both 2×2 and 8×8 blocks. Large differences in the message values indicate that the hierarchy failed to provide a good starting point for TRW-S on the lower level. If object boundaries are not aligned with the blocks used to create the hierarchy, the contrast-based smoothness cost scaling factors on the higher levels fail to represent this information. Consequently, messages generated on the higher levels of the hierarchy also fail



(a) Cones (Cropped)

(b) Teddy

(c) Tsukuba

(d) Venus

Figure 3.1: Convergence rates of a software Hierarchical TRW-S implementation compared with baseline (flat) TRW-S, shown as energy vs. message update plots for four Middlebury benchmarks. The number of hierarchical levels (N), block size (B), and additional iterations done over each hierarchical level (X) are varied. We observe that the increased computation overhead of more than two levels or performing additional work on higher levels offsets any benefits from these configurations, just as larger block sizes result in a poorer rate of convergence.

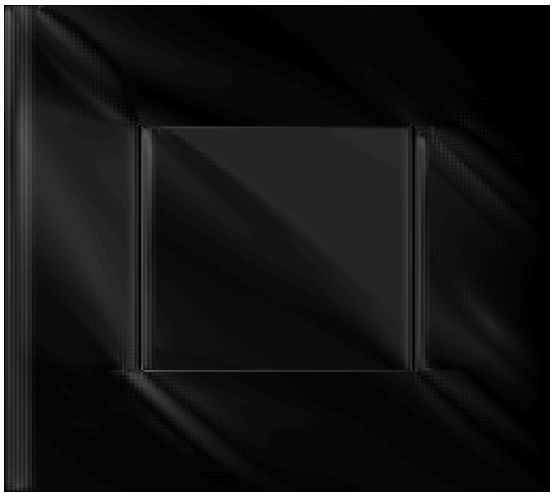
to encode this information on object boundaries, serving as a poor starting point for the original (lowest) level. Smaller block sizes have a higher probability of being aligned with object boundaries, thereby encoding this information, thus perform better. As a result, we set the block size to 2×2 for remaining experiments.

3.3 Hardware Implementation of Hierarchical TRW-S

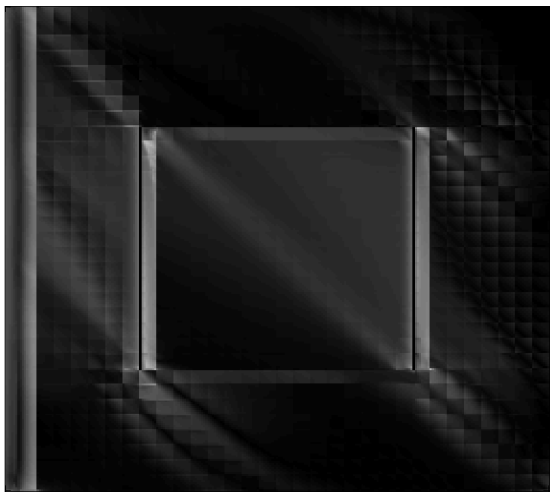
Choi and Rutenbar [19, 20] developed a streaming TRW-S implementation on a Convey HC-1 CPU-FPGA hybrid platform. The Convey HC-1 is equipped with an Intel Xeon dual-core processor and four Xilinx Virtex-5 (V5LX330) FPGAs tightly coupled via cache-coherent virtual memory. As the basic TRW-S algorithm is the same for both levels of the hierarchy, we reuse their implementation of streaming TRW-S and add small modifications to support constructing the hierarchy and copying messages across levels.

There are two possible approaches to implementing the modifications necessary for hierarchical TRW-S.

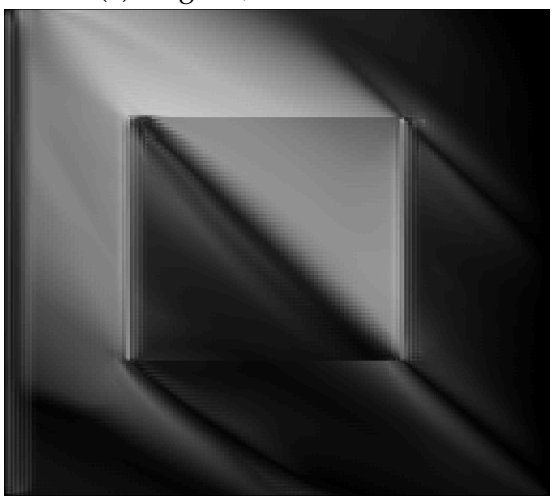
1. **Naive** The naive approach would be to leave the FPGA design as is, and use the CPU to construct the hierarchy and copy data across levels. This has the shortest implementation time. However, we observe (table 3.1) that the CPU takes very long for these operations, obliterating any advantages in convergence time arising from the hierarchy.
2. **Optimised** Due to the failure of the naive approach, we move the operations for constructing the hierarchy and copying messages across levels to



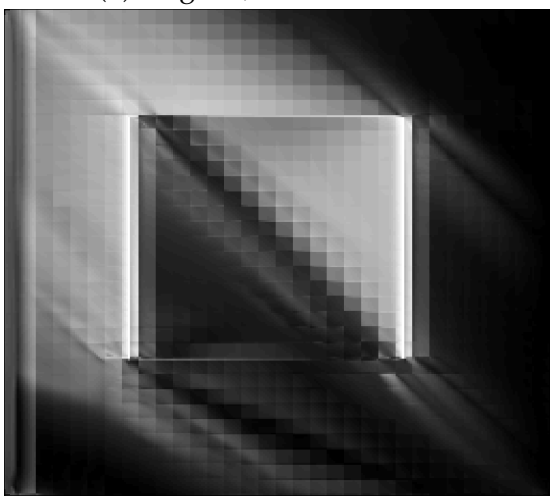
(a) Aligned, 2×2 block size



(b) Aligned, 8×8 block size



(c) Not Aligned, 2×2 block size



(d) Not Aligned, 8×8 block size

Figure 3.2: Heat maps showing the L1 norm between messages (black indicates very little or no difference, while white indicates large differences) at the 9^{th} iteration on the lower level and the 10^{th} iteration on the higher level. Lower differences indicate that the hierarchy provides a good starting point for the TRW-S algorithm. Two cases are shown – when the objects are aligned with the blocks used to create the hierarchy, and where they are not. If the objects are aligned with the blocks used to create the hierarchy, the hierarchical approach serves as a good starting point (lower differences in messages), while the other case results in a poorer starting point (larger differences between messages) irrespective of the block size used.

Table 3.1: Comparison of times spent on various tasks for baseline (flat) TRW-S and hierarchical TRW-S ($N = 2, X = 1$) algorithms for the Tsukuba head and lamp stereo pair benchmark. Times reported are for 29 iterations for baseline TRW-S and 12 for hierarchical TRW-S, as they result in nearly identical energy values (around 100.5% of the lower bound). STRM-TRW-S times for hierarchical TRW-S include time spent in both levels of the hierarchy.

Task	**Time Required (s)**		
	Baseline	**Hierarchical TRW-S**	
		Naive	**Optimised**
Construct	N/A	0.015	0.001
Copy	N/A	0.027	0.001
STRM-TRW-S	0.093	0.048	0.048
Total	**0.093**	**0.090**	**0.050**

the FPGA. In this approach, the CPU is responsible only for handling inputs and outputs and for coordinating different phases of the hierarchical TRW-S algorithm. Using the FPGA to handle these operations results in minimal overheads.

The overall architecture of hierarchical TRW-S on the Convey HC-1 is shown in fig. 3.3. The STRM-TRW-S hardware is unchanged from work by Choi and Rutenbar [19, 20], but two additional units, Construct and Copy, have been added. These units share memory resources with STRM-TRW-S. Data costs and messages are packed so that data costs are accessed using memory ports 0–3, and messages using memory ports 4–15. Each vector can be at most 16 elements, and each element requires 16 bits. If more labels (larger vectors) are required, this may be achieved by accessing data through multiple cycles, the STRM-TRW-S hardware uses a folded-pipeline architecture that allows it to process depth-from-stereo with up to 64 labels.

Choi and Rutenbar [19, 20] parallelise the TRW-S algorithm along the diag-

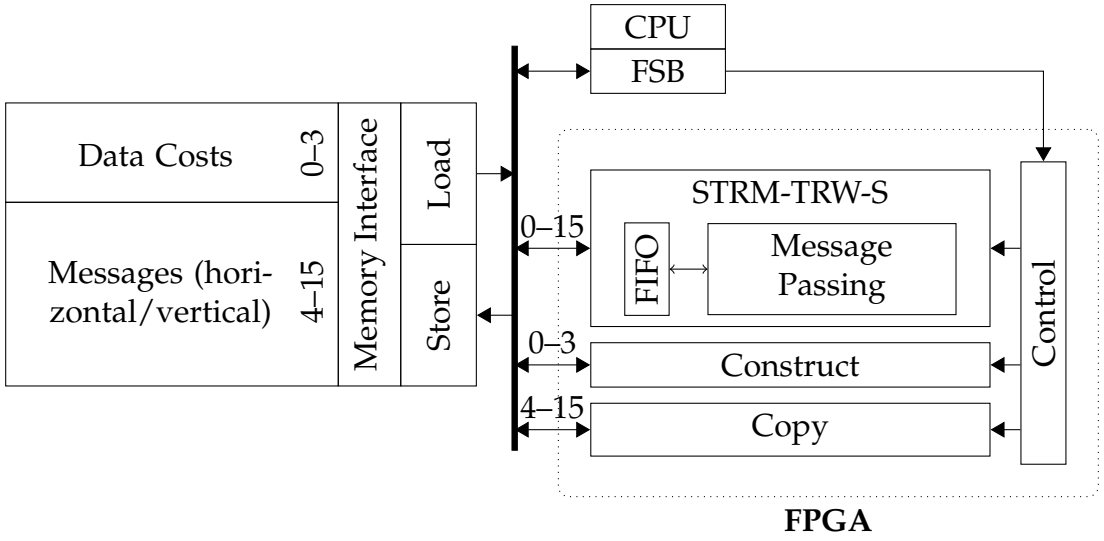


Figure 3.3: Architecture of the hierarchical TRW-S FPGA implementation. Memory ports 0–3 are used to transfer data costs, while memory ports 4–15 are used for messages for up to 16 labels in every cycle. The streaming TRW-S design by Choi and Rutenbar [19, 20] is unchanged, two additional units, Construct and Copy, are added. As Construct operates only on data costs, it uses memory ports 0–3, while Copy operates only on messages, so uses memory ports 4–15.

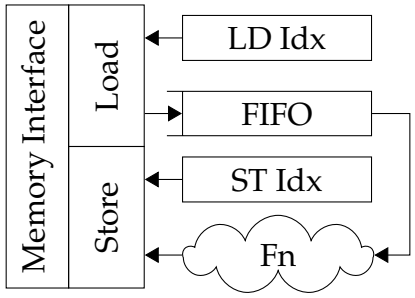


Figure 3.4: Architecture of the Construct and Copy units in the hierarchical TRW-S FPGA implementation. Both units consist of index control logic, a FIFO buffer, and functional logic. The index control logic is responsible for computing addresses for moving data between the different hierarchical levels. The functional logic for Construct consists of arithmetic units for computing data costs and multiplicative penalties for the smoothness cost, while the functional logic for Copy is a simple buffer.

Table 3.2: Device utilization summary for hierarchical TRW-S on Xilinx Virtex-5 (V5LX330) FPGA

Resources	STRM-TRW-S	Construct	Copy	Convey infrastructure	Total
Slice Registers (FF)	25,745	920	1,027	53,531	**81,223**
Slice LUTs (6 input)	24,967	1,362	698	58,916	**86,125**
Block RAM (36 kbit)	132	8	22	75	**237**

onal. Their STRM-TRW-S hardware therefore requires that the data be stored in a diagonal order. Construct and Copy, on the other hand, group data within 2×2 blocks, therefore a data layout in 2×2 blocks would be preferred. On a traditional memory system, these conflicting requirements would have resulted in some type of performance trade-off. The Convey HC-1, however, has scatter-gather DIMMs, which provide 8-byte access to any location in physical memory with minimal overhead, thereby reducing the inefficiencies associated with the non-sequential accesses by the Construct and Copy modules. This is reflected as a very high bandwidth utilisation by all three modules, Construct, Copy, and STRM-TRW-S.

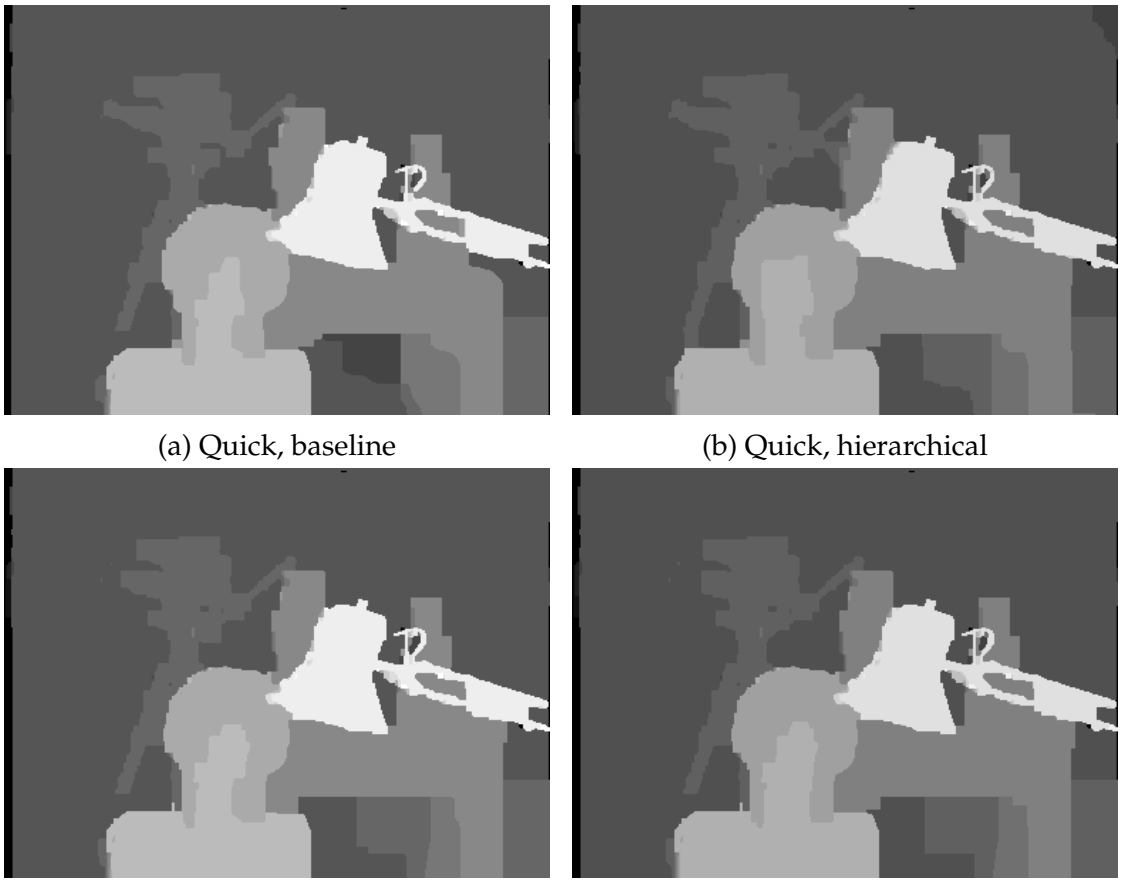
Figure 3.4 shows the architecture of the Construct and Copy units. Both units have index control logic for generating addresses in the upper and lower levels of the hierarchy, cognisant of the diagonal data layout. Construct's functional unit consists of arithmetic for accumulating data costs as well as the multiplicative penalties, while Copy's functional unit is a simple buffer. Table 3.2 shows the FPGA resources required by the various units. We see that the overheads due to Construct and Copy are small.

Table 3.3: A comparison between baseline TRW-S and hierarchical TRW-S ($N = 2$, $X = 1$) for the Middlebury stereo benchmark images. The times reported discount the initialization time, as it is not part of the MRF inference problem. The times reported for baseline include the time spent in STRM-TRW-S, while times reported for hierarchical TRW-S include the times spent in STRM-TRW-S for both levels, along with times required for Construct and Copy operations.

Benchmark	**Convergence**	**Runtime (s)**	
		Baseline	**Hierarchical**
Cones (Cropped)	Quick	2.529	1.313 (-48%)
	Better	11.080	5.185 (-53%)
Teddy	Quick	0.233	0.159 (-32%)
	Better	1.436	1.135 (-21%)
Tsukuba	Quick	0.022	0.014 (-38%)
	Better	0.093	0.050 (-46%)
Venus	Quick	0.115	0.069 (-40%)
	Better	0.336	0.199 (-41%)

3.4 Results

Table 3.3 shows the runtime of both baseline as well as hierarchical TRW-S for the four benchmarks under consideration. Two sets of results are shown. The first is when a quick, rough estimate is sufficient, in which case we run approximately ten baseline iterations. The second case is when we want results as close to convergence as possible, in which case we allow the energy to reach sufficiently close to its lower bound. Figure 3.5 shows the resulting disparity maps for the Tsukuba Head and Lamp dataset for the two cases. The hierarchical method speeds up the rate of convergence, but has its overheads. In the case of quick inference, these overheads must be amortised over a smaller number of iterations, thus contribute to a greater fraction of the runtime of the algorithm. In the second case, when energy is allowed to reach fairly close to the lower bound, the hierarchical method itself requires a larger number of iterations to reach the lower energy, so the amortised overheads are offset by diminishing re-



(a) Quick, baseline (b) Quick, hierarchical

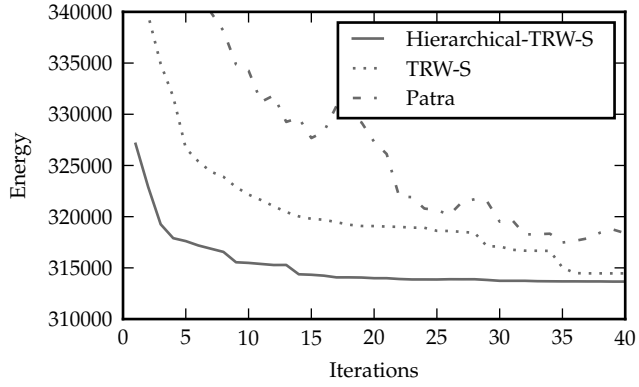
(c) Better, baseline (d) Better, hierarchical

Figure 3.5: A comparison of resulting disparity maps for baseline and hierarchical TRW-S for the Tsukuba dataset. Two cases are shown, one when quick, approximate inference is needed, and two when energy is allowed to reach sufficiently close to the bound.

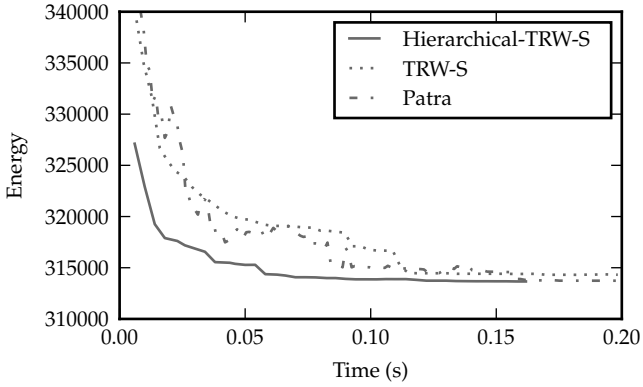
turns. We observe in table 3.3 that the optimised hierarchical TRW-S algorithm offers a 21 % to 53 % reduction in runtime over baseline TRW-S.

3.5 Discussion

The work presented in this chapter shows an FPGA implementation of hierarchical TRW-S. It shows that changing the underlying algorithm or software can improve the performance of an existing implementation with only modest



(a) Energy vs Iterations



(b) Energy vs Time

Figure 3.6: A comparison of the rates of convergence of Hierarchical TRW-S, baseline TRW-S, and Patra for the Tsukuba benchmark

overheads. There were also some important lessons learnt during this exercise.

A fast-converging algorithm is critical. Elidan et al. [24] showed that an asynchronous BP algorithm converges at least as fast as a synchronous BP algorithm. Zhao et al. [139] present Patra, a system that uses synchronous TRW message passing instead of the asynchronous TRW-S. By using a synchronous algorithm, they are able to significantly increase parallelism leading to much lower per-iteration runtime. However, as we see in fig. 3.6a, Patra requires many more iterations to reach the same energy as TRW-S. Even when we factor in the reduced per-iteration time of Patra (fig. 3.6b), we find that Patra performs only

comparably to baseline TRW-S, hierarchical TRW-S is still faster. This is with a system that has twice the peak DRAM bandwidth as the Convey HC-1 system used in our experiments. These results reinforce the need for a fast-converging asynchronous BP algorithm with sufficient parallelism.

Memory bandwidth is critical to the efficiency of Markov random field (MRF) inference. The Convey HC-1 system used had a peak memory bandwidth of $19.2\,\mathrm{GB\,s^{-1}}$, and the highest memory bandwidth utilisation was for the Tsukuba benchmark, at 92 %. Teddy and Venus did not perform as well because the STRM-TRW-S hardware by Choi and Rutenbar [19, 20] was designed for problems where the number of labels was a multiple of 16, while Teddy and Venus used 20 and 60 labels respectively, thereby wasting 37 % and 6 % of memory bandwidth respectively. Without the Convey's scatter-gather DIMMs, performance of the Construct and Copy operations required for hierarchical TRW-S would have been even lower.

Even if we could utilise all the available memory bandwidth on a conventional memory system efficiently, we would still be bottlenecked by the bandwidth. Consider the case of Cones, an image with 1800×1000 pixels (the actual image available via Scharstein and Szeliski [107] is larger in height at 1500 pixels, the reason for cropping this image is discussed in a later paragraph). Even quick hierarchical inference on this benchmark requires over a second for each frame, unacceptable for a real-time system. In order to achieve real-time performance of 25–30 fps, the memory bandwidth must increase by 25–30 times, impossible with a conventional memory system. Therefore, a number of related work try and reduce the amount of external memory bandwidth, for example, Liang et al. [77] break up the MRF into tiles and discard all messages within

a tile, storing messages only at tile boundaries. Through this reduced storage, they are able to store all messages within on-chip SRAM. Their work recomputes the messages within a tile from stored messages at the tile boundary by performing multiple BP iterations within the tile. Not only is this method approximate, the additional work required to recompute messages that were discarded from earlier iterations means that the number of effective iterations performed by this design is very low, as little as one effective iteration. Lin et al. [78] use the BRAMs within the FPGA to provide high-bandwidth data access to their processing engines. This method, however, cannot scale up to larger problems as the BRAMs are not as dense as DRAM. It also requires that the graph be loop-free, which disqualifies this system from working on loopy BP used in early vision systems. However, as discussed in section 5.2 such high bandwidth may be achieved with modern 3D-stacked memory systems such as Micron's hybrid memory cube (HMC) [53], or high-bandwidth memory (HBM) [1].

Other limitations of this work include that the system was designed to work with only min-sum message passing on a 2D grid using truncated finite-difference smoothness costs. As a result, the system was unable to handle smoothness costs that were not of the form $\theta_V(l_v, l_w) = f(\min\{|l_v - l_w|, 3\})$. Additionally, STRM-TRW-S used a FIFO within the FPGA to store messages from one diagonal to the next. The capacity of this FIFO limited the images processed to be no larger than 1000 pixels on their smallest side. As a result, we had to crop Cones, a stereo benchmark that was originally 1800×1500 in size. A truly programmable system for BP on MRFs should not make assumptions that limit its effectiveness.

CHAPTER 4

VIP: OVERVIEW AND ISA

The previous chapter discussed the design of an FPGA accelerator for one particular algorithm, sequential tree-reweighted belief propagation (TRW-S). Even a minor modification to this algorithm required a redesign for the accelerator, naively offloading the modifications to a CPU resulted in an unacceptable overhead. This chapter discusses the design of a system that is sufficiently programmable to work with a number of algorithms, including inference on probabilistic graphical models (PGMs), and deep neural networks (DNNs) such as convolutional neural networks (CNNs), multi-layer perceptrons (MLPs), and recurrent neural networks (RNNs), without requiring redesign, and has sufficient computational throughput and memory bandwidth to support these class of algorithms.

A system designed for fast inference in PGMs, CNNs, MLPs, and RNNs must have sufficient parallelism in order to achieve the required computational throughput. Additionally, the system must also have a high amount of memory bandwidth available so that the computational units are not starved. As we will see in section 7.2, the arithmetic intensity of the workloads considered in this work, especially belief propagation (BP) is low, typically around four operations for every byte of data moved.

There are three paradigms that could be leveraged to achieve the compute and memory bandwidth requirements of these algorithms. We could either use a number of independent processors in a MIMD paradigm, use a GPU with the SIMT paradigm, or use a vector processor with a SIMD paradigm.

Ahn et al. [4] propose Tesseract, a system that uses multiple (32) independent ARM Cortex-A5 in-order processors within a 3D stacked memory system, Micron's hybrid memory cube (HMC). While this system can indeed supply the required memory bandwidth, its computational capability is still very low. CPUs require a number of instructions for computing addresses, moving data between memory and registers, and executing loops, so the number of instructions that perform useful work is typically low. Even if we assume that each processor can work with a 2 GHz clock and process one useful operation every cycle (that is to say, we assume that all overheads disappear), Tesseract will still be able to process only 64 GOP/s, considerably lower than the 1 TOP/s required.

GPUs manage to provide both the required computational throughput as well as the memory bandwidth. However, they are typically designed so as to have significantly higher computational throughput than needed for applications with low arithmetic intensity, consequently, they are inefficient with respect to area and power. The SIMT processing paradigm used in GPUs also requires each thread to compute addresses, move data between memory and registers, execute loops. This adds instructions, saturating the instruction issue bandwidth.

A vector processing paradigm, on the other hand, is able to amortise the instruction issue bandwidth as each instruction operates over a number of data elements. In fact, a vector processing paradigm is well suited for the workloads considered in this work. Ni and Jain [90] present four categories of vector operations – $f_1 : V \to V$, $f_2 : V \to S$, $f_3 : V \times V \to V$, and $f_4 : V \times S \to V$, where V and S denote sets of vector and scalar operands respectively. I introduce some

shorthand notation for composition of operations from these categories in order to provide a basis for an ISA. $f_5 : V \times V \to V \to S$ is a composition of an f_3 operation followed by an f_2 operation. If we loop over an f_5 operation while keeping one of the vector operands constant, we create the category $f_6 : M \times V \to V$, where M represents the set of matrix operands. A vector dot-product is an example of an f_5 category operation, while a matrix-vector product is an example of the f_6 category. Operations involved in PGMs, CNNs, MLPs, and RNNs fall under the f_1, f_3, f_4, and f_6 categories. For example, eq. (2.10) may be written as

$$\hat{\theta_v}(l_w) = \theta_v(l_w) + \sum_{x \in \mathcal{N}(v) \setminus w} m_{x \to v}(l_w) \tag{4.1a}$$

$$m_{v \to w}(l_v) = \min_{l_w} \left\{ \theta_{\{v,w\}}(l_v, l_w) + \hat{\theta_v}(l_w) \right\} \tag{4.1b}$$

Here, eq. (4.1a) is an example of a f_3 operation, while eq. (4.1b) is a f_6 operation. Similarly, eq. (2.12) may be written as

$$R(x, i, y, j, z) = \sum_k I(x - i, y - j, k) h^z(i, j, k) \tag{4.2a}$$

$$Q(x, i, y, z) = \sum_j R(x, i, y, j, z) \tag{4.2b}$$

$$P(x, y, z) = \sum_i Q(x, i, y, z) \tag{4.2c}$$

$$O(x, y, z) = b(z) + P(x, y, z) \tag{4.2d}$$

Here, eq. (4.2a) is an f_5 operation, eqs. (4.2b) and (4.2c) are f_2 operations, and eq. (4.2d) is an f_3 operation. Sections 2.2.2 and 2.2.3 discussed how the operations in MLPs and RNNs are essentially dense matrix-vector multiplications from the f_6 category.

```
1 for i in range(n):
2     out[:] = op(out[:], vec[:][i])
```

(a) Multiple independent vector reductions

```
1 while(n/2 > 0):
2     vec[0:n/2-1] = op(vec[0:n/2-1], vec[n/2:n-1])
3     n /= 2
```

(b) Divide-and-conquer on a single vector

Figure 4.1: Two approaches to performing vector reductions on traditional vector processors

Traditional vector processors are efficient at operations in categories f_1, f_3, and f_4, but not at reduction operations at the core of operations in the f_2, f_5, and f_6 categories. Reductions on traditional vector processors may be achieved in one of two ways, either *a*) perform independent vector reductions in each vector element (fig. 4.1a), or *b*) perform reductions in a divide-and-conquer manner, dividing the input vector into half every loop iteration and performing element-by-element operations using the two halves as inputs until only a single vector element remains (fig. 4.1b). Both these methods have inefficiencies. Each instruction does less work than a single 'vector-reduce' instruction would, requiring an increase in the number of instructions fetched and issued for the same task. The explicit loop nature means that the program must be constantly engaged computing elements of a vector. Instead a single vector-reduce instruction allows the program execution to move ahead and set up the next outer loop iteration.

The Active Memory Cube [88], as a traditional vector processor, consists of 32 vector processors on the logic layer of a HMC. Each vector processor consists of four vector lanes, and each lane has a vector register file containing sixteen 256 B registers. The vector lanes are programmed through a VLIW program-

ming model, operating them in lock-step. Figure 4.2 shows how code for the min-sum BP kernel may be vectorised to run on such a system. (Here, we assume that the length of the vectors is shorter than the vector register length, negating the need to stripmine vector loops.) There are some interesting observations regarding vectorising this code. First, we observe that this code uses the first type of vector reduction, performing multiple vector reductions in different elements, as shown on line 9. In order to do this, it needs to address as many vectors as the columns of the smoothness cost matrix, in this case, the same as the number of labels. As each vector lane has access to only 16 vector registers, the Active Memory Cube will quickly run out of vector register names, requiring that either the smoothness cost matrix be streamed in from memory or vectors be packed and unpacked into the limited number of registers, decreasing performance. Even if we utilise all four lanes for message updates, we will have access to only 64 vector registers, which limits the versatility of this system as size of the problem (in terms of number of labels) increases. The vector registers are also underutilised – while the vector registers are 256 B each, vectors are only 32 B long for 16-label BP with 16 bit fixed point data, resulting in seven-eights of a vector register remaining unused if it stores a single (unpacked) vector. While line 3 can compute the sum of many messages in a single vector, line 8 serialises the computation of each outgoing message. Second, while we can compute the sum of various incoming messages in a vectorised format (line 3), the direction of vectorisation changes in the next loop (line 8), which means that the vector data must be moved to scalar registers in order to perform the category f_4 operation required.

On the other hand, the ARM scalable vector extensions (SVE) [9] ISA provides horizontal vector reduction operations (category f_2). Figure 4.3 shows how code

```
1  # Vectorise and eliminate variable v
2  for v in range(NUM_LABELS):
3    temp1[v] = msg_in1[v] + msg_in2[v] + msg_in3[v] + msg_in4[v]
4    msg_out[v] = MAX_INT
5  for w in range(NUM_LABELS):
6    # Vectorise and eliminate variable v
7    for v in range(NUM_LABELS):
8      temp2[v] = temp1[w] + smooth_cost[v][w]
9      msg_out[v] = min(msg_out[v], temp2[v])
```

Figure 4.2: Example code showing the vector execution of BP kernel on IBM's Active Memory Cube [88]. Here, we assume that `NUM_LABELS` is less than the vector register length.

```
1  for v in range(0, NUM_LABELS, VL):
2    # Vectorise and eliminate variable i
3    for i in range(VL):
4      temp1[v+i] = msg_in1[v+i] + msg_in2[v+i] + msg_in3[v+i] +
         ↪ msg_in4[v+i]
5      msg_out[v+i] = MAX_INT
6  for w in range(NUM_LABELS):
7    for v in range(0, NUM_LABELS, VL):
8      # Vectorise and eliminate variable i
9      for i in range(VL):
10       temp2[i] = temp1[v+i] + smooth_cost[w][v+i]
11       msg_out[w] = min(msg_out[w], temp2[i])
```

Figure 4.3: Example code showing the vector execution of BP kernel on ARM's SVE ISA [9]. The SVE ISA is hardware vector length agnostic, requiring all code be written with a stripmine loop to allow for different hardware vector lengths, `VL`.

may be vectorised for this ISA. SVE provides 32 vector registers, but the lengths of these vectors may range from 128 bit to 2048 bit. As a result, any vector code written for this ISA must contain a stripmine loop, as shown on lines 3 and 9. While SVE does provide more vector registers than IBM's Active Memory Cube, 32 instead of 16, it suffers from the same issues caused by a fixed number of vector register names. The stripmine loop necessitates that segments of the smooth-

ness cost matrix be streamed in from memory on every instance of line 10.

VIP recognises that this pattern of matrix-vector style operations (f_6 category) is common in the workloads to be accelerated, namely BP on PGMs and DNNs. It addresses the limitations of traditional vector architectures (such as the IBM Active Memory Cube and ARM SVE) by changing the processing paradigm from a vector-register paradigm (where data are stored in vector registers) to a vector memory-memory paradigm (data are stored in memory and accessed via pointers). This shift provides two key benefits. First, it makes a lot more efficient use of available storage space by preventing any loss due to misalignment with a hardware vector length. Second, it provides a very intuitive way to store and operate on matrices. In the VIP ISA, a matrix may be represented by just its starting address and dimensions, instead of packing and unpacking a matrix in multiple vector registers. A vector is represented by its starting address and dimension (which is the same as the number of matrix columns in the vector unit configuration registers). The ISA provides instructions to set the dimensions of the matrix in software, allowing VIP to operate on data where data size may change rapidly.

The vector memory-memory paradigm was used in older vector processors such as the CDC STAR-100 [108]; the vector units in these machines received operands and wrote data directly to magnetic core memory. A stream unit in the STAR-100 was responsible for supplying operands from and writing results back to storage. Modern DRAM, however, has unpredictable and long latency dependent on the state of the memory and pending requests. VIP provides a 4 KiB SRAM scratchpad memory that is managed in software. A programmer must stage data from DRAM memory into this scratchpad before the vec-

tor units in VIP can operate on data. Similar to a register file, this memory provides three read and two write ports, the difference is that it may be accessed at any byte location. The scratchpad ameliorates the issue of increased memory bandwidth with the vector memory-memory paradigm – classic vector memory-memory machines had to write all intermediate values back to memory, while vector-register machines could store these in registers and discard them when they were no longer needed. VIP requires that its vector units operate on scratchpad memory allowing temporary and intermediate values to be stored in the scratchpad, not in DRAM, reducing DRAM bandwidth.

In order to set up addresses and execute loops, VIP provides a scalar unit with a stripped down instruction set. It also provides a load-store unit to move data between local storage (SRAM scratchpad holding data for the vector unit, or register file holding data for the scalar unit) and DRAM main memory. The scalar and load-store units are meant to be operated in the shadow of the vector unit. Ideally, the programmer should set up a long latency vector operation; while the vector unit is busy, prefetch data for the next outer loop iteration by setting up the necessary addresses and issuing the required load or store commands. If done correctly and with adequate software prefetching, VIP's vector ALUs can be constantly kept busy, allowing VIP to achieve 100 % of its peak throughput. This is different from a GPU where each thread may have to perform redundant computation such as bumping induction variables, issuing load and store instructions, executing branches which take cycles away from performing useful work.

The workloads under consideration have a considerable amount of coarse-grained parallelism. For example, PGMs use graphs, and vertices of the graph

Table 4.1: A summary of the VIP instruction set

Vector Instructions	
Configuration	`set.{vl,mr},{v.drain}`
Matrix-vector	`m.v.{mul,add,sub,min,max.nop}.{add,min,max}`
Vector-vector	`v.v.{mul,add,sub,min,max}`
Vector-scalar	`v.s.{mul,add,sub,min,max}`
Scalar Instructions	
Reg-reg	`{add,sub,sll,srl,sra,and,or,xor}`
Reg-imm	`{add,sub,sll,srl,sra,and,or,xor}`
Move	`{mov,mov.imm}`
Control	`{blt,bge,beq,bne,jmp}`
Load-store Instructions	
SRAM	`{ld,st}.sram`
Reg	`{ld,st}.reg`
Sync	`memfence`

may be processed in parallel. Similarly, CNNs apply the same set of filters to different regions of the input image, these operations again may be done in parallel. Therefore, a system for inference on these workloads must also provide many independently controlled processing engines (PEs) that work in a MIMD paradigm, each PE internally using a SIMD paradigm for fine-grained parallelism. VIP provides multiple (128) independently controlled PEs. Each PE has a vector unit to operate on data, and a scalar unit to perform other operations such as setting up addresses and executing loops, along with a load-store unit for moving data between memory and local storage (registers or scratchpad). Table 4.1 shows a summary of the instructions supported by VIP in each of the three units.

The other component to VIP's design is to provide the necessary memory bandwidth. BP-M on full-HD video at 24 fps requires a memory bandwidth of $237\,\mathrm{GB\,s^{-1}}$, beyond the capability of a conventional memory system. VIP there-

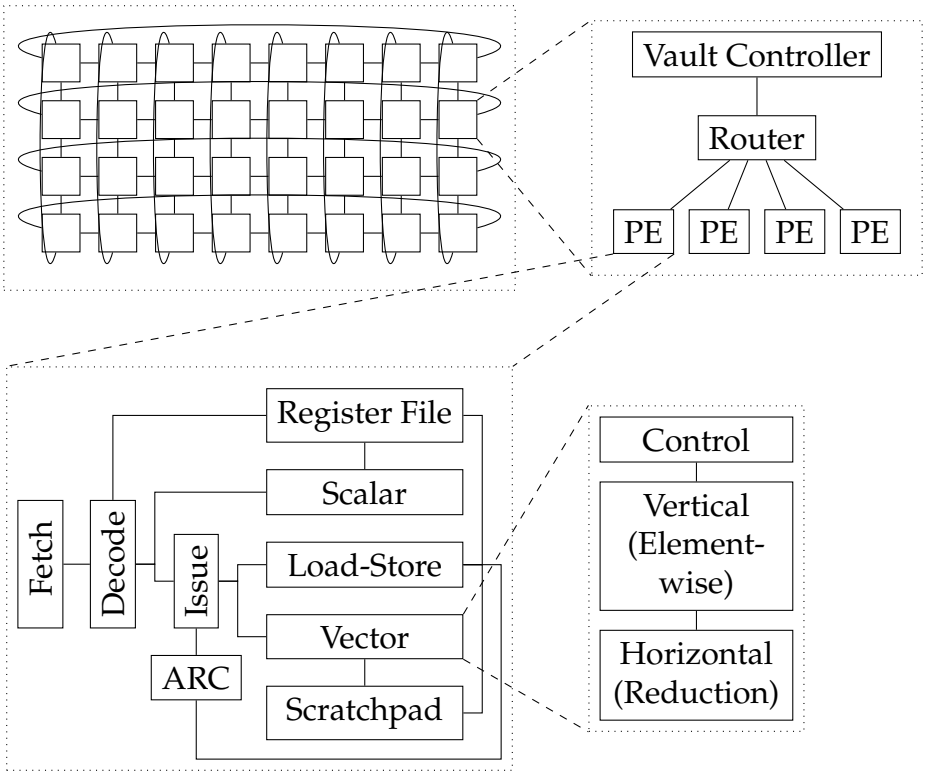


Figure 4.4: An overview of VIP's architecture, from the system level down to the data-path in a single PE. A hybrid memory cube (HMC) has 32 vaults, which are connected via a 2D torus network. A vault has four processing engines (PEs) associated with it. A PE consists of unified fetch and decode, followed by independent vector, scalar, and load-store units. A unified issue unit is responsible for detecting data hazards in the scratchpad (used by vector and load-store pipelines) through an associative array lookup (ARC). The vector unit itself consists of a vertical unit for element-wise operations, and a horizontal unit for reducing vectors to scalars. The vector pipeline has a 64 bit data-path which can perform one 64 bit, two 32 bit, four 16 bit, or eight 8 bit operations in a single cycle.

fore uses a modern, 3D stacked memory system similar to Micron's HMC [53] to provide the required memory bandwidth.

Figure 4.4 shows an overview of the VIP architecture.

CHAPTER 5

THE VIP HARDWARE

The previous chapter described an overview of the VIP ISA and philosophy. This chapter discusses the microarchitecture and memory system design decisions that allow VIP to achieve the required performance on the workloads considered in this work.

5.1 The VIP Microarchitecture

VIP uses a vector-processing paradigm to accelerate inference in probabilistic graphical models (PGMs), and deep neural networks (DNNs) including convolutional neural networks (CNNs), multi-layer perceptrons (MLPs), and recurrent neural networks (RNNs), as discussed in chapter 4. VIP's processing engines (PEs) are augmented with a vector reduction unit and with support for f_6 category (matrix-vector) operations. Unlike other processors developed for scientific computing that focus on only the multiply accumulate (MAC) operation, VIP's ISA allows any pair of operations to be composed between the vertical (element-wise) and horizontal (reduction, or across elements) vector units.

VIP's PEs therefore consist of a vector unit with a vertical or element-wise vector unit similar to traditional vector processors and a horizontal or reduction unit (shown in fig. 4.4). The reduction unit is bypassed for instructions that are not in the f_2, f_5, or f_6 categories. Operations from the f_2 category (vector reductions) can be accomplished by setting the number of matrix rows to one, and issuing a f_6 (matrix-vector) instruction with no operations for the ALUs in the element-wise vector unit.

There are trade-offs associated with choosing the data-path width of the vector pipeline. A wider data-path allows the PE to process more data in the same cycle, at the overheads of increased area. Additionally, the area and access time of the SRAM scratchpad used to supply data to the vector pipeline will also increase with an increased data-path width. If the data-path width is allowed to grow larger than the length of vectors that we expect will be processed, then ALUs are left idle without data to process, wasting area and power. In case of inference on Markov random fields (MRFs), we expect vector lengths to vary between 9 (for 3×3 optical flow) to 16 (depth-from-stereo), to 49 (7×7 optical flow). The example being considered in this work is depth-from-stereo with 16 labels. VIP employs a modest 64 bit wide data-path allowing the area to remain low. The data-path is capable of sub-word SIMD computation, so the data-path can perform one 64 bit, two 32 bit, four 16 bit, or eight 8 bit computations in a single cycle. The vectors encountered in real workloads, e.g.: belief propagation (BP) or DNNs will be longer than the data-path width. These vectors are sent down the vector pipeline in multiple cycles in the classic temporal vector processing paradigm employed by computers such as the CDC STAR-100 or the Cray 1 [108]. The choice of a relatively narrow data-path has yet another advantage: by increasing the latency of individual operations, it allows the latency of DRAM accesses and control code to be effectively hidden behind the longer computation time. While the VIP ISA does not specify a maximum vector length, RTL synthesis (described in section 7.1.3) allows a vector to contain up to 256 elements.

The VIP ISA uses a vector memory-memory paradigm for processing in order to better support matrix-vector operations, as well as to better support the applications considered. PGMs may result in varying matrix and vector sizes as

the algorithm traverses an arbitrary graph. CNNs also will require, for efficient computation, at least twice the number of vectors as the X-Y dimensions of the filters. Therefore, CNNs with large filters, such as AlexNet [74] with 11×11 filters will therefore require at least 242 vectors to be stored in order to process a single filter on an activation window. A vector-register architecture will restrict the number of vectors, e.g. the Active Memory Cube [88] provides 16 vector registers, ARM's scalable vector extensions (SVE) [9] provides 32. Limiting the number of vectors that may be addressed in local storage (registers) requires data be spilled to the memory hierarchy when the available vector names get used up, wasting performance, as discussed in chapter 4. While architectures such as Hwacha [75] and even the much older Fujitsu VP2000 [122] provide a way to vary the number of available vector register names by reducing the length of each vector, such configuration must be done before a kernel is run. This would have a detrimental effect on the performance of applications such as PGMs which may encounter different vector lengths and numbers from vertex to vertex. GPU kernels request the number of registers required, which determines the number of thread blocks that can be run simultaneously on a streaming multiprocessor (SM). An issue with this approach is that the performance of a GPU (the number of thread blocks that can be run on a SM) is limited by the maximum number of registers used in a thread block; this decreases performance if the number of registers requested by a thread block is not a perfect divisor of the total number of registers available (misalignment), and because the number of registers holding useful data (live registers) is typically lower than the maximum number of registers requested.

VIP provides a 4 KiB byte-addressable SRAM scratchpad memory for every PE with three read and two write ports. The scratchpad requires three read

ports, two for reading in vector operands, and one for reading data from the scratchpad and writing it to memory. Similarly, one write port is required for storing the results of the vector computation, another for reading data from memory. The scratchpad is made of eight SRAM banks, each with three 8 bit read and two 8 bit write ports. Logic similar to a barrel shifter is used to swizzle the order of these banks when the address requested is not aligned to a 8 B boundary, allowing any byte-aligned location to be read from or written to. Figure 5.1 shows the banked structure of VIP's scratchpad.

SRAMs generated by a memory compiler are read non-combinationally. First, the address to be read must be supplied to the read ports, then the output enable signal is pulled low. The data is available some time after the output enable is pulled low. This means that VIP must pipeline address computation and access into two separate cycles in order to access the scratchpad. According to CACTI 6.5 [85], the SRAM required to build the scratchpad has an access time of 0.19 ns, significantly lower than the 0.8 ns target clock. In fact, this access time is lower than half the clock period, 0.4 ns. VIP therefore pipelines the address generation and SRAM access into two halves of the clock cycles: address generation from either the vector control or load-store unit and address update by the swizzle logic within the scratchpad is done in the first half of the clock cycle, the SRAM output enable signal is provided by the falling clock edge, and the SRAMs are read and the output reassembled by the barrel shifter in the second half of the clock cycle to be latched at the input registers of the vector or load-store unit on the rising edge of the next clock.

In order to execute basic control flow operations such as loops and branches, and to compute DRAM addresses for moving data between memory and the

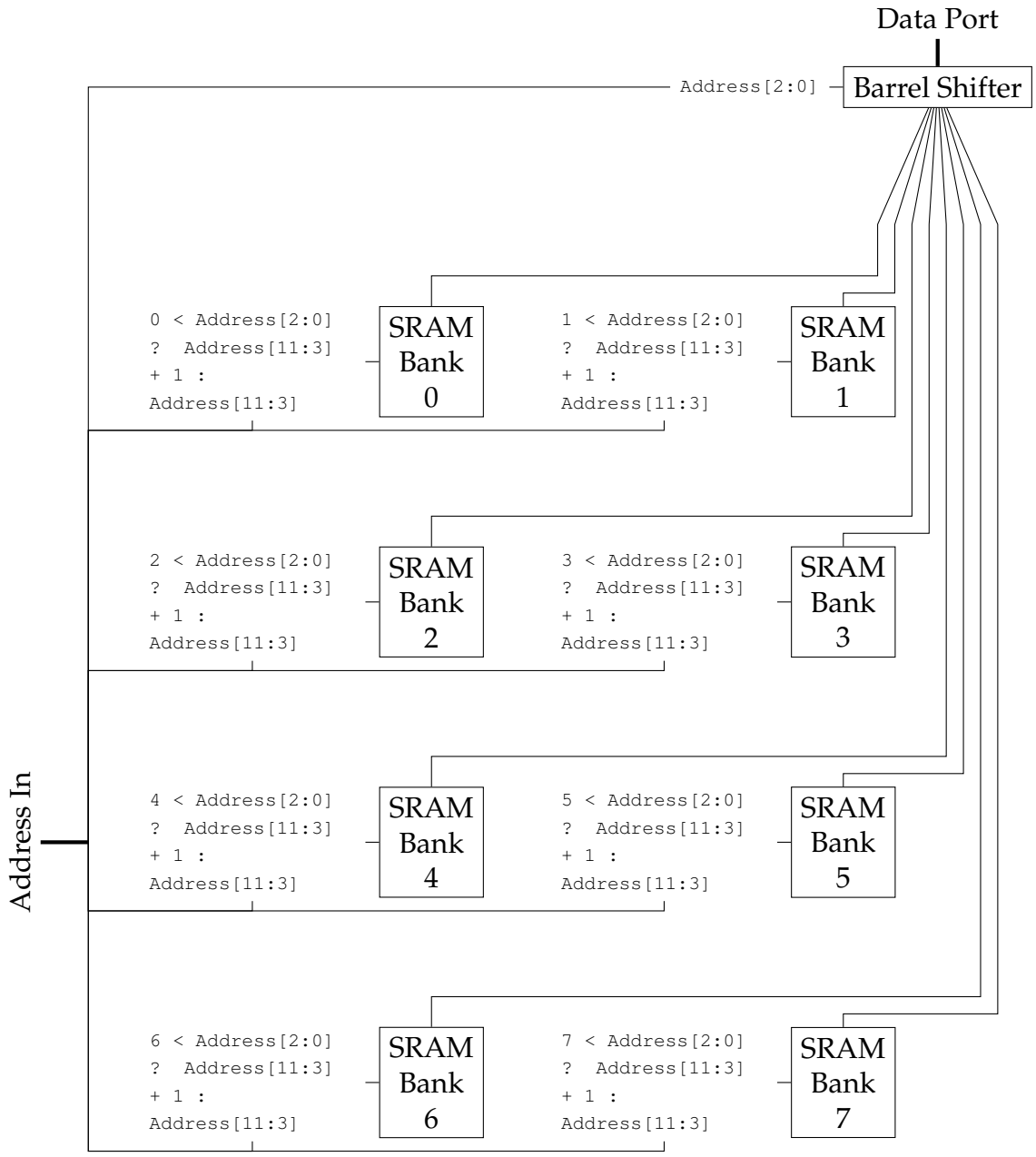


Figure 5.1: Banked Design of VIP's SRAM Scratchpad

scratchpad, VIP provides a scalar unit. The scalar pipeline supports only basic arithmetic, bitwise shift and logical, and conditional branch and unconditional jump instructions. The scalar unit has a 64-entry register file, which also holds pointers to data within the scratchpad. In order to mitigate hazards within the scalar pipeline, registers are augmented with a 'valid' bit. This bit is cleared whenever an instruction that writes to that register is issued. Instructions reading this register will stall until this valid bit is set. Similar to the scratchpad, the register file is also constructed using a SRAM; the address (register number) is generated by the decoder in the first half of the clock cycle, and the output enable asserted and data read out of the SRAM in the second half of the clock cycle. The valid bits are stored as a separate set of flip-flops; the SRAM is not read until the valid bits indicate that all data requested by the instruction is available to be read.

VIP has a load-store unit to move data between the PE and DRAM. This unit may load to or store from either a scalar register or the scratchpad. If loading to a register, the valid bit in the register is cleared until the data returns from DRAM, preventing invalid values from being read by subsequent instructions reading that register. Instructions that move data between the scratchpad and DRAM receive via registers, the start addresses in DRAM, in the scratchpad, and the length of data to be moved through scalar registers. This allows the program to work even if the length of vectors is unknown at compile time.

When an instruction is issued to load data into the scratchpad, the start and end addresses within the scratchpad are locked by writing into a data structure called array range check (ARC). The ARC is an associative array holding up to twenty entries. Any instruction reading or writing data to the scratchpad is

stalled if the range of addresses it accesses within the scratchpad overlap with a range stored within the ARC. An ARC entry is cleared upon the completion of the load associated with that range. Unlike a DMA system, this ARC-based mechanism is invisible to the programmer.

There are two choices around the use of the ARC. The ARC could be used as described, only for tracking when DRAM loads have completed, or to detect and mitigate hazards in the vector pipeline. The latter scenario has the advantage that the programmer is no longer responsible for mitigating hazards in VIP's vector pipeline, making the system easier to program. The drawback of this approach, however, is that it checks an associative array multiple times every cycle, resulting in many simultaneous comparison operations that require time and consume power. As a result VIP uses the ARC only to track DRAM loads. Another choice is the number of entries in the ARC. RTL synthesis results show that the size of the ARC cannot be increased beyond twenty entries without added complexity to meet the 0.8 ns clock cycle constraint.

The scalar and load-store units are designed to operate in the shadow of the vector pipeline, as discussed in chapter 4. An efficient program will issue a long-latency vector instruction (such as a matrix-vector instruction), and while the vector unit is processing this instruction, the scalar and load-store units prefetch the next set of data to be consumed by the vector unit. This process may be pipelined even deeper in software, if required.

VIP has a unified front end consisting of fetch and decode stages for all three – vector, scalar, and load-store – units. In order to keep complexity low, VIP uses a single-issue, in-order model for the front end. If an instruction is stalled in the front end, all subsequent instructions are stalled as well. As simulation results

in section 7.2 will show, VIP achieves very good performance through carefully scheduling instructions to minimise stalls, and software pipelining to ensure that data is almost always available when needed, mitigating the need for the added complexity of a multiple-issue or an out-of-order model. Instructions in VIP, however, complete out of order, as they take different amounts of time. As a result of this, and that it is difficult to checkpoint the state of the entire scratchpad memory without a severe penalty, VIP does not support precise exceptions. This is to trade off ease of programmability for improved performance.

VIP does not offer any mechanism to transfer data between the scalar and vector units. The rationale behind this decision is that these units will operate on different sets of data. The vector unit is meant to perform the actual data processing, while the scalar unit is meant for control operations like executing loops, traversing graphs, and to initiate data movement between the PE and DRAM memory. If the programmer wishes to transfer data between the scalar and vector units, they may do so by storing the data to memory and loading it again. Another way to transfer data between the scalar and vector units would be to add new a load-store instruction in VIP's ISA for performing the data movement, as the load-store unit can read and write from and to both the scalar register file as well as the scratchpad used by the vector unit. As none of the benchmarks considered required such an instruction, it is left out of the instruction set.

VIP's PEs are clocked at 0.8 ns, or 1.25 GHz. This assumption of clock speed is validated through synthesis of an RTL model of a VIP PE in a 28 nm library provided by ARM. Given that the target of achieving 24 fps inference over the benchmarks considered requires ∼800–900 GOP/s, and that prior work [16, 19,

20] has established that 16 bits of precision are sufficient, VIP requires $\sim$ 128 PE to provide the necessary computational throughput.

Software for VIP may be written in a way so as to ensure that PEs access their local vaults most of the time, with very little memory traffic across vaults. Chapter 6 describes how this is achieved. As a result, VIP can utilise a relatively simple network model, a 2D torus. The network supports a maximum bandwidth of $10\,\mathrm{GB\,s^{-1}}$ on each link, this ensures that the network link bandwidth is matched to the peak bandwidth from each hybrid memory cube (HMC) vault.

5.2 The VIP Memory System

The workloads considered require a memory system that has high amounts of memory-level parallelism, and can work efficiently with requests for short bursts of data. These characteristics are not typical to a traditional DRAM memory system, with few channels each consisting of a few ranks. Modern, 3D stacked memory like Micron's HMC [53] or JEDEC's high-bandwidth memory (HBM) standard [1] can provide these characteristics. Both these systems use multiple DRAM dies stacked one above the other, using through-silicon vias (TSVs) to connect the DRAM dies to a logic die or silicon interposer below.

VIP's memory is similar to an HMC rather than HBM because the HMC offers higher memory bandwidth and more memory parallelism, using 32 channels (called vaults) instead of 8 provided by HBM. VIP makes some modest but key modifications to the default HMC configuration in order to improve the performance even further. The memory system, similar to the HMC consists of 32 vertical partitions, or vaults, arranged in an 8×4 grid. Each vault contains 16

DRAM banks. Banks within a vault share data TSVs but have independent control TSVs, so each bank behaves like a rank. A bank contains 65,536 rows, each holding 256 bytes of data accessed as 32 byte columns (32 bit $\times$ $8n$ prefetch). Each vault has a data bandwidth of $10\,\mathrm{GB\,s^{-1}}$, for a total data bandwidth of $320\,\mathrm{GB\,s^{-1}}$ for the HMC.

VIP's 128 PEs are distributed among the 32 vaults, four PEs in each vault. The vaults are connected via a 2D torus network, and PEs within a vault along with the vault controller are connected in a star topology. The network links are bidirectional, 64 bit links in each direction. With a 1.25 GHz clock, this configuration provides a $10\,\mathrm{GB\,s^{-1}}$ bandwidth on each network link. The expectation is that software is written in a way that PEs communicate mostly with their local vaults, ensuring that the network does not become a bottleneck for most applications.

This work assumes that the VIP PEs are located within the HMC as this offers some advantages, notably in energy efficiency. Jeddeloh and Keeth [56] indicate that two-thirds of the energy per bit within the HMC is spent in the logic layers. Other works [5, 126] claim that the SERDES links within the HMC consume approximately a fifth of the energy per bit as the entire HMC, although work by Fukuda et al. [35] indicates that this energy may be only a tenth of the energy per bit of accessing the HMC. It is important to note, however, that this work does not assume any special capabilities arising from being located within the HMC, e.g. it does not assume that the bandwidth within the HMC is more than the bandwidth outside as is assumed in work such as Tesseract [4], nor does it assume any logic-in-memory requirements such as in Tetris [39]. If VIP is located outside the HMC, it will encounter some additional latency due to

the SERDES links; this latency, however, may be hidden through aggressive software pipelining, a technique already employed to hide the DRAM latency. There shall be a small penalty due to being unable to modify the default HMC behaviour, section 7.2.3 will discuss the impact of this decision.

The default HMC address mapping scheme interleaves vaults at the lower address bits. [53, Table 11] This exposes the most amount of memory parallelism by distributing a random sequence of request addresses across multiple vaults. The applications considered in this work, however, need to explicitly partition data across vaults so that PEs can access data in their local vaults as much as possible, without stressing the on-chip network. VIP, therefore, moves the vault address bits to the most significant bits of the address space. If VIP's PEs must be placed outside the 3D stack (accessed via SERDES links), the same effect can be achieved through a logical to physical address translation scheme. Unlike virtual memory, this scheme would be very simple as it simply involves shuffling some bits in hardware before issuing requests to memory.

The HMC uses a closed-page policy. This makes sense for workloads that access entire cache lines with a random sequence of memory addresses. The HMC uses DRAM pages containing just 256 B [56], significantly less than the 2 KiB to 4 KiB pages in traditional DRAM systems, reducing the overfetch (i.e. fetching more bits to the DRAM sense amplifiers than needed) problem. This decision helps when processors access an entire cache line of data at a time. VIP, however, forgoes caches as it can achieve good results without the overhead of caches. An open-page policy, therefore makes more sense in this scenario, as VIP is expected to exhibit some degree of temporal and spatial locality in its requests to memory. Experiments described in section 7.2.3 confirm this intuition,

therefore VIP uses an open-page policy.

Refresh in modern DRAM systems is a source of overhead. When a bank is being refreshed, no additional commands may be issued to the bank until the refresh cycle is completed. If we assume that the duration of refresh is similar to DDR3 DRAM, this locks up each bank within the HMC for approximately 400 cycles, leading to significantly higher latencies for some memory requests. Newer DRAM standards, such as JEDEC DDR4 [2, Table 24] allow refresh to occur at a higher frequency and for a shorter duration (effectively decreasing both t_{REFI} and t_{RFC}). Experiments discussed in section 7.2.3 show that reducing both t_{RFC} and t_{REFI} so that rows are refreshed every 1.95 µs instead of every 7.8 µs (approximately matching the DDR4 refresh_4x mode) results in low refresh overhead.

CHAPTER 6

WRITING SOFTWARE FOR VIP

This chapter describes how parallel implementations of BP-M, and convolutional and fully-connected layers for the VGG networks [112] were written. While VIP would make a good compiler target due to its use of a vector-processing paradigm, development of a compiler is the subject of future work; VIP code described in this work is manually written assembly.

6.1 Belief Propagation

Chapter 3 described the design of an FPGA accelerator for the sequential tree-reweighted belief propagation (TRW-S) algorithm. The TRW-S algorithm, however, requires that parallel processors performing message updates across the diagonal work in lock-step as they update messages. While this could be done on VIP, it will require many synchronisation operations that must be performed through DRAM, which will lead to a loss in performance. As a result, this chapter discusses another algorithm, Tappen's BP-M algorithm [119] (discussed in section 2.1.2) as it can be parallelised easily, and leads to only a small degradation in convergence as compared against TRW-S [118].

The BP-M algorithm imposes a strict sequential order for message updates in a given direction, however, parallelism exists in the orthogonal direction. We could, in theory parallelize line 1 in fig. 2.3 by allowing each processing engine (PE) to execute one instance of the loop. This naive approach is, however, inefficient when we consider data locality. For example, if we parallelise execution for the left-right directions, each PE will process a fixed number of rows

of the graph. Assuming that the data is laid out so that each PE accesses its own vault in this pass, the data will be scattered across multiple vaults in the next pass when the direction of message updates changes to the up-down direction. We could parallelise this execution by allowing each PE to only update the messages in the rows assigned to that PE, but this will lead to serialization of message updates (messages must be updated sequentially along columns, and PEs must update messages sequentially across rows due to the assignment of data to the PEs), throttling performance.

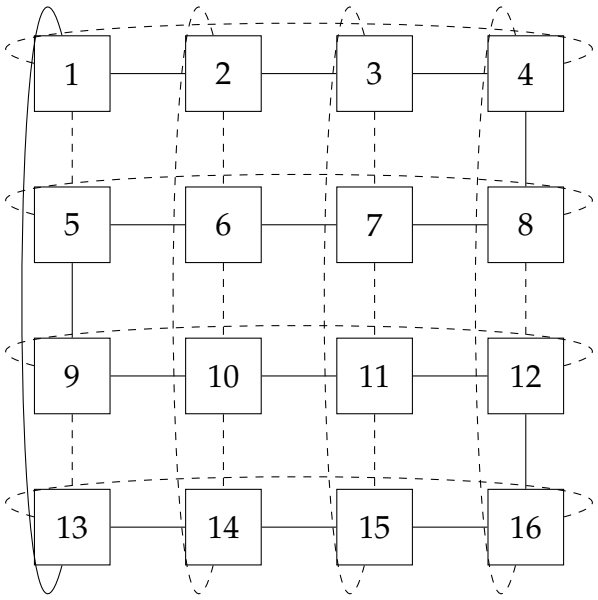
This calls for a different method to parallelise the execution of the BP-M algorithm. The best way to parallelise the execution is to divide the image into a square grid of tiles, and assign tiles to different vaults in the hybrid memory cube (HMC) such that each row and each column of this grid contains tiles that are assigned to different vaults. In this manner, we can ensure that PEs in each vault will be busy whether updating messages along rows or columns. Figure 6.1 shows how BP-M in one given direction (the same code as shown in fig. 2.3) is tiled, parallelised, and vectorised for execution on VIP. In order to minimise the data movement between vaults for messages sent across tile boundaries, we can ensure that neighbouring tiles are always assigned to vaults that are only a single network link away from each other. One way to do this is to use only the subset of links in the 2D torus network that form a ring network between all vaults, and assign vaults along this ring to each row of tiles. For the next row, we can shift the assignment by one. Figure 6.2 illustrates this assignment of tiles to vaults for a system with 4×4 vaults. Once the PEs update all messages within their assigned tile, they write messages crossing tile boundaries into the appropriate location of the next vault, and indicate that the messages are written using full-empty synchronisation variables. Once all

PEs have updated all messages in a certain direction, they synchronise using a distributed barrier (which is again written so that PEs access either their local vault or a vault one network link away). This ensures that we achieve parallelism when updating messages along both rows and columns, communication between vaults is limited to only messages that cross tile boundaries, and all communication occurs over at most one network link (along a ring connecting all vaults in the HMC). Given that non-local traffic occurs only at the tile boundaries, and that the tile size for full-HD video is 60×34, we can compute analytically that the non-local traffic will be 0.41 % to 0.72 %.

The basic kernel for updating messages in belief propagation (BP) consists of adding up four vectors (three incoming messages and data costs, corresponding to the loop on line 18 in fig. 6.1), then performing the min-sum update with the smoothness cost (matrix-vector operation corresponding to the loop on line line 21 in fig. 6.1). Finally, to prevent data overflow or underflow, we can subtract either the minimum, maximum, or even the first entry of the resulting message, as information in messages is encoded as a difference between values of the vector, not their absolute values. Figure 6.3 shows a code fragment for VIP assembly code performing BP-M within a tile, corresponding to the loop on line 13 in fig. 6.1. The actual code used for evaluation is a loop-unrolled and pipelined version of this code. BP with sixteen labels will require 80 cycles for the message update kernel, this implies that the maximum achievable frame rate with full-HD video will be 30 fps.

```
1  # Parallelise over NUM_VAULTS vaults
2  for v in range(NUM_VAULTS):
3    # Parallelise over N PEs in vault
4    for n in range(N):
5      # Code executed by each PE
6      # Iterate over the tiles in the direction messages are
         ↪ being updated
7      for t in range(NUM_VAULTS):
8        # tx and ty are physical location of tile in grid
9        tx, ty = get_tile_phys_loc(v,t)
10       # Wait for corresponding PE from another vault to finish
           ↪ working on the previous tile
11       while not done(tx-1,ty,n):
12         continue
13       for j in range(0, TILE_Y,N):
14         for i in range(0, TILE_X):
15           y = ty * TILE_Y + j + n
16           x = tx * TILE_X + i
17           # Vectorise and eliminate w
18           for w in range(NUM_LABELS):
19             temp[w] = data_cost[x][y][w] + msg_right[x][y][w]
                 ↪ + msg_up[x][y][w] + msg_down[x][y][w]
20           # Vectorise (matrix-vector) and eliminate w and v
21           for v in range(NUM_LABELS):
22             msg_right[x+1][y][v] = MAX_INT
23             for w in range(NUM_LABELS):
24               msg_right[x+1][y][v] = min(msg_right[x+1][y][v],
                   ↪ temp[w] + smooth_cost[v][w])
25           # Vectorise and eliminate w
26           for w in reversed(range(NUM_LABELS)):
27             msg_right[x+1][y][w] -= msg_right[x+1][y][0]
28       # Mark this tile as done
29       mark_done(tx,ty,n)
```

Figure 6.1: Code fragment showing parallel and vector execution of BP-M on VIP. All PEs work in parallel, accessing tiles of data sequentially. As the tiles are allocated in the manner shown in fig. 6.2b, the physical X-Y locations of a tile must be obtained via a lookup, implemented as a pointer in the tile data structure in VIP assembly code.



(a) Network links used

1	2	3	4	8	7	6	5	9	10	11	12	16	15	14	13
2	3	4	8	7	6	5	9	10	11	12	16	15	14	13	1
3	4	8	7	6	5	9	10	11	12	16	15	14	13	1	2
4	8	7	6	5	9	10	11	12	16	15	14	13	1	2	3
8	7	6	5	9	10	11	12	16	15	14	13	1	2	3	4
7	6	5	9	10	11	12	16	15	14	13	1	2	3	4	8
6	5	9	10	11	12	16	15	14	13	1	2	3	4	8	7
5	9	10	11	12	16	15	14	13	1	2	3	4	8	7	6
9	10	11	12	16	15	14	13	1	2	3	4	8	7	6	5
10	11	12	16	15	14	13	1	2	3	4	8	7	6	5	9
11	12	16	15	14	13	1	2	3	4	8	7	6	5	9	10
12	16	15	14	13	1	2	3	4	8	7	6	5	9	10	11
16	15	14	13	1	2	3	4	8	7	6	5	9	10	11	12
15	14	13	1	2	3	4	8	7	6	5	9	10	11	12	16
14	13	1	2	3	4	8	7	6	5	9	10	11	12	16	15
13	1	2	3	4	8	7	6	5	9	10	11	12	16	15	14

(b) Assignment of tiles to vaults

Figure 6.2: Assignment of tiles to vaults in VIP shown for an example system with just 4×4 vaults. The number of vaults is reduced for the sake of illustration. Also shown, the network links used out of the links in the 2D torus network. We can confirm that any communication across tile boundaries occurs only across the links indicated by solid lines, the links indicated by dashed lines are never used.

```
1  update_loop:
2          mov r1, #0; set j variable
3          ; Load data such as block size,
4          ; DRAM addresses for messages, etc.
5          ...
6  j_loop:
7          mov r2, #0; set i variable
8          ; set DRAM addresses for messages in
9          ; r7--r10, SRAM addresses in r11--r15
10         ...
11         ; Load in the fourth message just once
12         ; will be shared between iterations of
13         ; the i_loop
14         ; r61 contains length of vector
15         ld.sram [16-bit] r14, r10, r61
16 i_loop:
17         ; Load messages here
18         ld.sram [16-bit] r11, r7, r61
19         ld.sram [16-bit] r12, r8, r61
20         ld.sram [16-bit] r13, r9, r61
21         ; Perform computation to update message
22         v.v.add [16-bit] r11, r11, r12
23         v.v.add [16-bit] r11, r11, r13
24         v.v.add [16-bit] r11, r11, r14
25         ; Smoothness matrix SRAM address in r15.
26         ; r16 = temporary storage in SRAM.
27         m.v.add.min [16-bit] r16, r15, r11
28         v.s.sub [16-bit] r14, r16, r16
29         ; Write updated message to DRAM
30         st.sram [16-bit] r10, r14, r61
31         ; Bump loop variable and pointers
32         add r2, r2, #1
33         ...
34         ; Inner loop for updating messages
35         blt r2, r20, i_loop
36         ; Bump outer loop variable and pointers
37         add r1, r1, #1
38         ...
39         ; Outer loop
40         blt r1, r21, j_loop
```

Figure 6.3: VIP assembly code fragment for BP-M on a tile

6.2 Convolutional Neural Networks

Convolutional neural networks (CNNs) are easier to parallelize as each output feature for each layer may be computed in parallel. We utilize the X-Y structure of the activations, and divide the inputs for each layer into a series of X-Y tiles, which are assigned to vaults within VIP in the corresponding X-Y locations. The general pattern for computation in any CNN follows a template. Load in as many $k \times k \times z$ filters into the scratchpad as possible, while being able to also store $(k+1) \times k \times z$ inputs. While applying the loaded filters to the $k \times k$ window of inputs, prefetch the next $1 \times k \times z$ column of inputs. When these selected filters have been applied to all the inputs assigned to that PE, load in the next set of filters and repeat the process. VIP's use of a vector memory-memory paradigm allows us to use this template for any shape of convolutional inputs. Figure 6.4 shows this basic template. Additionally, code is written in a way that outputs from one layer are already in the right location to be consumed as inputs by the next layer. Figure 6.5 shows the basic kernel used for computing convolutions within a tile, the actual code used applies multiple filters within the inner loop.

The length of filters in the Z dimension can get much larger than can be stored in a 4 KiB scratchpad. In this case, we break the inputs and filters into segments along the Z dimension so that they may fit within VIP's scratchpad, and distribute these segments across vaults in the X dimension of the memory system (correspondingly, we have fewer tiles in the X dimension, leading to an increase in the dimension of the tile; this is desirable as it increases data reuse), and assign segments of inputs and filters to PEs in corresponding vaults. The PEs can apply these partial filters to partial inputs to generate partial results.

```
1  # Parallelise over vaults in Y direction
2  for y in range(0, IMG_Y, TILE_Y):
3    # Parellelise over vaults in X direction
4    for x in range(0, IMG_X, TILE_X):
5        # Parallelise over N PEs in a vault
6        for p in range(N):
7          # Code executed by each PE
8          for z in range(0, OUTPUT_CHANNELS, N):
9            for m in range(TILE_Y):
10             for n in range(TILE_X):
11               out[x+m][y+n][z+p] = 0;
12               for i in range(KERNEL_X):
13                 # Vectorise the vertical and horizontal vector
                      ↪ operations in the nested loops below
14                 for j in range(KERNEL_Y):
15                   for k in range(INPUT_CHANNELS):
16                     out[x+m][y+n][vx][z+p] +=
                          ↪ weight[z+p][i][j][k] *
                          ↪ in_[x+m-i][y+n-j][k]
17               out[x+m][y+n][z+p] += bias[z+p]
18               out[x+m][y+n][z+p] = max(out[x+m][y+n][z+p], 0)
```

Figure 6.4: Code fragment showing the parallelisation and vector execution of CNNs on VIP. This algorithm used for the later VGG layers is modified by tiling the application of each filter and allowing PEs from different vaults to compute partial results which are subsequently accumulated.

Finally, the PEs across the multiple vaults synchronise and combine their partial results. Experimental results show that the accumulation of partial results takes a very small fraction of the time taken to compute the partial results themselves.

The input features are streamed in, and filters reused to compute the resulting outputs. The VGG-16 convolutional layers [112] fall into three categories, depending on the size of the input features and number of filters.

Short input features The first convolutional layer in VGG-16 has just three input feature maps and 64 filters. As all 64 filters can be stored in the scratchpad of a single PE along with a window of input features, the most efficient utilisation

```
1          ; Load in filter
2          ...
3          ; Set up i loop variable (9x9 tile)
4          mov r5, #9
5  i_loop:
6          ; Set up DRAM and SRAM addresses, load initial
               ↪ features into the SRAM.
7          ...
8          ; Set up j variable (9x9 tile)
9          mov r4, #9
10 j_loop:
11         ; Vector length of 64 * 3 = 192 elements
12         mov r18, #192
13         set.vl r18
14         ; Single matrix row, i.e. f2 category operation
15         mov r18, #1
16         set.mr r18
17         ; Partial convolution, first column
18         m.v.mul.add [16-bit] r15, r7, r20
19         ; Pipeline loads for next input feature column here
20         ...
21         ; Bump the output SRAM pointer to the next element
22         add r15, r15, #2
23         ; Partial convolution, second column
24         m.v.mul.add [16-bit] r15, r8, r21
25         add r15, r15, #2
26         ; Partial convolution, third column
27         m.v.mul.add [16-bit] r15, r9, r22
28         ; Accumulate partial convolution results
29         mov r18, #3
30         set.vl r18
31         ; Ensure convolutions are complete by draining vector
               ↪ pipeline
32         v.drain
33         ; Accumulate partial results here.
34         m.v.nop.add [16-bit] r24, r24, r24
35         ; Write results back to DRAM
36         ...
37         sub r4, r4, #1
38         bne r4, r0, j_loop
39         sub r5, r5, #1
40         bne r5, r0, i_loop
```

Figure 6.5: VIP assembly code fragment for a CNN tile

of PEs is to have them compute independent strips of a single tile. We create a 4×8 grid of tiles, and assign PEs from a vault to work on a single tile. Four PEs work on a 56×28 tile of input features, each PE operating on 7 rows.

Sixty-four input features The next convolutional layers in VGG-16 use 64 input feature maps. In this case, we only have space for two filters and a $4 \times 3 \times 64$ input feature volume. This may be scheduled by allocating different filters to different PEs while keeping the same basic tiling structure as the first layer.

More than sixty-four input features There are two trends in later VGG-16 layers. First, the number of input features increases to more than 64, which means that a complete filter cannot fit into VIP's scratchpad. Second, the feature size (and consequently the tile size) in the X and Y dimensions decreases. Decreasing tile size in the X dimension increases overheads as an entire feature window must be loaded in before a row can be processed. These two trends change the way the later layers are stored and processed. Instead of storing an entire tile in a single vault, different tile slices in the Z dimension may be stored in different vaults, and the tile size in the X dimension can be increased. PEs in a vault compute convolutions between the filter and the feature slices (64 elements in the Z dimension) stored in that vault. Finally, PEs (across multiple vaults) assigned a tile synchronise and accumulate these partial convolutions, add biases, and perform the ReLU operation. This computation is structured so that PEs accumulate only the subset of outputs that will be used as inputs for the next layer in the same vault.

Pooling layers Pooling operations can be merged into the same phase of the code which collects partial results, adds biases, and applies the ReLU activation function. Later pooling layers, however, become too large to be merged into this phase without running out of scratchpad space. Therefore, these operations are performed separately.

6.3 Multi-layer Perceptrons

Parallelizing multi-layer perceptrons (MLPs) is fairly straightforward. At the end of the last pooling layer, a 25088×1 vector remains, distributed in segments among the vaults at the very top of the HMC. We can distribute tiles from the 4096×25088 weight matrix among all the vaults of the HMC. A fully-connected layer is executed in three passes. First, all PEs copy assigned segments of the input vector into their local vaults. In the second pass, PEs in a vault compute partial products of their respective matrix tiles with their vector segments. In the third and final pass, PEs in the vaults on the left side of the HMC accumulate partial products from vaults in the same row, add biases and perform the rectified linear unit (ReLU) operation. Figure 6.6 shows how this code is parallelised and vectorised for execution on VIP. Subsequent fully-connected layers are executed in a similar way, alternating the way data are moved, from the left-side vaults to the top vaults in the HMC.

As MLPs consist of matrix-vector multiplications, these algorithms are often memory bandwidth bound. This is discussed in some detail in section 7.2.1. A common trick to increase the number of operations performed for each byte moved (arithmetic intensity) is to batch execution, applying the same weights

to multiple vectors in the batch, reusing weights. When operating on batched MLPs, the second phase of matrix-vector computation is augmented with an inner loop that loops over different input vector segments in the batch.

6.4 Recurrent Neural Networks

The recurrent neural network (RNN) used in this work is adapted from work by Han et al. [46]. It is similar to the design described in section 2.2.3, with some modifications. Notably, the weight matrix for peephole connections is restricted to be diagonal, essentially replacing that matrix-vector multiplication with an element-wise multiplication

$$f_t = \sigma\left(W_f[y_{t-1}, x_t] + w_{fc} \odot c_{t-1} + b_f\right) \tag{6.1a}$$

$$i_t = \sigma\left(W_i[y_{t-1}, x_t] + w_{ic} \odot c_{t-1} + b_i\right) \tag{6.1b}$$

$$\tilde{c}_t = \tanh\left(W_{\tilde{c}}[y_{t-1}, x_t] + b_{\tilde{c}}\right) \tag{6.1c}$$

$$c_t = i_t \odot \tilde{c}_t + f_t \odot c_{t-1} \tag{6.1d}$$

$$o_t = \sigma\left(W_o[y_{t-1}, x_t] + w_{oc} \odot c_t + b_o\right) \tag{6.1e}$$

$$m_t = o_t \odot \tanh(c_t) \tag{6.1f}$$

$$y_t = W_y m_t \tag{6.1g}$$

```
# Parallelise over vaults in Y dimension
for y in range(VAULTS_Y):
  # Parallelise over vaults in X dimension
  for x in range(VAULTS_X):
    # Parallelise over N PEs in vault
    for p in range(N):
      # Code executed by each PE
      IN_V = INPUT_LEN / VAULTS_X
      IN_P = IN_V / N
      for b in range(BATCH_SIZE):
        for i in range(0, IN_P, VL):
          # Vectorise and eliminate v
          for v in range(VL):
            in_l[x][y][b][p*IN_P+i+v] =
               ↪ in_[b][i+v+p*IN_P+x*IN_V]
```

(a) Copy data to local vaults

```
# Parallelise over vaults in Y dimenion
for y in range(VAULTS_Y):
  # Parallelise over vaults in X dimension
  for x in range(VAULTS_X):
    # Parallelise over N PEs in vault
    for p in range(N):
      # Code executed by each PE
      # MR = Number of matrix rows that fit in scratchpad
      OUT_V = OUTPUT_LEN / VAULTS_Y
      OUT_P = OUT_V / N
      for i in range(0, OUT_P, MR):
        for j in range(0, IN_P, VL):
          for b in range(BATCH_SIZE):
            # Vectorise and eliminate k and l
            for k in range(MR):
              out_l[x][y][b][p*OUT_P+i+k] = 0
              for l in range(VL):
                out_l[x][y][b][p*OUT_P+i+k] +=
                   ↪ weights[x][y][p*OUT_P+i+k][j+l] *
                   ↪ in_l[x][y][b][j+l]
```

(b) Perform local matrix-vector multiplication

Figure 6.6: Code fragment showing parallelisation and vector execution of a (batched) matrix-vector multiplication on VIP. This code consists of three phases – copy inputs to local vaults, perform local matrix-vector multiplication, and finally accumulate partial results of local matrix-vector multiplication.

```
# Parallelise over vaults in Y dimension
for y in range(VAULTS_Y):
  # Parallelise over PEs within vault
  for p in range(N):
    # Code executed by each PE
    # Only PEs in the left-most vault work in this phase
    OUT_V = OUTPUT_LEN / VAULTS_Y
    OUT_P = OUT_V / N
    if VAULT_ID.X == 0:
      for i in range(0, OUT_P, VL):
        for b in range(BATCH_SIZE):
          # Vectorise and eliminate j
          for j in range(VL):
            out[b][y*OUT_V+p*OUT_P+i+j] =
               ↪ bias[y*OUT_V+p*OUT_P+i+j]
          for x in range(VAULTS_X):
            # Vectorise and eliminate j
            for j in range(VL):
              out[b][y*OUT_V+p*OUT_P+i+j] +=
                 ↪ out_l[x][y][b][p*OUT_P+i+j]
          # Vectorise and eliminate j
          for j in range(VL):
            out[b][y*OUT_V+p*OUT_P+i+j] =
               ↪ max(out[b][y*OUT_V+p*OUT_P+i+j], 0)
```

(c) Accumulate partial results, add biases, and activation

Figure 6.6: (continued)

As RNNs are also composed of matrix-vector multiplications, their software implementation on VIP is adapted from the MLP implementation (shown in fig. 6.6). As the matrices used in ESE are taller than they are wide, the outputs from the previous time step (y_{t-1}) and inputs (x_t) are concatenated at the left-most vaults in VIP, and the internal state (c_t) as well as the vectors that are used to update the internal state (f_t, i_t, $\tilde{c}_t$, o_t, m_t) are stored at the top-most vaults. The three matrix-vector multiplications for computing parts of f_t, i_t, and o_t are computed so as to place the partial results in the top-most vaults. At the same time, the element-wise multiplication with c_{t-1} or c_t is performed. Computation for $\tilde{c}_t$ skips this element-wise multiplication step. Updates for c_t and m_t

are simple element-wise multiplications performed by the top-most vaults in VIP, while the matrix-vector multiplication for computing y_t is performed in a manner that places y_t back in its appropriate location in the left-most vaults.

CHAPTER 7

VIP EVALUATION

This chapter discusses the methods used to evaluate the performance, area, and power required by VIP on the workloads considered, as well as the results of these evaluations.

7.1 Evaluation Methods

This work evaluates VIP's performance through detailed, execution-driven microarchitecture simulations. Area and power are evaluated by synthesising a VIP processing engine (PE) in 28 nm TSMC technology using a commercial standard-cell library supplied by ARM. Custom CUDA code for BP-M is written in order to evaluate its performance on an Nvidia Titan X (Pascal) GPU, existing accelerator [16] and GPU [58, 93] results are used as baselines for convolutional neural network (CNN) performance. GPUs are the closest approximation to vector processors readily available today; IBM's Active Memory Cube [88] is still a research project and ARM's scalable vector extensions (SVE) specification [9] is relatively new, but does not specify a microarchitecture making performance estimation difficult. Therefore, this work uses GPUs instead of vector processors to serve as a baseline.

7.1.1 Simulation Infrastructure

This work uses an execution-driven microarchitecture simulator which faithfully models all pipelines including stalls and contention on shared ports in the

Table 7.1: Parameters used in memory simulation

Parameter	Value	Parameter	Value
HMC vaults	32	Banks per vault	16
HMC vault data width	32 bits	Burst length	8
Row buffer policy	open-page	Trans queue depth	32
Address mapping	vault-row-bank-col	Cmd queue depth	32
t_{CK}	0.8 ns	t_{RP}	13.75 ns
t_{CCD}	5 ns	t_{RCD}	13.75 ns
t_{CL}	13.75 ns	t_{WR}	15 ns
t_{RAS}	27.5 ns	t_{RFC}	81.5 ns
t_{REFI}	1.95 μs		

VIP PE. The simulator models the same pipeline structure validated through synthesis of an RTL implementation of a VIP PE. The network is modelled via contention and bandwidth at all injection and ejection ports but not link-level contention for the sake of simulation speed; DRAMSim2 [104], an open source DRAM simulator, is used to model the memory system.

Timing parameters used by Kim et al. [66] have since been widely used in literature relating to hybrid memory cube (HMC) simulation. [4, 6, 12, 105] This work uses the same parameters with some changes, such as changing the address mapping scheme, row-buffer policy, and refresh rate as discussed in section 5.2. Table 7.1 lists the key parameters used in the memory system simulation.

A single BP-M iteration is run on the simulator to get the number of cycles required for one iteration. The simulated code is verified to be correct by comparing its outputs against a reference C++ implementation. Similarly, an independent tile from each VGG-16 convolutional layer is run on the simulator and its results verified against a reference C++ implementation. An independent tile is a segment of the input features that does not share any resources (PEs, memory

requests, or network bandwidth) with another tile. All independent tiles have the same amount of work; if the amount of work is unequal, the largest tile is simulated. Simulating a single independent tile greatly reduces the simulation time without affecting the reported execution time. As the matrix-vector (or matrix-matrix in the case of batched inputs) multiplications comprising the fully-connected layers in VGG-16 and in the recurrent neural network (RNN) workloads cannot be broken into independent tiles, we must simulate the complete network.

7.1.2 Baseline Implementations

GPU Implementation of BP-M

While there has been prior work on accelerating belief propagation (BP) on GPUs [42, 77, 134], such work has been done on older GPUs which makes simply reporting numbers unfair. This difficulty is compounded by the fact that these works make assumptions on the nature of the smoothness costs (e.g.: that they are linear or quadratic and diagonally dominant), use different message update schemes from BP-M, and that the source code for these implementations is not available.

It is therefore necessary to write a new BP-M implementation. Writing this implementation in Tensorflow [3], resulted in unacceptably long compilation and run times. This is probably because each message update had to be expressed as a combination of Tensorflow primitive operations which resulted in a very large number of nodes in the Tensorflow dataflow graph. Extracting high performance from Tensorflow would require writing and compiling native code

either for the BP-M algorithm or for a subset of operations involved. Instead of writing Tensorflow kernels in CUDA for the BP-M algorithm, we might as well write a native CUDA implementation of BP-M knowing that the performance of the native implementation will be at least as good as the implementation that could be written using Tensorflow and these new custom operators. Nvidia profiling tools help tune the implementation on an Nvidia Titan X (Pascal) GPU.

The Titan X utilises Nvidia's Pascal architecture and providing $480\,\mathrm{GB\,s^{-1}}$ and 11 TFLOP/s of peak memory bandwidth and computational throughput respectively [131]. This GPU has the required memory bandwidth and significantly higher compute capability than required for BP-M. Shared memory on the GPU is used to store the smoothness costs as well as the intermediate results of computation. This is done because the latency of accessing shared memory is much less than the latency of accessing the GDDR5X main memory. Code is tuned to minimise the number of bank conflicts in shared memory, however, it is not tuned so that it would work well for only a particular image size or number of labels, instead these as parameters are supplied to the kernel. No assumptions are made regarding the structure of the smoothness cost functions either. This is a fair comparison as these assumptions are not made for the simulated VIP hardware either.

As we cannot efficiently write the code for BP-M in Tensorflow without registering new native operations for the primitives encountered in BP, we cannot run BP-M on the Google Cloud TPU without knowledge of the low-level details on the tensor processing unit (TPU). However, based on publicly available information on the TPU published by Jouppi et al. [59], we see that the TPU systolic array has support only for multiply accumulate (MAC) operations. As

discussed in section 2.1.2, BP requires sum-min operations.

Reference CNN Architectures

There are a number of existing accelerator implementations for CNNs, e.g.: Cambricon by Liu et al. [80], Eyeriss by Chen et al. [16], Neurocube by Kim et al. [65], Neurostream by Azarkhish et al. [13], and Tetris by Gao et al. [39]. Unfortunately, many of these implementations use different neural networks which makes direct comparison difficult. Eyeriss is a popular, widely-cited architecture which reports absolute performance on the VGG-16 network, making it a suitable baseline. Tetris builds on Eyeriss by significantly increasing the number of PE compared to Eyeriss, and using a HMC to supply high bandwidth to these PEs. Unfortunately, Tetris only reports speedups with respect to a 2D design with LPDDR DRAM; the absolute performance numbers for this baseline are not available either, making direct performance comparisons impossible. Finally, GPU reference implementations of various CNNs are readily available and benchmarked on various GPU platforms, including Nvidia Titan X (Pascal) and Nvidia Jetson [58, 93].

7.1.3 RTL Implementation

It is critical to have good power and area estimates in order to evaluate the feasibility of a design that is to be integrated into a 3D logic+memory stack.

Synthesis of a single VIP PE in 28 nm TSMC technology using a commercial standard-cell library from ARM offers estimates for VIP's area and power.

CACTI 6.5 [85] provides estimates for the area and energy required by SRAMs. (Note that the smallest feature size that CACTI can model is 32 nm, so the area and power estimates provided by CACTI will be somewhat pessimistic.) In order to make the final design as close as possible to a layout with real SRAMs, the area and energy numbers from CACTI are used to create black-box SRAM models for the ASIC toolflow. VIP requires eight 512×8 bit SRAMs for the scratchpad, a 64×64 bit SRAM for the register file, a 64×32 bit SRAM to model the load-store queue, and a 1024×32 bit SRAM for the instruction buffer. Small BP and CNN kernels running on the RTL model are used to verify its correctness against a reference implementation. Switching activity factors from these RTL simulations are supplied as inputs to Synopsys PrimeTime to estimate power consumption.

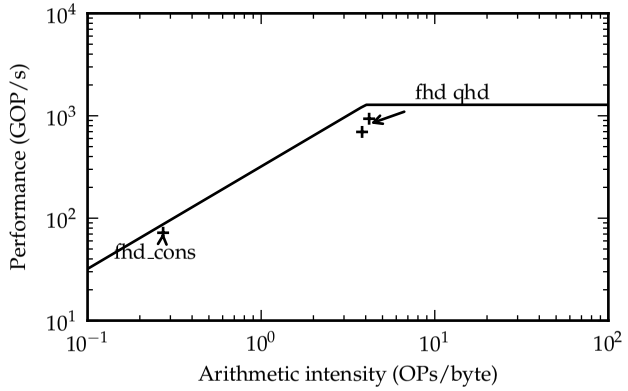
7.2 Experimental Results

7.2.1 End-to-end Application Performance

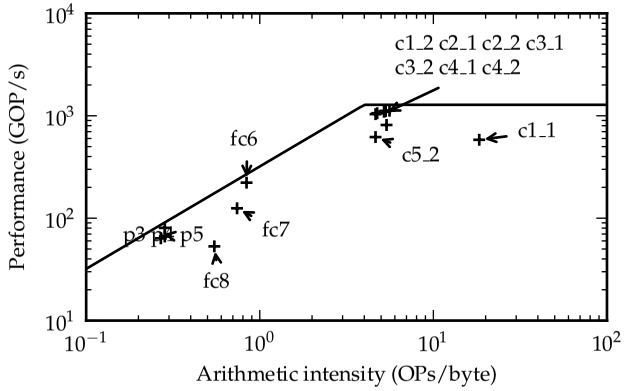
Table 7.2 shows a summary of VIP's performance (execution time) on the applications considered, BP on Markov random fields (MRFs) and deep neural networks (DNNs) including CNNs, multi-layer perceptrons (MLPs), and RNNs; and how VIP's execution time, area, technology, and power compare against the baseline architectures considered. Figure 7.1 shows these applications on a roofline plot for VIP. The 'roofline' represents the maximum achievable performance of a system given the peak memory bandwidth and computational throughput available. The horizontal axis indicates the arithmetic intensity of

Table 7.2: A summary of the performance of this work – VIP – compared against prior work on ASIC accelerators and GPUs for MRF, CNN, and RNN workloads.

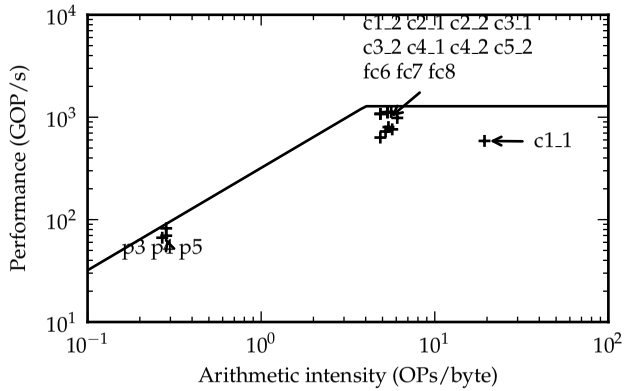
Markov random field					
System	**Iterations**	**Time (ms)**	**Power (W)**	**Technology (nm)**	**Area (mm^2)**
Optical Gibbs Sampling [127]	5000	1100	12	15	200 + 12
Tile BP (720p) [17]	(1,2)	32.7	0.242	90	9
Pascal Titan X	8	92.2	250	16	471
VIP	**8**	**40.8**	**3.5**	**28**	**18**
Convolutional neural network – VGG-16 (convolutional layers only)					
System	**Batch Size**	**Time (ms)**	**Power (W)**	**Technology (nm)**	**Area (mm^2)**
Eyeriss [16]	3	4309	0.236	65	12
VIP	**3**	**91.5**	**4.8**	**28**	**18**
Convolutional neural network – VGG-16 (full network)					
System	**Batch Size**	**Time (ms)**	**Power (W)**	**Technology (nm)**	**Area (mm^2)**
Pascal Titan X [58]	16	41.6	250	16	471
VIP	**16**	**492**	**4.8**	**28**	**18**
VIP	**1**	**32.2**	**4.8**	**28**	**18**
Convolutional neural network – VGG-19 (full network)					
System	**Batch Size**	**Time (ms)**	**Power (W)**	**Technology (nm)**	**Area (mm^2)**
Nvidia Volta [21, 93]	1	2.2	144	12	815
Nvidia Jetson TX2 [93]	1	42.2	10	16	unknown
VIP	**1**	**40.5**	**4.8**	**28**	**18**
Recurrent neural network – ESE					
System	**Batch Size**	**Time (µs)**	**Power (W)**	**Technology (nm)**	**Area (mm^2)**
ESE (sparse) [46]	32	82.7	41	20	unknown
Pascal Titan X (dense) [46]	32	240	202	16	471
VIP (dense)	**32**	**514**	**4.8**	**28**	**18**



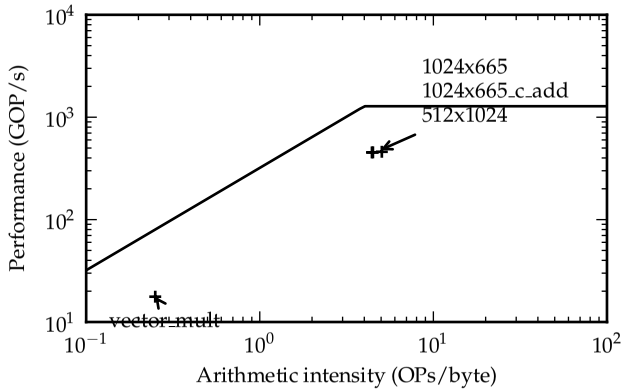
(a) Belief propagation



(b) VGG-16, batch size 1



(c) VGG-16, batch size 16



(d) RNN used in ESE

Figure 7.1: Roofline plots for VIP running belief propagation on a full-HD (fhd) and quarter-HD (qhd) image, and convolutional neural network (c), pooling (p) and multi-layer perceptron (fc) layers from the VGG-16 [112] network for batch sizes of 1 and 16, and the recurrent neural network used by Han et al. [46].

the application (i.e. the operations performed for each byte of data moved). The vertical axis represents the performance of the application (i.e. operations per second). (As VIP supports multiple data types, we must be careful when defining peak performance. In these plots, peak performance is computed by multiplying the number of ALUs in the vector unit, in both the horizontal and vertical stages, with the clock frequency and the number of PEs. The ALU in the scalar pipeline is ignored as it is meant only for control operations. However, memory accesses by both the scalar and vector units are considered when computing arithmetic intensity.) The distance of an application below the roofline indicates the performance lost relative to the peak. Applications with low arithmetic intensity, i.e. to the left of the knee-point of the roofline will be bound by memory bandwidth, while applications with high arithmetic intensity, i.e. to the right of the knee-point will be limited by the computational throughput of a system. A system that balances the compute and memory bandwidth requirements will find that applications fall at or near the knee-point, as is the case for all the applications considered in this thesis. This is no surprise; VIP was designed in order to balance the memory bandwidth and computational intensity of these applications. The subsequent sections discuss the performance of individual applications.

Belief Propagation

Baseline, as well as hierarchical BP-M algorithm are run for full-HD images with 16 labels. (The hierarchical BP-M algorithm is similar to the one proposed by Felzenszwalb and Huttenlocher [33].) The baseline BP-M algorithm has been discussed in section 2.1.2, the hierarchical implementation consists of

four phases – construct a coarser version of the graph by pooling neighbouring data costs, perform BP-M on the quarter-sized image, and finally copy the resulting messages back to the original, full-sized graph. As hierarchical BP-M converges faster than baseline BP-M, it only requires five iterations to reach the same energy as eight iterations of baseline BP-M, allowing these overheads to be amortised. A single iteration is run in order to reduce simulation time, results for multiple iterations are extrapolated.

A single iteration of BP-M takes 5.1 ms. Eight BP-M iterations will, therefore, require 40.8 ms, allowing VIP to execute a depth-from-stereo task at a real-time frame rate of 24 fps on a full-HD image with good results. The construct and copy operations for hierarchical BP-M require 0.35 ms and 1.25 ms respectively, while an iteration of BP-M on the quarter-HD MRF requires 1.7 ms. As construct and copy are performed only once per frame, five iterations of hierarchical BP-M will require 35.6 ms, which again allows VIP to process full-HD video at 24 fps.

The plot in fig. 7.1a shows that all the BP kernels lie near the knee-point with the exception of the construct operation, which is necessarily memory bandwidth bound because of its low arithmetic intensity. This indicates that our system balances the memory bandwidth and compute required for the BP kernel. Even the construct operation is near the roofline indicating that we achieve close to peak performance for a kernel bound by memory bandwidth.

The primary baseline for BP is a CUDA implementation running on an Nvidia Titan X (Pascal) GPU, as discussed in section 7.1.2. (Other systems are discussed in chapter 8.) The Titan X requires 11.5 ms for one iteration of BP-M. This may be attributed to the fact that GPUs are engineered to work with very wide vectors with high levels of data-parallelism; BP uses much shorter vectors. The

Nvidia profiler reports that the BP-M algorithm does not have enough parallelism to keep the Titan X fully occupied, these two factors result in the GPU being bottlenecked by both instruction and memory latency. (The GPU has a prefetcher, so a lack of prefetching is not the problem here.) The Nvidia profiler also reports the utilised memory bandwidth halves when updating messages in the left-right directions compared against message updates in the up-down direction. This is probably due to the fact that we store data in a row-major fashion on the GPU. (Note that tiling data is used in case of VIP for data locality within a vault, a GPU will actually benefit from row-major storage.) When updating messages in the up-down directions, the GPU access entire rows of data as a wide memory fetch. However, for the left-right directions, the GPU must access data in a column, which wastes some of the fetched data decreasing performance.

It is possible that the Nvidia Volta [94] GPU may perform better on BP-M, as the Volta uses high-bandwidth memory (HBM) DRAM instead of GDDR used in the Titan X. As I do not have access to a Volta GPU, it is difficult to evaluate its performance on BP-M, as these numbers are not reported, unlike for CNNs. Additionally, the Volta has tensor cores designed to accelerate training large neural networks. These tensor cores, however, can only process MAC operations, therefore, are unable to work with the min-sum BP update. On the other end of the spectrum, the Nvidia Jetson TX2 provides a peak bandwidth of just $59.7\,\mathrm{GB\,s^{-1}}$. BP-M is a streaming workload, therefore has little potential for data reuse. As a result, the performance of a GPU will be dependent on the memory bandwidth available. As fig. 7.2a indicates, BP-M requires a memory bandwidth of $223\,\mathrm{GB\,s^{-1}}$ for a frame rate of 24 fps. Even if the Jetson achieves its peak DRAM bandwidth, its performance will be a third of VIP's performance.

Another interesting accelerator to be considered is Intel's Knights Landing [62], which can provide 3 TFLOPS and a bandwidth of $460\,\mathrm{GB\,s^{-1}}$. The performance of this system, however, will depend on whether it can efficiently achieve close to its peak memory bandwidth. The Knights Landing co-processor ultimately follows a many-core approach with built-in AVX-512 extensions. Ultimately, this means that the Knights Landing dedicates resources towards hardware complexity to maximise single-thread performance on each of its cores, consuming 245 W of power.

Convolutional Neural Networks

Convolution layers Individual layers from the VGG-16 [112] network are run on the VIP simulator. Execution time (and data moved) for each layer is listed in table 7.3. VGG-19 is another CNN similar to VGG-16, with the conv3_2, conv4_2, and conv5_2 layers replicated, making it easy to calculate VIP's performance on this network as well. The convolution, ReLU, and pooling layers before the first fully-connected layer in VGG-16 require a total of 30.9 ms. The VGG-19 layers require 39.1 ms.

Almost all other systems, whether accelerators or GPUs use a technique called batching to improve the arithmetic intensity of the algorithm. Batching works by applying the same neural network weights to multiple inputs, reusing weights, increasing the operations performed for the weights moved. Eyeriss [16] presents results with a batch size of three, while the Nvidia Titan X (Pascal) GPU was benchmarked on the VGG-16 network with a batch size of sixteen [58]. Execution time for the Nvidia Volta and Jetson TX2 GPUs is available with multiple batch sizes, including a batch size of just one. While batching

can improve the throughput of a system by reusing weights across multiple inputs, it may also increase latency. Therefore, batching may not be suitable for real-time systems with strict latency constraints.

Eyeriss reports their runtime for a batch size of three, 4309 ms. VIP requires 91.5 ms with the same batch size. Without any normalisation, it would appear that VIP is 47× faster than Eyeriss. However, we must normalise area, and technology in order to make a more meaningful comparison. Eyeriss occupies 12 mm^2 in 65 nm technology, while VIP requires 18 mm^2 in 28 nm. Therefore, if we divide Eyeriss' execution time by $18/12$ to compensate for the difference in area, and by $(65/28)^2$ in order to compensate for technology, we find that the difference drops to $\sim 6\times$. Further, VIP operates at 1.25 GHz, while Eyeriss operates at just 200 MHz. Assuming, optimistically, that Eyeriss will be able to operate at the faster clock speed at the smaller technology node and that Eyeriss' performance would scale linearly without any other bottlenecks (e.g.: memory bandwidth), the net result is that VIP would be only about 7 % worse than this scaled version of Eyeriss, at Eyeriss' own and only game.

Figure 7.1b shows various CNN kernels on a roofline plot. We can see that the pooling layers are memory-bound, but very close to the roofline. The convolution layers lie near the knee-point of the roofline plot, indicating once again that VIP is well-balanced for convolutional layers, which account for a bulk of the execution time. There are some exceptions, however. The first convolutional layer (`c1_1`) is different from the other convolutional layers, as discussed in section 6.2. The ability to load all filters into the scratchpad means that more arithmetic operations are performed on features every time they are loaded, increasing arithmetic intensity. The small vector lengths, however, mean that the

data are processed quickly, which exposes latency of the control code (loops, data-movement). This prevents VIP from achieving peak performance for this layer. On the other hand, the later convolutional layers (`c5*`) suffer in performance because the input feature maps are very small, leading to small tile sizes distributed across only half the vaults, halving both memory bandwidth and computational capacity.

Fully connected layers Fully-connected layers from the VGG-16 network are also run on the VIP simulator. As these layers consist of matrix-vector multiplications, they benefit from batching as it changes the matrix vector multiplication to matrix-matrix multiplications with higher arithmetic intensity. With a batch size of one, the fully-connected layers require 1.35 ms, with a batch size of three, they require 1.79 ms, and with a batch size of 16, 4.30 ms. Figures 7.1b and 7.1c show where the fully-connected layers lie under the performance roofline of VIP for batch sizes of one and sixteen. With a batch size of one, the fully-connected layers are memory bandwidth limited. Only the first fully-connected layer (`fc6`) lies near the roofline, the other layers lie below the roofline. The loss in performance of these layers is attributed to the fact that the matrices to be multiplied are small, meaning that synchronisation between the three phases of computation described in fig. 6.6 contributes a significant overhead to the runtime of the algorithm. Batching computation in fully connected layers increases both the arithmetic intensity as well as the computation performed between synchronisations. This allows VIP to achieve close to peak performance on these layers with larger batch sizes.

To summarise, VIP requires 32.2 ms for the entire VGG-16 network, providing a frame rate of 30 fps with a batch size of one. This indicates that VIP is an

Table 7.3: Execution time and data movement required by VGG-16 convolutional and fully connected layers on VIP. Results are shown with different batch sizes (BS).

Layer	Execution time (ms)			DRAM accesses (MiB)		
	BS = 1	**BS = 3**	**BS = 16**	**BS = 1**	**BS = 3**	**BS = 16**
conv1_1	0.319	0.954	5.078	9.666	27.965	146.914
conv1_2	3.325	9.949	53.004	635.876	1975.938	10138.566
conv2_1	2.292	6.861	37.249	330.844	218.406	5259.281
conv2_2	3.343	9.992	53.232	669.028	2207.404	10636.093
conv3_1	1.757	5.211	27.618	373.995	2004.016	5847.080
conv3_2	3.356	10.015	53.302	674.016	2004.016	10649.016
conv3_3	3.364	10.038	53.419	676.039	2010.086	10681.391
conv4_1	1.794	5.253	27.665	382.212	1110.437	5843.869
conv4_2	3.397	10.069	53.351	682.348	2010.968	10646.025
conv4_3	3.401	10.083	53.420	683.360	2014.003	10662.212
conv5_1	1.502	4.352	23.525	191.109	555.164	2925.972
conv5_2	1.502	4.352	23.525	191.109	555.164	2925.972
conv5_3	1.504	4.359	23.561	191.643	556.764	2934.503
fc6	0.929	1.330	3.394	234.601	348.735	524.021
fc7	0.270	0.311	0.717	43.470	61.079	91.511
fc8	0.155	0.145	0.184	14.358	17.848	24.429
Total	**32.211**	**93.274**	**492.246**	**5.843 GiB**	**17.264 GiB**	**87.829 GiB**

excellent fit for applications with real-time constraints on latency, and not necessarily requiring the highest possible throughput. VIP requires 492 ms for processing sixteen frames through the VGG-16 network. While this performance is lower than that of the Nvidia Titan X (Pascal) GPU that requires just 41.6 ms, the Titan X requires significantly more area and power than VIP, and its throughput oriented nature indicates that its performance would not scale proportionally if the batch size was reduced to one. We can also compute the time required for inference on the VGG-19 network by double-counting the time required for layers conv3_2, conv4_2, and conv5_2 as VGG-19 contains two sets of these layers instead of just one as in VGG-16. VIP will require 40.5 ms for the VGG-19 network, still achieving a frame rate of over 24 fps. As a comparison, the Nvidia

Jetson TX2 requires 42.2 ms [93] on the same network, achieving a frame rate just under 24 fps. The Nvidia Volta requires 2.2 ms for VGG-19, considerably faster than VIP, although it does so with a die area of 815 mm^2 in 12 nm technology against VIP's 18 mm^2 in 28 nm, a similar scaling analysis as with Eyeriss puts this at $\sim 246\times$ VIP's area. The Volta has special tensor cores that allow it to process matrix multiplications efficiently, although using these tensor cores to full capacity will require applications with very high arithmetic intensity, the Volta GPU with its tensor cores provides a peak throughput of 125 TFLOP/s, while its HBM system provides a peak bandwidth of 900 $GB\,s^{-1}$ [94]. This means that applications must have a arithmetic intensity of at least 138 operations for each byte moved in order to not be limited by memory bandwidth.

Data reuse in convolutional layers CNNs have a significant amount of data reuse because the same weights are applied to all parts of the image. However, table 7.3 indicates that the data movement for processing VGG layers is much higher than that required by architectures such as Eyeriss [16]. This is attributed to the way the VGG layers are parallelised in VIP. Each VIP PE has a 4 KiB scratchpad. In the general case (discounting the special first convolutional layers), the four PEs within a vault are applying distinct filters to the same inputs. The filters are reused as they are applied across a tile. However, PEs in different vaults apply the same filters to a different input tiles, wasting some potential for filter reuse. Inputs are loaded in as $k \times 1$ columns. Each column of input loaded is reused k times for producing k output features. The limited size of VIP's scratchpads, however, evict this input feature before it can be reused for the next row of outputs. Additionally, the scratchpads can store only a limited number of filters, which forces VIP to load the same input tile again and again

applying a new set of filters to the input, further wasting potential for input reuse. This is an unfortunate drawback of VIP's design, it was initially designed for streaming workloads with minimal data reuse, therefore it forgoes the typical strategies for exploiting temporal and spatial data locality such as a cache or a global buffer accessed by all PEs. The high memory bandwidth, however, is able to overcome this limitation, allowing VIP to achieve close to its peak performance.

Recurrent Neural Networks

The RNN used in this work is adapted from ESE by Han et al. [46]. Their work constructs an RNN for speech recognition and uses sparsity in weights to reduce the amount of computation performed. Only 10 % of weights in this RNN are non-zero allowing ESE to effectively exploit sparsity. On the other hand, VIP is designed for operation on dense data and therefore performs poorly as it wastes time computing the 90 % of zero multiplications. As a result, while ESE requires 82.7 µs for each time-step in the RNN, VIP requires 514 µs for the same. Both these results are for a batch of 32 separate inputs. For context, an Nvidia Titan X (Pascal) GPU requires 240 µs for the same benchmark, again working with dense data. Its performance with sparse data is worse, requiring 287 µs.

Both VIP and the Nvidia GPU perform poorly on the RNN used by ESE because the size of matrices multiplied are very small. In fact, VIP's runtime is just over 2.1× the performance of the Titan X, despite the Titan X advertising a peak computational throughput 10× that of VIP. VIP requires a barrier be introduced between the three phases described in fig. 6.6, the barrier itself takes a non-trivial amount of time leading to a loss in performance as the overheads as-

sociated with the barrier cannot be amortised by useful computation performed in the three phases. This is also reflected in the roofline plot shown in fig. 7.1d, as the performance of the matrix-multiplication routine is significantly below the roofline. VIP, however, will perform better with larger matrices, as shown in results for the fully-connected layers in VGG-16.

7.2.2 Sensitivity to Architectural Choices

There are a number of architectural design choices that enable VIP to achieve the performance target of real-time inference on the benchmarks considered, 16-label MRF inference on full-HD video, and the VGG-16 CNN. It is interesting to consider how VIP's performance would scale under a different set of design choices, given the evaluation presented in section 7.2.1.

VIP differs from a traditional vector architecture in two significant ways. One, it uses a vector memory-memory paradigm supplying a software-managed scratchpad memory instead of a traditional vector register file. This allows the programmer to pack irregularly sized vectors and matrices much more efficiently. Second, it provides a hardware reduction unit and support for matrix-vector operations in its ISA, which serve as a way to offload loop overheads from software to hardware, critical in the case of short vectors where the latency of these loop instructions would be difficult to hide behind vector execution.

In order to evaluate the benefits of these approaches, we can emulate a fixed vector register file in VIP by restricting the starting addresses where vectors can be stored in the scratchpad. The IBM Active Memory Cube provides sixteen 256 B registers in each vector lane [88]. For the purposes of these experiments,

I emulate the same register file configuration as it is similarly sized with VIP. If we were to naively store one vector per register, we would quickly run out of vector registers. A more efficient way to process vectors would be to pack multiple vectors into the same register. Noting that vectors are just 32 B long in the depth-from-stereo workload, we can pack eight such vectors into a single register. These eight vectors can be loaded and stored using a single gather-load or scatter-store instruction.

As VIP does not support gather-loads or scatter-stores, we can, for the purposes of writing this micro-benchmark, store data in a way that a single contiguous vector load can load these eight vectors. We can further assume that we can ignore the overheads of these operations, specifically, the extra storage required in order to store indices or addresses for the gather-load and scatter-store operations and the serialisation of these operations that would be necessary in order to maintain memory consistency. Further, as we would only like to analyse the effect from a microarchitecture perspective without inadvertently suffering secondary effects due to a change in memory access patterns, and because moving 256 B between DRAM and scratchpad is a legal instruction in VIP, the code for VIP (using the scratchpad, reduction unit, and matrix-vector instructions) also accesses memory in the same manner as the traditional vector processor.

While multiple vectors can be packed into a single vector register, operating upon vectors packed in different locations in different registers is impossible without first extracting the packed vectors. Moving N elements will require at least $\lceil N/w \rceil$ cycles, where w is the width of the scratchpad port, we can assume (optimistically) that this is the time required, and that there are no other overheads. Going back to fig. 4.2, line 3 can work with packed vectors, the rest of

the code will require extracting and repacking vectors into registers, whether or not we assume the presence of a reduction unit.

If we assume that our baseline vector processor has a reduction unit, the packed messages must be extracted into individual registers. Each row of the smoothness cost matrix must also be extracted, and an operation similar to a vector-vector dot-product be performed for computing each outgoing message element. These resulting scalar values must be repacked into their appropriate location in the vector register. If, on the other hand, we assume that a vector unit is not present, then we must execute the code as shown in fig. 4.2. We must allocate different vector registers to hold the accumulated outgoing messages as they are being generated, extract a scalar value and a smoothness matrix column, add the two and accumulate. Finally, the resulting eight outgoing messages must be repacked and stored to memory.

The micro-benchmark used to evaluate these three configurations performs BP on a 32×32 tile, in the direction such that eight independent messages can be updated by a PE using a single load or store instruction. The tile dimension is chosen to be 32 because it equals 8×4, the four vector processors in a vault can update eight messages in parallel each. If the tile size is not a multiple of 32 (as in the case of BP on a full-HD image, which uses tiles of size 60×34), the performance of the vector processor would suffer even more.

We can compute analytically, as well as simulate the performance of these three points in the design space – VIP, a vector processor with a vector register file and reduction units, and a vector processor with a vector register file and no reduction units. Analytical results show that the vector processor with a vector register file with reduction units will take $1.45\times$ as long as VIP, while a vector

processor with a vector register file without reduction units will take 2.1× as long as VIP. Experimental results from the VIP simulator with DRAMSim show that this is indeed the case, the losses in performance of the two configurations with respect to VIP are 1.57× and 2.03× respectively.

Figure 7.1 shows that all the workloads considered in this work have low arithmetic intensity, and that VIP as it is presently designed balances the compute and memory bandwidth requirements for these workloads. As a result, improving VIP's performance will require us to increase both the computational throughput as well as the memory bandwidth. Just as multiple HMC devices may be interconnected using SERDES links, we could connect multiple VIP devices to provide higher performance through parallel execution. This will not require a significant change to software for VIP either – it is already written in a parallel fashion, with each PE accessing its own local vault most of the time.

Merely scaling up the number of PEs in each vault may not be as effective. Increasing the number of PEs will move the compute roofline upwards, workloads will now be bottlenecked by the memory bandwidth unless we can provide a method to increase the arithmetic intensity. We cannot increase the arithmetic intensity of BP-M, as it is a streaming application. CNNs, however, have potential for data reuse, as described in section 7.2.1. In the present implementation, filters are stored in the SRAM scratchpad and applied to multiple input feature data. PEs within a vault, however, must stream in the same input features, increasing the memory bandwidth requirement unnecessarily. Further, the number of filters that may be stored in the scratchpad is limited by the size of the scratchpad itself. Increasing the scratchpad size can also increase data reuse in CNNs. This, however, comes at the cost of increased area require-

ments. Presently, the 4 KiB scratchpad occupies 49.5 % of the area of a VIP PE, increasing the scratchpad capacity will increase the area proportionally.

VIP was designed primarily for BP, which is a streaming workload with very little data reuse. As a result, VIP forgoes caches as they would add area with little benefit. However, memory-side caches inserted at each HMC vault may potentially be useful for CNNs, as the caches will act as a coalescing buffer when multiple PEs request the same set of input features. This can potentially reduce the DRAM accesses saving energy required to access DRAM. This, however, will probably not lead to any performance improvements without increasing both the computational throughput of the system as well as the network bandwidth.

While it may be tempting to assume that the same computation throughput may be achieved using fewer PEs with a wider vector datapath, this approach will not work in practice. First, a wider datapath may lead to wasted ALUs in the vector pipeline if the length of the vector being processed is shorter than the width of the pipeline. Second, software data prefetching must be made even more aggressive in order to hide DRAM latency. Consider the case when the width of the vector datapath is increased 4×, to 256 bit instead of the present 64 bit. In this scenario, computing eq. (4.1a) in depth-from-stereo with 16 labels will require three vector additions, each taking exactly one clock cycle. Additionally, bubbles must be introduced into the pipeline to handle the read-after-write dependencies between these instructions. Similarly, eq. (4.1b) will require 16 cycles to execute, with another cycle required for a normalisation step to ensure that values do not go out of range. DRAM latency for accessing data may be as large as 240 cycles, implying that loads must be issued at least 12 loop

iterations ahead of when data will be used, this will also fill up the 20-entry array range check (ARC), making it a bottleneck that serialises loads from DRAM. The effect on CNNs, however, will be exactly the opposite. If we combine the vector datapaths and the scratchpads of the four PEs within a HMC vault, the additional scratchpad capacity will allow for more filters to be stored in the scratchpad, improving data reuse and increasing the arithmetic intensity. This is attributable to a fundamental difference between BP and CNNs – BP requires a microarchitecture capable of working efficiently with short vectors and low arithmetic intensity, while CNNs use longer vectors with more data reuse requiring lower arithmetic intensity. The same difference also carries in to the behaviour of the DRAM memory system, as described in section 7.2.3.

Section 5.1 discussed how the ARC structure may be used to detect and mitigate all hazards in the vector pipeline at the cost of increased power. As loads are issued once in many cycles, VIP presently has to check the ARC a few times in many cycles. Using the ARC to detect all scratchpad hazards will necessitate multiple checks every cycle, resulting in increased area and power for the ARC. Additionally, it is not clear from RTL results if the ARC can be extended to support the additional entries and read ports required by this change without violating timing constraints.

Finally, VIP's ISA may be extended by adding functional units within the same basic vector unit structure. Some candidates include support for floating point maths, and units for fast transcendental functions such as exponents, logarithms, and trigonometric functions. Adding these functional units will increase area but not change the programming model. In fact, RNNs necessitate the addition of units for the logistic function as this is used for activations

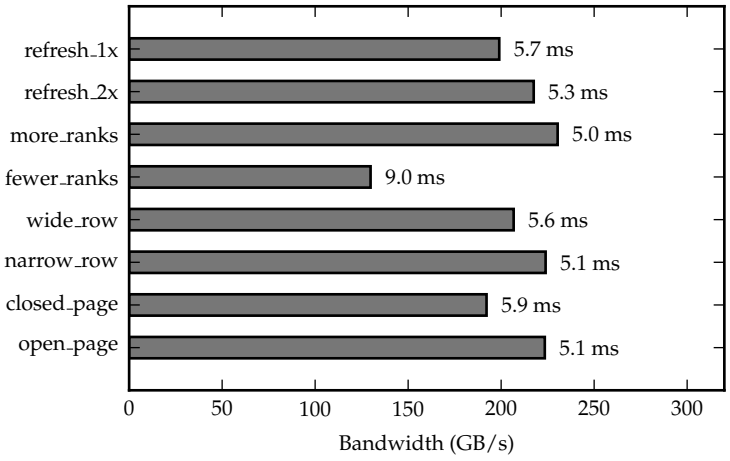
instead of ReLU. This may be achieved as quadratic interpolation using fast lookup tables as described by Oberman and Siu [95].

7.2.3 Sensitivity to Memory Parameters

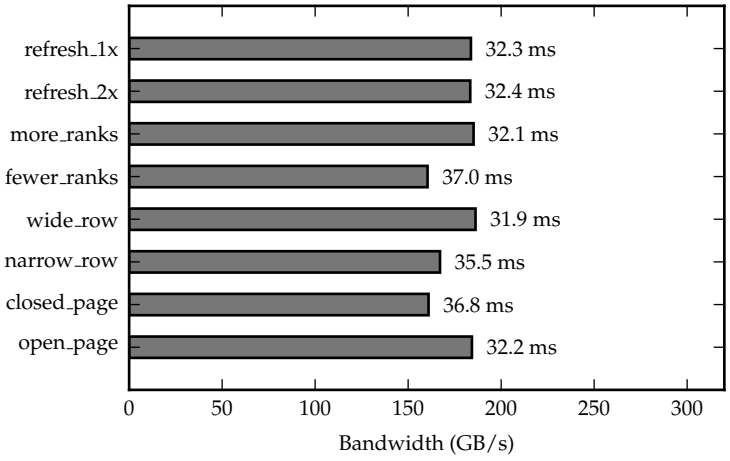
This work assumes that the memory system configuration is very similar to the HMC with some modest changes (e.g.: open-page policy and faster refresh). This section explores the performance of VIP under different parameter choices.

Starting with the configuration shown in table 7.1 ('open_page'), we can change the row-buffer policy to closed page ('closed_page'). To simulate the effect of varying memory parallelism, we may again start with the open_page policy, and reduce and increase the number of ranks (the HMC has one bank per rank, so these terms are used interchangeably) by $4\times$ (we increase and decrease the number of rows per bank to keep the DRAM size constant) ('fewer_ranks' and 'more_ranks'). The effect of wider and narrower rows may be studied by increasing and decreasing the width of the DRAM rows (decreasing and increasing the number of rows) by $4\times$ ('wide_row' and 'narrow_row'). Finally, to study the effect of changing refresh policy, we may start with the open_page policy (which implicitly uses the refresh_4x mode described in the JEDEC DDR4 standard [2, Table 24]) and study the effect of increasing both t_{RFC} and t_{REFI} by $2\times$ and $4\times$ ('refresh_2x' and 'refresh_1x'). Figure 7.2 shows the achieved memory bandwidth (and execution time) for the end-to-end benchmarks – one iteration of full-HD BP-M and the entire VGG-16 network under these configurations.

A closed page policy instead of an open page policy has a detrimental effect on the effective memory bandwidth, which confirms the intuition in section 5.2.



(a) Belief propagation



(b) CNN VGG-16

Figure 7.2: Memory bandwidth (and execution time) for BP on a full-HD image and the VGG-16 [112] network, end-to-end (including the convolution, rectified linear unit (ReLU), pooling, and fully-connected layers) under various memory configurations obtained by tweaking the default HMC configuration. Open page refers to the configuration in table 7.1. Closed page is the same configuration, but with a closed page policy. Wide and narrow rows refers to the open-page configuration with DRAM rows $4\times$ wider and narrower, similarly fewer ranks and more ranks refers to $4\times$ fewer and more ranks per vault. Also shown is the effect of increasing both t_{REFI} and t_{RFC} by $2\times$ and $4\times$ respectively.

Decreasing the number of ranks in the system causes the memory bandwidth to drop and the execution time to increase. This is not surprising – increased memory-level parallelism allows more DRAM requests to be in flight at any time, increasing the achievable bandwidth. Increasing the refresh interval (and consequently the refresh cycle time) significantly decreases memory bandwidth and increases execution time for BP, while CNNs are affected to a much lesser degree. This is attributed to the fact that CNNs access memory through few, large requests, while BP accesses memory through many small requests. As a result, BP is more sensitive to memory system stalls. Similarly, CNNs show higher bandwidth with wider rows, while BP shows a higher bandwidth with narrow rows. Again, this is attributed to the nature of DRAM requests – CNNs issue few large requests, while BP issues many small requests. Narrow rows result in data being scattered across multiple rows, necessitating the row to be closed and reopened when another PE requests the same data.

7.2.4 RTL Results

We can estimate the area and power required by VIP by synthesising RTL code for a single PE. The RTL code is verified through unit tests as well as integration tests that run small example kernels for both BP and CNNs. Switching activity factors from these RTL simulations are supplied to Synopsys PrimeTime to estimate the power required by a VIP PE. The RTL design synthesis estimates that a single PE requires 0.141 mm^2 in silicon area after place-and-route, and consumes 27 mW and 38 mW of power for BP and CNNs respectively. (CNNs require more power than BP as they use multipliers, while BP does not.) 128 PEs will therefore occupy 18 mm^2 in silicon area and consume 3.5 W to 4.8 W of

Table 7.4: Area required by various components in a VIP PE, as reported after place-and-route.

Component	Area (μm^2)	Percentage
Fetch	16892	12.0
Decode	990	0.7
Register File	15944	11.3
Issue	2147	1.5
ARC	5365	3.8
Scalar	2547	1.8
Vector-control	554	0.4
Vector-vertical	16355	11.6
Vector-horizontal	5030	3.6
Scratchpad	69661	49.5
Load-store	3274	2.3
Other	1832	1.3
Total	**140592**	**100.0**

power, in addition to the power consumed by the 3D-stacked DRAM memory. Figure 7.3 shows a single VIP PE after place-and-route. Table 7.4 shows the area occupied by each component in VIP. Most of the area of a VIP PE comes from the SRAMs in the scratchpad, the register file, and the instruction buffer. Most of the area of the vertical vector unit comes from the pipelined multipliers. The unit responsible for horizontal vector reduction operations requires significantly less area, at just 3.6 % of the total area of the PE.

These area and power numbers presented do not account for the logic required to support logistic functions for RNNs activation functions. Logistic functions could be implemented either as lookup tables or through quadratic interpolation described by Oberman and Siu [95]. If these logistic functions are implemented as lookup tables, the number of entries in these lookup tables will depend on the amount of error that will be tolerated by the RNN, but could range from 64 (used by Nurvitadhi et al. [91]) to 2048 (used by Han et al. [46]).

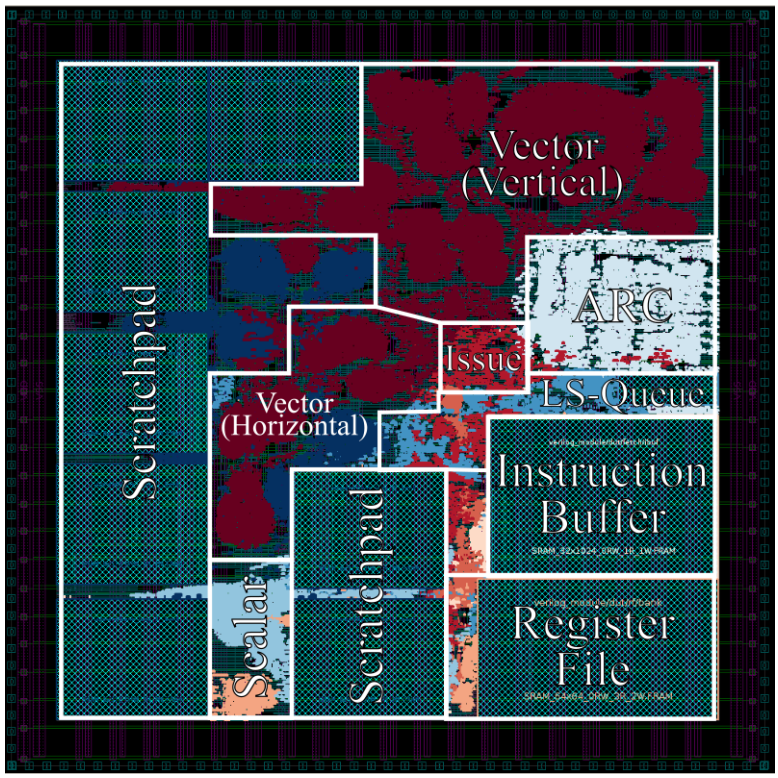


Figure 7.3: The generated layout of one VIP processing engine (PE).

In addition, VIP's on-chip network as well as the HMC memory system will also consume power. As discussed in chapter 6, software is written so that the on-chip network is very lightly loaded, so the main contribution for power will be from the HMC itself. Jeddeloh and Keeth [56] report the power of an early HMC prototype at 10 pJ bit^{-1} in 50 nm technology. At full bandwidth (320 GB s^{-1}), this translates to a power of 25.6 W for the HMC. Jacob et al. [54] report the DRAM power of the IBM Active Memory Cube (which uses a HMC as well) at approximately 5 W using a 14 nm process. Therefore, the total power required by VIP will be 8 W to 30 W.

CHAPTER 8

RELATED WORK

There are a number of related work, both in the field of processing in memory (PIM) as well as in developing architectures for belief propagation (BP) on probabilistic graphical models (PGMs), convolutional neural networks (CNNs), and multi-layer perceptrons (MLPs).

8.1 Prior Work in Processing-In-Memory

Early PIM concepts – first LSI memories The concept of PIM isn't new. The oldest references in literature are from the late 1960s and early 1970s, when memory saw a transition from magnetic core to transistor-based memory. At that time, the complexity of electronic chips was dominated by the pin count instead of the transistor count, thereby creating for a need for chips that could do more with less pins. At this point, there were proposals for augmenting the electronic cache chips with additional logic to perform some tasks. For instance, Levitt and Kautz [76] proposed adding logic to the cache chips for error-correcting codes. Kautz [63] advocated instead for developing memory as logic-and-storage circuits with 10–50 gates each that could be programmed to serve various functions. In this way, through programming each cell to serve different functions, the memory as a whole may be used to achieve arbitrarily complex tasks, such as a memory that keeps all words sorted, as a stack, a heap, or as content-addressed memory. Stone [116] motivated his work as trying to increase the complexity of large-scale integrated circuits in the cache memories while keeping pin counts (and therefore the complexity and costs of modules) low. He proposed a number of operations that may be executed by such a logic-

in-memory cache, such as searching for values that match certain criteria (such as equality, inequality), modifying results of matched searches (such as copying over tag bits that indicate results, or performing logic operations such as AND over tag bits), adding words present in two different sectors in memory, or multiplying every element in a sector of memory.

First generation of PIMs – planar chips There's little work on PIM from the 1970s until the early 1990s. Elliott et al. [25] presented work on Computational RAM. They argued that the memory bandwidth available at the sense amplifiers in a contemporary DRAM system was 1.1 TB s^{-1}, then throttled by limited pins on DRAM devices and on the system bus down to 33 MB s^{-1}. Even with aggressive caching, a contemporary cache only provided 130 MB s^{-1} of memory bandwidth, which was orders of magnitude less than the memory bandwidth available at the DRAM sense amplifiers. They developed a proof-of-concept chip which has 8 Kibit of SRAM and 64 processing elements. The authors specifically targeted such a chip towards digital signal processing, such as computing the discrete cosine transform typically used in image processing, and 2D convolutions. Kogge et al. [67] present EXECUBE, a system that ties in eight processing elements inside a conventional DRAM die. Each processing element has 64 kB of its own memory, a 16 bit processor, and can communicate with other devices through off-chip links. The processing elements can work either decoupled (in MIMD mode) or coupled (SIMD mode) wherein an external controller broadcasts instructions to these processing elements. The EXECUBE is designed so as to have multiple devices running in parallel, and a real hardware implementation consisting of 64 chips was developed. The EXECUBE, the authors claim, can execute 50 MIPS while consuming 2.7 W. Kogge et al. [68]

further discuss this paradigm, which they term 'Shamrock', which consists of tiles of CPU and DRAM memory laid out on a die. The chip is designed to function in one of three modes, an accelerator model – the PIM is designed to perform certain tasks offloaded by a CPU, massively parallel peer model – the PIM chips are networked and work independently without a host, and an active memory management model – the PIM chips perform some operations for managing the memory hierarchy in a computer. This is particularly interesting taxonomy as it can be seen in PIM architectures even today. Murphy et al. [86] propose using Data Intensive Systems benchmark suite instead of the then conventional SPEC'95 suite. Their reasoning was that SPEC has fairly regular memory accesses, whereas the Data Intensive Systems suite does not. Of particular interest in the Data Intensive Systems suite is a simplified object-oriented database which supports insert, delete, and query operations, although the paper only focuses on the query operation. This system closely couples a vector register file with the row buffer in DRAM. The 2 Kibit row buffer provides 256 bit as a vector to be stored in the vector register file.

Gokhale et al. [41] present Terasys, a massively parallel PIM array that could be used as memory or as a PIM. Their system consists of SRAMs 2048×64 bit, with 64 bit-serial ALUs (each ALU operates on one bit of memory). Many such PIM units can be tied together to create a system with 32 k single-bit ALUs and 8 MiB of address space. The authors also develop a data-parallel bit C language for programming such a system.

Two well known works in PIM include FlexRAM by Kang et al. [61], and IRAM by Patterson et al. [99]. IRAM advocates for putting in vector processors inside DRAM. Specifically, Patterson et al. [99] cite the advantages of a vector

processor as not requiring cache due to their generally operating on an entire memory block, and accessing data in fairly uniform strides. Some of the numbers mentioned in this work are significantly large for their times. For instance, it is claimed that IRAM would offer 100 GB s^{-1} of memory bandwidth, be able to support thirty two 64-element vector registers, and have support for floating point and integer operations. This in contrast to IBM's Active memory cube [88], almost two decades later, which also sports fairly similar numbers, albeit with many more vector processors and achieves a bandwidth of 320 GB s^{-1} through 3D stacking. FlexRAM by Kang et al. [61] follows a different approach. It uses an array of 64 single program multiple data (SPMD) processors, each processor capable of accessing 1 MB of DRAM and communicating with two neighbours. A two-issue RISC processor is also provided for broadcast and reduction operations.

Hall et al. [45] present DIVA, a different take on the PIM paradigm. In this case, the authors put in a node processor, but also include another processor that works at the sense amps. The processor at the sense amps can perform wide bit-operations such as searches, pattern matching, and even limited pointer chasing along with associative-and-commutative reduction operations. PIMs in DIVA communicate on a bus separate from the host bus, therefore do not cause traffic on the host bus. Memory on the PIM may be used either as dumb memory – wherein it is used only by the host processor and not by the processors in memory. It may also be used as PIM-only memory – accessible only to the PIM processors but not to the host, or as global memory – accessible to both the host and PIM processors. Oskin et al. [96] present Active pages, a different model that seeks to offload intensive computation to memory. The authors use reconfigurable-architecture DRAM (RADram) which integrates FPGAs with

DRAM. The use of FPGAs as opposed to processors allows for a greater deal of flexibility in terms of the operations that may be supported. The authors show how Active pages can be used to reduce the overhead of array insertion and deletion to $\mathcal{O}(1)$ by moving array elements in parallel. They also show implementation of database query operations, and implement MMX primitives on Active pages.

Another interesting development during this period is the Impulse memory controller by Zhang et al. [137]. Technically, this is not a PIM system, it is a memory controller that maps the unused memory address space to perform interesting operations on data. For instance, if the memory address space can address 4 GiB but only 1 GiB is installed, it can map the remaining 3 GiB of space to aliases within the installed memory. A system could use this to improve cache locality, for instance, by transposing a matrix at the memory controller.

Second generation of PIMs – 3D stacked chips The next wave of PIM research starts with 3D stacked designs that make it possible to integrate modern logic and memory systems through the use of through-silicon vias (TSVs). One such example (discussed extensively in this work) is Micron's hybrid memory cube (HMC) [53] that abstracts away the DRAM interface. Loh et al. [81] argue that this constitutes processing in memory. The HMC provides support for a very limited number of atomic memory operations, which is why it features in this list. Zhu et al. [140] present a framework for generating ASICs compatible with 3D stacked DRAM. A follow-up [141] implements ASICs for Sp-GEMM (used in graph processing) and 2D FFT. Guo et al. [44] present a system where the logic die on a Hybrid memory cube consists of a number of domain-specific accelerators. The system can then be programmed to create a pipeline of these

accelerators to solve an interesting problem. Pugsley et al. [101] present a system that uses low power ARM Cortex A5 like cores in the logic layer of a Hybrid memory cube, with a MapReduce programming paradigm. Ahn et al. [4] follow a similar paradigm, using single-issue in-order cores within the logic layer of a HMC, along with some additional hardware for efficient message passing between cores. They also use two kinds of prefetchers to hide the memory latency, a conventional list prefetcher, and another prefetcher that is triggered on receiving a message from a core. They target their system for processing graphs, although they limit their evaluation to fairly simple kernels. Nai and Kim [87] study breadth-first search graph traversals and show how the HMC 2.0 standard with its atomic compare-and-swap instruction can significantly reduce the bandwidth required for graph traversals. Their argument is that graph traversal is an irregular workload that does not benefit through caching. By offloading this to the memory system, they can reduce the amount of data movement significantly. Ahn et al. [6] propose another system, one that develops special instructions that may be executed on the memory system. When a processor encounters such an instruction, it offloads the instruction to the memory. However, this system relies on the host processor translating the virtual addresses to physical addresses, which may potentially make the host processor a bottleneck when issuing multiple instructions to multiple PIM devices. Moreover, the number of PIM enabled instructions is limited.

Farmahini-Farahani et al. [31] also use 3D stacking for near-data processing; instead of using a 3D stacked memory system such as the Hybrid memory cube, they stack CGRAs with commodity 2D DRAM using TSVs. Akin et al. [7] present a system for reorganizing data in memory, again using 3D stacked DRAM and near-data processing. This is interesting because it brings in similar

functionality to the Impulse memory controller [137].

Nair et al. [88] present Active Memory Cube, a design by IBM in order to reach exascale computing at less than 20 MW. They put in vector processors at the logic layer of a HMC. The architecture of the Active Memory Cube has been discussed extensively in this work. Nair et al. target applications typically used in supercomputers, such as DAXPY and DGEMM, along with some other applications [89]. The particularly interesting aspect of this work is that it revisits near-memory computing two decades after Gokhale et al. [41] used a similar argument for surmounting the then barrier for supercomputing, petascale computing.

Jayasena et al. [55] argue that PIMs will benefit most from heterogeneity. While memory-bound applications benefit from in-memory execution, compute-bound applications take longer to run (as in-memory processors must be designed to be energy efficient) compared to a host that would have a CPU and GPU on the same die. This follows a trend of papers from AMD research, such as work by Zhang et al. [136], which advocates a similar approach of placing GPU compute units within the logic die of a 3D stacked memory similar to the Hybrid memory cube. Their performance model for the PIM system is computed by scaling performance and power on existing GPUs. Xu et al. [135], again from AMD research, discuss the performance of Accelerated compute units (similar to the PIMs considered by Jayasena et al.) consisting of CPU+GPU for deep learning.

Xi et al. [133] present JAFAR, a system that exists as a discrete IC mounted on a DDR3 DIMM for filtering database queries. The system as presented works only for integer comparisons, but the authors claim that it could be extended.

When a select operation is sent to a database, the system only returns matched queries.

Azarkhish et al. [12] study the effect of address scrambling on the performance of various graph and PARSEC benchmarks operating under a PIM environment through the study of program traces, although they do not specify what their 'PIM clusters', the units that actually perform computation, look like. Another work by the same authors [13] considers the case when the PIM unit is an ASIC optimized for neural networks.

Gao et al. [37] develop a system similar to the one proposed by Ahn et al. [4] in that it uses in-order single-issue cores, similar to ARM Cortex A7. They evaluate performance on various workloads including MapReduce, graph, and deep neural networks. Another work by Gao and Kozyrakis [38] proposes a heterogeneous reconfigurable logic that provides both coarse-grained functional units and fine-grained logic blocks to serve as a bridge between CGRAs and FPGAs, and evaluate its performance on MapReduce, graph, and deep neural network benchmarks.

Morad et al. [84] propose a different approach to computations using ReRAM technology. They propose using SIMD like processing units much like PIM technology that would use any other memory technology such as SRAM or DRAM. However, unlike SRAM or DRAM, the authors note that ReRAM crossbars are typically created in the higher metal layers of a chip, while the lower layers are available for CMOS logic. The authors are thus able to create a chip that has both compute and memory on the same die without using 3D stacking.

Processing-using-memory An alternate form of in-memory processing is not to add additional logic units to memory, but to use properties of the memory itself in order to perform computation. This is a paradigm called processing-using-memory rather than processing-in-memory. For instance ReRAMs are gaining popularity in approximate computing. The grid structure of ReRAM arrays allows for performing approximate matrix-vector multiplications by summing currents through the array [43]. This approximate matrix-vector multiplication is useful in the evaluation of neural networks, which has prompted a number of publications in the field, including Bojnordi and Ipek [15], Chi et al. [18], Hu et al. [51] and Shafiee et al. [110]. These are discussed in section 8.3. Finally, Gao et al. [36] present a DRAM based reconfigurable fabric that replaces the SRAM based lookup tables in FPGAs. They perform some optimizations to hide the larger latencies of DRAM accesses while reducing power.

8.2 Prior Work in Vector Computing

Early vector processors The earliest vector processors were the CDC STAR-100 and the Cray-1 [108]. These computers developed using scalar pipelines, and vectorisation was used as a tool to amortise the long startup latencies. The CDC STAR-100 used a vector memory-memory processing paradigm, which meant that the operands as well as the results would be streamed in from and out to main memory, which consisted of ferrite core. The STAR-100 also had virtual memory support, which was unusual for supercomputers of the era, as well as support for vector reduction instructions in its instruction set. However, the scalar performance of the STAR-100 was fairly low; Amdahl's law kicked in resulting in lower than expected performance for most applications. The Cray-

1 on the other hand used a vector-register paradigm, the pipelines operated on data stored in vector registers instead of main memory. The programmer had to move data between memory and registers. The vector-register paradigm has a distinct advantage in reducing memory bandwidth required as intermediate data may be stored in vector registers instead of being written back to memory. Additionally, the latency of accessing registers was just one cycle, whereas accessing memory required 11 cycles. The Cray-1 allowed vector operations to be chained, i.e. connecting the output of one vector pipeline to the input of another vector pipeline, potentially executing two operations every clock cycle.

Multi-pipeline vector processors While the CDC STAR-100 and the Cray-1 were essentially scalar pipelines that used vector processing to amortise the startup overheads of these pipelines and parallelising execution in time, vector processors introduced in the 1980s by Fujitsu, Hitachi, and NEC used multiple parallel datapaths to parallelise program execution in both time and space. For example, the Hitachi S-820 had an add-logical pipeline consisting of four fully segmented pipelines, a multiply-add pipeline again consisting of four fully segmented pipelines, and a division pipeline consisting of one fully segmented pipeline [64]. The Fujitsu VP2000 series introduced a variable vector register configuration, the compiler or programmer could choose a register file configuration that varied the number of vector registers between 8 and 64, corresponding to a vector length between 2048 B and 256 B respectively [122]. Similarly, the NEC SX-2 also had four 4-wide vector pipelines [32].

Single-chip vector processors and later efforts Vector processors described in the previous paragraphs were typically implemented as multiple ECL chips.

Asanović [10] proposed T0, a single-chip vector processor in CMOS technology. T0 evolved into Spert-II [128], developed as a computer for accelerating neural networks. Scale [73] proposed a new vector-thread paradigm that unifies vector and multi-threaded execution. Maven [14] was a single-lane vector-thread architecture which was simpler to implement and had similar energy efficiency as a multi-lane implementation.

On the other end, work by Espasa and Valero [28, 29] and Espasa et al. [30] focussed on improving the performance of vector processors by allowing them to hide latency using similar techniques as microprocessors. These included decoupling access and execute, and adding multithreaded and out-of-order execution to vector processors. These techniques were already successfully used by microprocessors for hiding memory latency, supporting these in vector processors was tricky as the vector register file became more complex and these techniques relied upon handling data dependencies and precise exceptions correctly.

Espasa et al. [27] and Quintana et al. [102] advocate adding vector coprocessors to existing superscalar processors. As these vector units have multiple pipelines (spatial SIMD), they must communicate with a special high-bandwidth cache so that data movement does not become a bottleneck. To this end, both these works propose special vector caches or modifications of the existing L2 cache to achieve this high bandwidth.

Vector IRAM [71] took a different approach to providing high bandwidth to vector processors, coupling processors with embedded DRAM on the same die. CODE [72] tackled the increasing complexity of vector register files by using a clustered vector register file approach, dividing the vector processor into

clusters with local register files. Clusters are designed to access only their local registers, and registers may be moved between clusters using an on-chip network. This network may be designed to be either simple or complex depending on the implementation point. This system allows each individual register file to have fewer ports.

Another related system is the IBM Cell processor [60], which shares some characteristics with VIP, although there are some key differences between the two. Cell employs a SIMD processing paradigm for its synergistic processing elements, which have a 128 bit datapath. These processing elements operate on a 256 KiB local storage that contains both instructions as well as data. The local storage, however, is very different from VIP's scratchpad. While VIP's scratchpad is controlled using the array range check (ARC) unit in a way that is invisible to the programmer, Cell uses a DMA engine to move data between local storage and main memory. Additionally, Cell's synergistic processing elements internally use SIMD registers that are loaded from local storage, which has only one read and one write port. Essentially, the local storage in Cell operates as a software managed cache for the synergistic processing element, whereas, in VIP, it operates in lieu of a register file. Additionally, Cell does not provide support for arbitrarily long vectors or matrix-vector operations in its ISA; as discussed in chapter 4, these are critical for efficient performance on the applications considered in this work.

8.3 Prior Work in Hardware Architectures

ASIC Implementations Prior work involving ASIC for BP includes work by Cheng et al. [17] and Liang et al. [77] using a tile-based BP algorithm, which stores only messages at the edge of a tile, and recomputes messages within a tile through multiple BP-M iterations within the tile. Their system can achieve 30 fps for 720p video, but performs only one effective iteration. Work by Tseng and Chang [121] uses a 'spinning message' update, based on the message storage scheme for bipartite graphs proposed by Felzenszwalb and Huttenlocher [33]. ASIC implementations for CNNs include Eyeriss by Chen et al. [16], which has been discussed extensively in previous chapters and served as a baseline CNN accelerator. Tetris by Gao et al. [39] extends Eyeriss by coupling the Eyeriss architecture with 3D stacked memory similar to a HMC. Tetris' results are presented only with regards to a hypothetical baseline, and are therefore hard to compare against. Other work includes Neurostream by Azarkhish et al. [13] and Neurocube by Kim et al. [65]. Both these works involve multiply accumulate (MAC) units on the logic die of a HMC, supplied data by finite state machines that generate address request streams to DRAM. Cnvlutin by Albericio et al. [8] saves computation by skipping multiplications where one of the operands is a zero. SCNN by Parashar et al. [97] goes further by storing the weights and activations in a sparse-compressed format. Cambricon by Liu et al. [80] presents itself as an ISA for neural networks, but it dedicates almost all its hardware resources to matrix-matrix multiplication using MAC units. The tensor processing unit (TPU) by Jouppi et al. [59], deployed in Google's datacentres, makes a similar design choice. Not only can these systems not work with min-sum BP due to their reliance on the MAC primitive as a building block,

their lack of resources towards element-wise vector operations means that they will struggle to provide good performance on adding multiple incoming messages, a crucial part of min-sum BP. PuDianNao by Liu et al. [79] is a machine learning accelerator for multiple machine learning kernels. PuDianNao offers a pipelined, fixed-function implementation of parallel and reduction operations. VIP, on the other hand, leaves the choice of these operations to the programmer thereby providing more flexibility than a datapath with fixed functional units. Wawrzynek et al. [128] developed Spert-II in the 90s, adding vector instructions to the MIPS ISA for training and inference on neural networks. Their design used 16-bit vector elements, and was 30 times faster than a SPARC-20 workstation at the time. Venkataramani et al. [123] propose ScaleDeep, a system that allows the execution of neural networks at scale, such as in a data-centre. Their approach is a modular one – providing both compute-heavy and memory-heavy tiles which are optimized for their respective tasks, which are composed to create a big system.

FPGA Implementations Prior work involving FPGA implementations for BP include work by Park et al. [98] that uses 320 parallel processing engines spread across two Xilinx Virtex-II FPGAs. Lin et al. [78] use the block RAMs (BRAMs) inside the FPGA to provide high-bandwidth memory. Their solution, however, does not work for graphs with loops and therefore cannot be used for vision applications. Additionally, it is limited by the size of BRAMs inside the FPGA. Severance and Lemieux [109] develop Venice, a soft FPGA-based vector processor. As a soft accelerator on an FPGA, Venice runs at a low clock frequency and therefore cannot provide the necessary throughput. Unlike VIP where data transfers are invisible to the programmer, Venice relies on a DMA controller to

move data between DRAM and the BRAMs inside the FPGA. Wei et al. [129] use systolic array for CNNs that achieves comparable performance against VIP. Nurvitadhi et al. [92] and Zhao et al. [138] employ binarised neural networks, i.e. neural networks where weights are expressed as binary values instead of as real numbers, and map these to FPGA platforms. Aydonat et al. [11] use OpenCL to map deep neural networks (DNNs) to an FPGA platform, their work uses special Winograd transformations to reduce the computational complexity of the convolution operation. Han et al. [46] propose ESE, an FPGA system for sparse long short-term memory (LSTM) networks, which has also been extensively discussed as a baseline architecture for recurrent neural networks (RNNs). Mahajan et al. [82] present Tabla, a system that can generate FPGA accelerators for a class of machine learning algorithms that employ stochastic gradient descent, while Sharma et al. [111] present DnnWeaver, a system that generates FPGA implementations for DNNs expressed in the Caffe framework [57]. Tabla focuses on training, not inference, while DnnWeaver works on DNNs, not BP. Brainwave by Chung et al. [22] stores neural network parameters within the BRAMs within FPGAs, using data-centre scale deployment for large models. Brainwave presently does not support BP. Even if support for BP were added to Brainwave, this model may be inefficient for very large data sizes, e.g.: BP-M on a full-HD image requires over 300 MB of storage ($\sim$30 FPGAs).

Non-conventional Devices Wang et al. [127] propose a system that uses optical resonant Gibbs sampling units for Markov random field (MRF) inference. Unlike BP, their approach uses Gibbs sampling. As a result, they require many more iterations than VIP, which greatly increases execution time (e.g.: 1100 ms per frame for full-HD video with just five labels). Work by Bojnordi and Ipek

[15], Chi et al. [18], Shafiee et al. [110] and Song et al. [114] uses ReRAM crossbar arrays to perform dense matrix multiplications for DNNs. These do not work for BP.

CHAPTER 9

CONCLUSIONS

9.1 Summary and Contributions

This work presents a system – VIP (Versatile Inference Processor) – for fast inference in machine learning algorithms such as probabilistic graphical models (PGMs), convolutional neural networks (CNNs), multi-layer perceptrons (MLPs), and recurrent neural networks (RNNs). Unlike prior work, VIP achieves high programmability, competitive performance, and low power. It achieves this through three mechanisms – 1. using a vector processing paradigm that is well understood by hardware and software engineers alike; 2. presenting a paradigm that allows a vector processor to perform matrix-vector (and vector reduction) operations without making assumptions of the composition of operators within these operations 3. coupling multiple, independent processing engines (PEs) with 3D stacked DRAM memory in order to supply high bandwidth access to data.

VIP's development was motivated by a number of key observations about the benchmarks considered.

Algorithms are in constant development The applications considered have algorithms that are in constant development. Sufficient care must be taken to ensure that any accelerator designed for these applications can adapt to changing state-of-the-art algorithms. For instance, in our case study with Hierarchical TRW-S, adding a hierarchical modification to the algorithm resulted in significant improvement in performance. Exploiting this performance, however,

required that new functional units be written and integrated with prior work by Choi and Rutenbar [19, 20]. If their accelerator were developed as an ASIC instead of on an FPGA, we would be out of luck. It is critical, therefore, to have accelerators that can be programmed to support not just the algorithms known today, but any future developments as well.

Memory bandwidth is critical The applications and algorithms considered all have very high memory bandwidth, and very low arithmetic intensity, much lower than that required by contemporary CPUs and GPUs for good performance. As a result, new systems must be designed in a manner that they are able to work efficiently with the low arithmetic intensity of the applications considered. Modern 3D stacked DRAM and processing in memory (PIM) systems are promising in this regard.

Vector processing is a good fit The vector processing paradigm is an excellent fit for the machine learning benchmarks presented in this work. However, we must adopt the right paradigm for vector processing. VIP works particularly efficiently by recognising that almost all these algorithms consist of vector operations immediately followed by reduction, and that these operations extend beyond the multiply accumulate (MAC) unit that is prevalent in scientific computing workloads. Additionally, VIP ensures that its vector units remain busy by allowing the program to set up subsequent loop iterations while the vector unit is busy with a long-latency operation.

9.2 Future Work

There are a number of possible extensions to VIP that may be incorporated in the future. First, VIP may be augmented with additional functional units in the vector unit to support a broader class of applications. For instance, adding support for RNN requires adding a unit to compute the logistic function. Other functional units that may be added include exponents, logarithms, and trigonometric functions.

VIP has been designed for accelerating the computation involved in inference on PGMs; it could be augmented with additional support for traversing irregular graphs. There are a number of hardware architectures designed for accelerating graph traversals, but they do not include support for vector processing required by PGMs. Integrating graph processing accelerators with VIP can bridge this divide, offering competitive performance on more general PGM applications.

We observed, in section 7.2.1 that VIP does not perform as well as ESE as it is unable to exploit sparsity. At the moment, VIP has been designed only for dense matrix-vector algebra. It is possible to extend VIP's model to be truly versatile by adding support for arbitrary dimensional tensors as well as for sparse tensors.

Other extensions of this work include development of a high-level programming model for VIP and an associated compiler. In this regard, a domain-specific language or framework such as TensorFlow [3] or Halide [103] may be useful in such development and mapping additional applications to VIP.

BIBLIOGRAPHY

[1] JC-42.3. *High bandwidth memory DRAM*. Tech. rep. JESD235A. JEDEC, Nov. 2015 (cit. on pp. 38, 57).

[2] JC-42.3C. *DDR4 SDRAM*. Tech. rep. JESD79-4B. JEDEC, June 2017 (cit. on pp. 60, 99).

[3] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu and X. Zheng. 'TensorFlow: a system for large-scale machine learning'. In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2016, pp. 265–283 (cit. on pp. 78, 122).

[4] J. Ahn, S. Hong, S. Yoo, O. Mutlu and K. Choi. 'A scalable processing-in-memory accelerator for parallel graph processing'. In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM, 2015, pp. 105–117. DOI: 10.1145/2749469.2750386 (cit. on pp. 40, 58, 77, 109, 111).

[5] J. Ahn, S. Yoo and K. Choi. 'Dynamic power management of off-chip links for hybrid memory cubes'. In: *Proceedings of the The 51st Annual Design Automation Conference*. ACM Press, 2014, pp. 1–6. DOI: 10.1145/2593069.2593128 (cit. on p. 58).

[6] J. Ahn, S. Yoo, O. Mutlu and K. Choi. 'PIM-enabled Instructions: a low-overhead, locality-aware processing-in-memory architecture'. In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM, 2015, pp. 336–348. DOI: 10.1145/2749469.2750385 (cit. on pp. 77, 109).

[7] B. Akin, F. Franchetti and J. C. Hoe. 'Data reorganization in memory using 3D-stacked DRAM'. In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM Press, 2015, pp. 131–143. DOI: 10.1145/2749469.2750397 (cit. on p. 109).

[8] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger and A. Moshovos. 'Cnvlutin: ineffectual-neuron-free deep neural network computing'. In: *Proceedings of the 43rd Annual International Symposium on Computer Architecture*. IEEE, June 2016, pp. 1–13. DOI: 10.1109/ISCA.2016.11 (cit. on p. 116).

[9] *ARM architecture reference manual supplement: the scalable vector extension (SVE), for ARMv8-A (Beta)*. Tech. rep. DDI 0584A.a. ARM, 2017 (cit. on pp. 43, 44, 51, 76).

[10] K. Asanović. 'Vector microprocessors'. PhD thesis. University of California, Berkeley, 1998 (cit. on p. 114).

[11] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling and G. R. Chiu. 'An OpenCL deep learning accelerator on Arria 10'. In: *Proceedings of the 25th ACM/SIGDA International Symposium on Field-programmable Gate Arrays*. 2017, pp. 55–64. DOI: 10.1145/3020078.3021738 (cit. on p. 118).

[12] E. Azarkhish, C. Pfister, D. Rossi, I. Loi and L. Benini. 'Logic-base interconnect design for near memory computing in the smart memory cube'. In: *IEEE Transactions on Very Large Scale Integration Systems* 25.1 (Jan. 2017), pp. 210–223. DOI: 10.1109/TVLSI.2016.2570283 (cit. on pp. 77, 111).

[13] E. Azarkhish, D. Rossi, I. Loi and L. Benini. 'Neurostream: scalable and energy efficient deep learning with smart memory cubes'. In: *IEEE Transactions on Parallel and Distributed Systems* 29.2 (Feb. 2018), pp. 420–434. DOI: 10.1109/TPDS.2017.2752706 (cit. on pp. 80, 111, 116).

[14] C. Batten. 'Simplified vector-thread architectures for flexible and efficient data-parallel accelerators'. PhD thesis. Massachusetts Institute of Technology, 2010 (cit. on p. 114).

[15] M. N. Bojnordi and E. Ipek. 'Memristive Boltzmann machine: a hardware accelerator for combinatorial optimization and deep learning'. In: *IEEE International Symposium on High Performance Computer Architecture*. IEEE. 2016, pp. 1–13. DOI: 10.1109/HPCA.2016.7446049 (cit. on pp. 112, 118).

[16] Y.-H. Chen, T. Krishna, J. S. Emer and V. Sze. 'Eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks'. In: *IEEE Journal of Solid-State Circuits* 52.1 (Jan. 2017), pp. 127–138. DOI: 10.1109/JSSC.2016.2616357 (cit. on pp. 3, 4, 56, 76, 80, 82, 87, 91, 116).

[17] C.-C. Cheng, C.-T. Li, C.-K. Liang, Y.-C. Lai and L.-G. Chen. 'Architecture design of stereo matching using belief propagation'. In: *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*. IEEE, May 2010, pp. 4109–4112. DOI: 10.1109/ISCAS.2010.5537613 (cit. on pp. 3, 4, 82, 116).

[18] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang and Y. Xie. 'PRIME: a novel processing-in-memory architecture for neural network computation in ReRAM-based main memory'. In: *Proceedings of the 43rd Annual International Symposium on Computer Architecture*. IEEE, June 2016, pp. 27–39. DOI: 10.1109/ISCA.2016.13 (cit. on pp. 112, 118, 119).

[19] J. Choi and R. Rutenbar. 'Hardware implementation of MRF MAP inference on an FPGA platform'. In: *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications*. IEEE, Aug. 2012, pp. 209–216. DOI: 10.1109/FPL.2012.6339183 (cit. on pp. 3, 10, 29, 31, 32, 37, 56, 121).

[20] J. Choi and R. Rutenbar. 'Video-rate stereo matching using Markov random field TRW-S inference on a hybrid CPU+FPGA computing platform'. In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM Press, Feb. 2013, pp. 63–71. DOI: 10.1145/2435264.2435278 (cit. on pp. 3, 10, 29, 31, 32, 37, 57, 121).

[21] J. Choquette. 'Nvidia's Volta GPU. Programmability and performance for GPU computing'. HotChips 29. 21st Aug. 2017 (cit. on p. 82).

[22] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengil, M. Liu, D. Lo, S. Alkalay, M. Haselman, C. Boehn, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, T. Juhasz, R. K. Kavvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, S. Reinhardt, A. Sapek, R. Seera, B. Sridharan, L. Woods, P. Yi-Xiao, R. Zhao and D. Burger. 'Accelerating persistent neural networks at datacenter scale'. HotChips 29. 22nd Aug. 2017 (cit. on p. 118).

[23] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous and A. R. LeBlanc. 'Design of ion-implanted MOSFET's with very small physical dimensions'. In: *IEEE Journal of Solid-State Circuits* 9.5 (Oct. 1974), pp. 256–268. DOI: 10.1109/JSSC.1974.1050511 (cit. on pp. 2, 4).

[24] G. Elidan, I. McGraw and D. Koller. 'Residual Belief Propagation: Informed Scheduling for Asynchronous Message Passing'. In: *Proceedings of the 22nd Conference on Uncertainty in AI*. 2006 (cit. on p. 36).

[25] D. G. Elliott, W. M. Snelgrove and M. Stumm. 'Computational RAM: a memory-SIMD hybrid and its application to DSP'. In: *1992 Proceedings of the IEEE Custom Integrated Circuits Conference*. May 1992, pp. 30.6.1–30.6.4. DOI: 10.1109/CICC.1992.591879 (cit. on p. 105).

[26] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam and D. Burger. 'Dark silicon and the end of multicore scaling'. In: *IEEE Micro* 32.3 (May 2012), pp. 122–134. DOI: 10.1109/MM.2012.17 (cit. on p. 6).

[27] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina and A. Seznec. 'Tarantula: a vector extension to the alpha architecture'. In: *Proceedings of the 29th Annual International Symposium on Computer Architecture*. IEEE Computer Society, 2002, pp. 281–292. DOI: 10.1109/ISCA.2002.1003586 (cit. on p. 114).

[28] R. Espasa and M. Valero. 'Decoupled vector architectures'. In: *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture*. IEEE Computer Society Press, 1996, pp. 281–290. DOI: 10.1109/HPCA.1996.501193 (cit. on p. 114).

[29] R. Espasa and M. Valero. 'Multithreaded vector architectures'. In: *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*. IEEE Computer Society Press, 1997, pp. 237–248. DOI: 10.1109/HPCA.1997.569677 (cit. on p. 114).

[30] R. Espasa, M. Valero and J. Smith. 'Out-of-order vector architectures'. In: *Proceedings of the 30th Annual International Symposium on Microarchitecture*. IEEE Computer Society, 1997, pp. 160–170. DOI: 10.1109/MICRO.1997.645807 (cit. on p. 114).

[31] A. Farmahini-Farahani, J. H. H. Ahn, K. Morrow and N. S. Kim. 'NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules'. In: *Proceedings of the 21st International Symposium on High Performance Computer Architecture*. Feb. 2015, pp. 283–295. DOI: 10.1109/HPCA.2015.7056040 (cit. on p. 109).

[32] R. A. Fatoohi. 'Vector performance analysis of the NEC SX-2'. In: *SIGARCH Computer Architecture News* 18.3b (Sept. 1990), pp. 389–400. DOI: 10.1145/255129.255180 (cit. on p. 113).

[33] P. F. Felzenszwalb and D. P. Huttenlocher. 'Efficient belief propagation for early vision'. In: *International Journal of Computer Vision* 70.1 (May 2006), pp. 41–54. DOI: 10.1007/s11263-006-7899-4 (cit. on pp. 2, 17, 23–26, 84, 116).

[34] M. Fishelson and D. Geiger. 'Exact genetic linkage computations for general pedigrees'. In: *Bioinformatics* 18 Suppl 1 (2002), S189–98. PMID: 12169547 (cit. on p. 2).

[35] K. Fukuda, H. Yamashita, G. Ono, R. Nemoto, E. Suzuki, T. Takemoto, F. Yuki and T. Saito. 'A 12.3mW 12.5Gb/s complete transceiver in 65nm CMOS'. In: *Proceedings of the 2010 IEEE International Solid-State Circuits Conference*. IEEE, Feb. 2010, pp. 368–369. DOI: 10.1109/ISSCC.2010.5433824 (cit. on p. 58).

[36] M. Gao, C. Delimitrou, D. Niu, K. T. Malladi, H. Zheng, B. Brennan and C. Kozyrakis. 'DRAF: a low-power DRAM-based reconfigurable acceleration fabric'. In: *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture*. June 2016, pp. 506–518. DOI: 10.1109/ISCA.2016.51 (cit. on p. 112).

[37] M. Gao, G. Ayers and C. Kozyrakis. 'Practical near-data processing for in-memory analytics frameworks'. In: *Proceedings of the 2015 International Conference on Parallel Architectures and Compilation Techniques*. Mar. 2016. DOI: 10.1109/PACT.2015.22 (cit. on p. 111).

[38] M. Gao and C. Kozyrakis. 'HRL: efficient and flexible reconfigurable logic for near-data processing'. In: *Proceedings of the 2016 IEEE International Symposium on High-Performance Computer Architecture*. Apr. 2016. DOI: 10.1109/HPCA.2016.7446059 (cit. on p. 111).

[39] M. Gao, J. Pu, X. Yang, M. Horowitz and C. Kozyrakis. 'TETRIS: scalable and efficient neural network acceleration with 3D memory'. In: *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, 2017, pp. 751–764. DOI: 10.1145/3037697.3037702 (cit. on pp. 3, 58, 80, 116).

[40] F. A. Gers, J. Schmidhuber and F. Cummins. 'Learning to forget: continual prediction with LSTM'. In: *Neural Computation* 12.10 (Oct. 2000), pp. 2451–2471. DOI: 10.1162/089976600300015015 (cit. on p. 22).

[41] M. Gokhale, B. Holmes and K. Iobst. 'Processing in memory: the Terasys massively parallel PIM array'. In: *Computer* 28.4 (Apr. 1995), pp. 23–31. DOI: 10.1109/2.375174 (cit. on pp. 7, 106, 110).

[42] S. Grauer-Gray, C. Kambhamettu and K. Palaniappan. 'GPU implementation of belief propagation using CUDA for cloud tracking and reconstruction'. In: *2008 IAPR Workshop on Pattern Recognition in Remote Sensing*. Dec. 2008, pp. 1–4. DOI: 10.1109/PRRS.2008.4783169 (cit. on p. 78).

[43] P. Gu, B. Li, T. Tang, S. Yu, Y. Cao, Y. Wang and H. Yang. 'Technological exploration of RRAM crossbar array for matrix-vector multiplication'. In: *The 20th Asia and South Pacific Design Automation Conference*. Jan. 2015, pp. 106–111. DOI: 10.1109/ASPDAC.2015.7058989 (cit. on p. 112).

[44] Q. Guo, N. Alachiotis, B. Akin, F. Sadi, G. Xu, T. M. Low, L. Pileggi, J. C. Hoe and F. Franchetti. '3D-stacked memory-side acceleration: accelerator and system design'. In: *2nd Workshop on Near-Data Processing*. 2014 (cit. on p. 108).

[45] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin and J. Park. 'Mapping irregular applications to DIVA, a PIM-based data-intensive architecture'. In: *SC Conference*. Nov. 1999, pp. 57–57. DOI: 10.1109/SC.1999.10019 (cit. on p. 107).

[46] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang et al. 'ESE: efficient speech recognition engine with sparse LSTM on FPGA'. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*. 2017, pp. 75–84. DOI: 10.1145/3020078.3021745 (cit. on pp. 72, 82, 83, 92, 102, 118).

[47] K. He, X. Zhang, S. Ren and J. Sun. 'Deep Residual Learning for Image Recognition'. In: *ArXiv e-prints* (Dec. 2015). arXiv: 1512.03385 [cs.CV] (cit. on pp. 18, 20).

[48] D. E. Heckerman, E. J. Horvitz and B. N. Nathwani. 'Toward normative expert systems: Part I. The Pathfinder project'. In: *Methods of information in medicine* 31.2 (June 1992), pp. 90–105. PMID: 1635470 (cit. on p. 2).

[49] M. D. Hill and M. R. Marty. 'Amdahl's law in the multicore era'. In: *Computer* 41.7 (July 2008), pp. 33–38. DOI: 10.1109/MC.2008.209 (cit. on p. 5).

[50] M. D. Hill and M. R. Marty. 'Retrospective on Amdahl's law in the multicore era'. In: *Computer* 50.6 (June 2017), pp. 12–14. DOI: 10.1109/MC.2017.164 (cit. on p. 6).

[51] M. Hu, J. P. Strachan, Z. Li, E. M. Grafals, N. Davila, C. Graves, S. Lam, N. Ge, J. J. Yang and R. S. Williams. 'Dot-product engine for neuromorphic computing: programming 1T1M crossbar to accelerate matrix-vector multiplication'. In: *Proceedings of the 53rd Annual Design Automation Conference*. June 2016, pp. 1–6. DOI: 10.1145/2897937.2898010 (cit. on p. 112).

[52] S. Hurkat, J. Choi, E. Nurvitadhi, J. F. Martínez and R. A. Rutenbar. 'Fast hierarchical implementation of sequential tree-reweighted belief propagation for probabilistic inference'. In: *25th International Conference on Field Programmable Logic and Applications*. IEEE, 2015. DOI: 10.1109/FPL.2015.7293934 (cit. on pp. 3, 4, 10, 23).

[53] Hybrid Memory Cube Consortium. *Hybrid Memory Cube Specification 2.1*. Tech. rep. (cit. on pp. 8, 38, 48, 57, 59, 108).

[54] A. C. Jacob, Z. Sura, T. Chen, C. Bertolli, S. Antao, O. Sallenave, K. O'Brien, H. Jacobson, R. Nair, J. R. Brunheroto, P. Jacob, B. S. Rosenburg, Y. Park, A. E. Eichenberger and C. Kim. *Compiling for the Active Memory Cube*. Tech. rep. RC25644 (WAT1612-008). IBM Research Division, Dec. 2016 (cit. on p. 103).

[55] N. Jayasena, D. P. Zhang, A. Farmahini-Farahani and M. Ignatowski. 'Realizing the full potential of heterogeneity through processing in memory'. In: *3rd Workshop on Near-Data Processing*. 2015 (cit. on p. 110).

[56] J. Jeddeloh and B. Keeth. 'Hybrid memory cube: new DRAM architecture increases density and performance'. In: *Symposium on VLSI technology*. IEEE, June 2012, pp. 87–88. DOI: 10.1109/VLSIT.2012.6242474 (cit. on pp. 58, 59, 103).

[57] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama and T. Darrell. 'Caffe: convolutional architecture for fast feature embedding'. In: *ArXiv e-prints* (2014). arXiv: 1408.5093 [cs.CV] (cit. on p. 118).

[58] J. Johnson. *CNN benchmarks*. URL: https://github.com/jcjohnson/cnn-benchmarks (visited on 12/11/2017) (cit. on pp. 76, 80, 82, 87).

[59] N. P. Jouppi, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, C. Young, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, N. Patil, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Patterson, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, G. Agrawal, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, R. Bajwa, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, S. Bates, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, D. H. Yoon, S. Bhatia and N. Boden. 'In-datacenter performance analysis of a tensor processing unit'. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM Press, 2017, pp. 1–12. DOI: 10.1145/3079856.3080246 (cit. on pp. 3, 4, 79, 116).

[60] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer and D. Shippy. 'Introduction to the Cell multiprocessor'. In: *IBM journal of Research and Development* 49.4.5 (2005), pp. 589–604. DOI: 10.1147/rd.494.0589 (cit. on p. 115).

[61] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik and J. Torrellas. 'FlexRAM: toward an advanced intelligent memory system'. In: *Proceedings of the 2019 International Conference on Computer Design: VLSI in Computers and Processors*. Oct. 1999, pp. 192–201. DOI: 10.1109/ICCD.1999.808425 (cit. on pp. 7, 106, 107).

[62] D. Kanter. 'Xeon Phi 7200 boots up for HPC'. In: *Microprocessor report* (11th July 2016) (cit. on p. 87).

[63] W. H. Kautz. 'Cellular logic-in-memory arrays'. In: *IEEE Transactions on Computers* C-18.8 (Aug. 1969), pp. 719–727. DOI: 10.1109/T-C.1969.222754 (cit. on p. 104).

[64] S. Kawabe, H. Murayama and T. Odaka. 'HITACHI supercomputer S-820 system overview'. In: *Japanese Supercomputing*. Ed. by R. H. Mendez and S. A. Orszag. Springer New York, 1988, pp. 128–135. DOI: 10.1007/978-1-4613-9600-0_10 (cit. on p. 113).

[65] D. Kim, J. Kung, S. Chai, S. Yalamanchili and S. Mukhopadhyay. 'Neurocube: a programmable digital neuromorphic architecture with high-density 3D memory'. In: *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture*. IEEE, June 2016, pp. 380–392. DOI: 10.1109/ISCA.2016.41 (cit. on pp. 80, 116).

[66] G. Kim, J. Kim, Jung Ho Ahn and Jaeha Kim. 'Memory-centric system interconnect design with Hybrid Memory Cubes'. In: *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. IEEE, Sept. 2013, pp. 145–155. DOI: 10.1109/PACT.2013.6618812 (cit. on p. 77).

[67] P. M. Kogge, T. Sunaga, H. Miyataka, K. Kitamura and E. Retter. 'Combined DRAM and logic chip for massively parallel systems'. In: *Proceedings Sixteenth Conference on Advanced Research in VLSI*. Mar. 1995, pp. 4–16. DOI: 10.1109/ARVLSI.1995.515607 (cit. on pp. 7, 105).

[68] P. M. Kogge, J. B. Brockman, T. Sterling and G. Gao. 'Processing in memory: chips to petaflops'. In: *Workshop on Mixing Logic and DRAM: Chips that Compute and Remember*. 1997 (cit. on p. 105).

[69] D. Koller and N. Friedman. *Probabilistic graphical models: principles and techniques*. The MIT Press, 2009. ISBN: 978-0-262-01319-2 (cit. on pp. 12, 14).

[70] V. Kolmogorov. 'Convergent tree-reweighted message passing for energy minimization.' In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28.10 (Oct. 2006), pp. 1568–1583. DOI: 10.1109/TPAMI.2006.200 (cit. on pp. 17, 23, 25).

[71] C. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft and K. Yelick. 'Scalable processors in the billion-transistor era: IRAM'. In: *Computer* 30.9 (1997), pp. 75–78. DOI: 10.1109/2.612252 (cit. on p. 114).

[72] C. Kozyrakis and D. Patterson. 'Overcoming the limitations of conventional vector processors'. In: *Proceedings of the 30th Annual International Symposium on Computer Architecture*. ACM, 2003, pp. 399–409. DOI: 10.1145/859618.859664 (cit. on p. 114).

[73] B. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper and K. Asanovic. 'The vector-thread architecture'. In: *IEEE Micro* 24.6 (Nov. 2004), pp. 84–90. DOI: 10.1109/MM.2004.90 (cit. on p. 114).

[74] 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. DOI: 10.1145/3065386 (cit. on pp. 20, 51).

[75] Y. Lee, C. Schmidt, A. Ou, A. Waterman and K. Asanović. *The Hwacha vector-fetch architecture manual, version 3.8.1*. Tech. rep. UCB/EECS-2015-262. University of California at Berkeley, Dec. 2015 (cit. on p. 51).

[76] K. N. Levitt and W. H. Kautz. 'Cellular arrays for the parallel implementation of binary error-correcting codes'. In: *IEEE Transactions on Information Theory* 15.5 (Sept. 1969), pp. 597–607. DOI: 10.1109/TIT.1969.1054347 (cit. on p. 104).

[77] C.-K. Liang, C.-C. Cheng, Y.-C. Lai, L.-G. Chen and H. H. Chen. 'Hardware-efficient belief propagation'. In: *IEEE Transactions on Circuits and Systems for Video Technology* 21.5 (May 2011), pp. 525–537. DOI: 10.1109/TCSVT.2011.2125570 (cit. on pp. 37, 78, 116).

[78] M. Lin, I. Lebedev and J. Wawrzynek. 'High-throughput Bayesian computing machine with reconfigurable hardware'. In: *Proceedings of the 18th annual ACM/SIGDA International Symposium on Field-programmable Gate Arrays*. ACM Press, Feb. 2010, pp. 73–82. DOI: 10.1145/1723112.1723127 (cit. on pp. 38, 117).

[79] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou and Y. Chen. 'PuDianNao: A polyvalent machine learning accelerator'. In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2015, pp. 369–381. DOI: 10.1145/2694344.2694358 (cit. on p. 117).

[80] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen and T. Chen. 'Cambricon: an instruction set architecture for neural networks'. In: *Proceedings of the 43rd Annual International Symposium on Computer Architecture*. IEEE, June 2016, pp. 393–405. DOI: 10.1109/ISCA.2016.42 (cit. on pp. 3, 80, 116).

[81] G. H. Loh, N. Jayasena, M. H. Oskin, M. Nutter, D. Roberts, M. Meswani, D. Ping and Z. Mike. 'A processing-in-memory taxonomy and a case for studying fixed-function PIM'. In: *1st Workshop on Near-Data Processing*. 2013, pp. 1–5 (cit. on p. 108).

[82] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim and H. Esmaeilzadeh. 'TABLA: a unified template-based framework for accelerating statistical machine learning'. In: *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture*. IEEE, Mar. 2016, pp. 14–26. DOI: 10.1109/HPCA.2016.7446050 (cit. on p. 118).

[83] G. E. Moore. 'Cramming more components onto integrated circuits'. In: *Electronics* 38.8 (19th Apr. 1965) (cit. on pp. 2, 4).

[84] A. Morad, L. Yavits, S. Kvatinsky and R. Ginosar. 'Resistive GP-SIMD Processing-In-Memory'. In: *ACM Transactions on Architecture Code Optimization* 12.4 (Jan. 2016), 57:1–57:22. DOI: 10.1145/2845084 (cit. on p. 111).

[85] N. Muralimanohar, R. Balasubramonian and N. P. Jouppi. *CACTI 6.0: a tool to model large caches*. Tech. rep. HPL-2009-85. HP Laboratories, Apr. 2009 (cit. on pp. 52, 81).

[86] R. C. Murphy, P. M. Kogge and A. Rodrigues. 'The characterization of data intensive memory workloads on distributed PIM systems'. In: *The Second International Workshop on Intelligent Memory Systems*. Springer-Verlag, 2001, pp. 85–103. DOI: 10.1007/3-540-44570-6_6 (cit. on p. 106).

[87] L. Nai and H. Kim. 'Instruction offloading with HMC 2.0 standard: a case study for graph traversals'. In: *Proceedings of the 2015 International Symposium on Memory Systems*. ACM Press, 2015, pp. 258–261. DOI: 10.1145/2818950.2818982 (cit. on p. 109).

[88] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C.-Y. Y. Cher, C. H. A. Costa, J. Doi, C. Evangelinos, B. M. Fleischer, T. W. Fox, D. S. Gallo, L. Grinberg, J. A. Gunnels, A. C. Jacob, P. Jacob, H. M. Jacobson, T. Karkhanis, C. Kim, J. H. Moreno, J. K. O'Brien, M. Ohmacht, Y. Park, D. A. Prener, B. S. Rosenburg, K. D. Ryu, O. Sallenave, M. J. Serrano, P. D. M. Siegl, K. Sugavanam and Z. Sura. 'Active memory cube: a processing-in-memory architecture for exascale systems'. In: *IBM Journal of Research and Development* 59.2/3 (Mar. 2015), 17:1–17:14. DOI: 10.1147/JRD.2015.2409732 (cit. on pp. 42, 44, 51, 76, 93, 107, 110).

[89] R. Nair. 'Active memory cube'. In: *2nd Workshop on Near Data Processing*. Dec. 2014 (cit. on p. 110).

[90] L. M. Ni and A. K. Jain. 'A VLSI Systolic Architecture for Pattern Clustering'. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-7.1 (Jan. 1985), pp. 80–89. ISSN: 0162-8828. DOI: 10.1109/TPAMI.1985.4767620 (cit. on p. 40).

[91] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez and C. Guestrin. 'GraphGen: FPGA framework for vertex-centric graph computation'. In: *Proceedings of the 22nd International Symposium on Field-Programmable Custom Computing Machines*. 2014, pp. 25–28. DOI: 10.1109/FCCM.2014.15 (cit. on p. 102).

[92] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra et al. 'Can FPGAs beat GPUs in accelerating next-generation deep neural networks?' In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*. ACM. 2017, pp. 5–14. DOI: 10.1145/3020078.3021740 (cit. on p. 118).

[93] *Nvidia deep learning platform. Technical overview*. 2017 (cit. on pp. 76, 80, 82, 91).

[94] *NVIDIA Tesla V100 GPU architecture. The world's most advanced data center GPU*. Aug. 2017 (cit. on pp. 8, 86, 91).

[95] S. F. Oberman and M. Y. Siu. 'A high-performance area-efficient multifunction interpolator'. In: *17th IEEE Symposium on Computer Arithmetic*. July 2005. DOI: 10.1109/ARITH.2005.7 (cit. on pp. 99, 102).

[96] M. Oskin, F. T. Chong and T. Sherwood. 'Active pages: a computation model for intelligent memory'. In: *Proceedings of the 25th Annual International Symposium on Computer Architecture*. IEEE, 1998, pp. 192–203. DOI: 10.1109/ISCA.1998.694774 (cit. on p. 107).

[97] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler and W. J. Dally. 'SCNN: an accelerator for compressed-sparse convolutional neural networks'. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM Press, 2017, pp. 27–40. DOI: 10.1145/3079856.3080254 (cit. on p. 116).

[98] S. Park, C. Chen and H. Jeong. 'VLSI Architecture for MRF Based Stereo Matching'. In: *Embedded Computer Systems: Architectures, Modeling and Simulation*. Ed. by S. Vassiliadis, M. Bereković and T. D. Hämäläinen. Vol. 4599. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 55–64. DOI: 10.1007/978-3-540-73625-7_8 (cit. on p. 117).

[99] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas and K. Yelick. 'A case for intelligent RAM'. In: *IEEE Micro* 17.2 (Mar. 1997), pp. 34–44. DOI: 10.1109/40.592312 (cit. on pp. 7, 106).

[100] J. Pearl. 'Reverend Bayes on inference engines: A distributed hierarchical approach'. In: *Proceedings of the AAAI National Conference on AI*. 1982, pp. 133–136 (cit. on p. 14).

[101] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis and F. Li. 'NDC: analyzing the impact of 3d-stacked memory+logic devices on MapReduce workloads'. In: *2014 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, Mar. 2014, pp. 190–200. DOI: 10.1109/ISPASS.2014.6844483 (cit. on p. 109).

[102] F. Quintana, J. Corbal, R. Espasa and M. Valero. 'Adding a vector unit to a superscalar processor'. In: *Proceedings of the 13th International Conference on Supercomputing*. ACM, 1999, pp. 1–10. DOI: 10.1145/305138.305148 (cit. on p. 114).

[103] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand and S. Amarasinghe. 'Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines'. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2013, pp. 519–530. DOI: 10.1145/2491956.2462176 (cit. on p. 122).

[104] P. Rosenfeld, E. Cooper-Balis and B. Jacob. 'DRAMSim2: a cycle accurate memory system simulator'. In: *IEEE Computer Architecture Letters* 10.1 (Jan. 2011), pp. 16–19. DOI: 10.1109/L-CA.2011.4 (cit. on p. 77).

[105] P. Rosenfeld. 'Performance exploration of the Hybrid Memory Cube'. PhD thesis. University of Maryland, 2014 (cit. on p. 77).

[106] D. Scharstein and R. Szeliski. 'A taxonomy and evaluation of dense two-frame stereo correspondence algorithms'. In: *International Journal of Computer Vision* 47.1-3 (Apr. 2002), pp. 7–42. DOI: 10.1023/A:1014573219977 (cit. on p. 26).

[107] D. Scharstein and R. Szeliski. 'High-accuracy stereo depth maps using structured light'. In: *Proceedings of the 2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society, 2003. DOI: 10.1109/CVPR.2003.1211354 (cit. on pp. 26, 37).

[108] P. B. Schneck. *Supercomputer architecture*. Ed. by D. DeGroot. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1987. ISBN: 978-1-4615-7959-5. DOI: 10.1007/978-1-4615-7957-1 (cit. on pp. 45, 50, 112).

[109] A. Severance and G. Lemieux. 'VENICE: a compact vector processor for FPGA applications'. In: *2012 International Conference on Field-Programmable Technology*. 2012, pp. 261–268. DOI: 10.1109/FPT.2012.6412146 (cit. on p. 117).

[110] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams and V. Srikumar. 'ISAAC: a convolutional neural network accelerator with in-situ analog arithmetic in crossbars'. In: *43rd Annual International Symposium on Computer Architecture*. IEEE, June 2016, pp. 14–26. DOI: 10.1109/ISCA.2016.12 (cit. on pp. 112, 118, 119).

[111] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra and H. Esmaeilzadeh. 'DnnWeaver: from high-level deep neural models to FPGAs'. In: *49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, Oct. 2016, pp. 1–12. DOI: 10.1109/MICRO.2016.7783720 (cit. on p. 118).

[112] K. Simonyan and A. Zisserman. 'Very deep convolutional networks for large-scale image recognition'. In: *ArXiv e-prints* (Sept. 2014). arXiv: 1409.1556 [cs.CV] (cit. on pp. 2, 20, 21, 61, 68, 83, 87, 100).

[113] S. A. Solla and Y. Lecun. 'Constrained neural networks for pattern recognition'. In: *Neural networks. Concepts, applications and implementations*. Ed. by P. Antognetti and V. Milutinović. Vol. 4. Prentice Hall, 1991, pp. 142–161 (cit. on p. 19).

[114] L. Song, X. Qian, H. Li and Y. Chen. 'PipeLayer: a pipelined ReRAM-based accelerator for deep learning'. In: *2017 IEEE International Symposium on High-Performance Computer Architecture*. 2017, pp. 541–552. DOI: 10.1109/HPCA.2017.55 (cit. on pp. 118, 119).

[115] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov. 'Dropout: a simple way to prevent neural networks from overfitting'. In: *Journal of Machine Learning Research* 15.1 (Jan. 2014), pp. 1929–1958 (cit. on p. 18).

[116] H. S. Stone. 'A logic-in-memory computer'. In: *IEEE Transactions on Computers* C-19.1 (Jan. 1970), pp. 73–78. DOI: 10.1109/TC.1970.5008902 (cit. on p. 104).

[117] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke and A. Rabinovich. 'Going deeper with convolutions'. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition*. 2015. DOI: 10.1109/CVPR.2015.7298594 (cit. on p. 20).

[118] R. Szeliski, R. Zabih, D. Scharstein, O. Veksler, V. Kolmogorov, A. Agarwala, M. Tappen and C. Rother. 'A comparative study of energy minimization methods for Markov random fields with smoothness-based priors'. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 30.6 (June 2008), pp. 1068–1080. DOI: 10.1109/TPAMI.2007.70844 (cit. on pp. 18, 23, 24, 61).

[119] M. F. Tappen and W. T. Freeman. 'Comparison of graph cuts with belief propagation for stereo, using identical MRF parameters'. In: *Proceedings of the Ninth International Conference on Computer Vision*. Vol. 2. IEEE, 2003, pp. 900–906. DOI: 10.1109/ICCV.2003.1238444 (cit. on pp. 17, 61).

[120] J. Torrellas. 'FlexRAM: toward an advanced intelligent memory system: a retrospective paper'. In: *The 2012 IEEE 30th International Conference on Computer Design*. IEEE Computer Society, 2012, pp. 3–4. DOI: 10.1109/ICCD.2012.6378607 (cit. on p. 7).

[121] Y.-C. Tseng and T.-S. Chang. 'Architecture design of belief propagation for real-time disparity estimation'. In: *IEEE Transactions on Circuits and Systems for Video Technology* 20.11 (Nov. 2010), pp. 1555–1564. DOI: 10.1109/TCSVT.2010.2087434 (cit. on p. 116).

[122] K. Uchida. 'Fujitsu VP2000 series supercomputer'. In: *Supercomputers and Their Performance in Computational Fluid Dynamics*. Ed. by K. Fujii. Vieweg+Teubner Verlag, 1993, pp. 17–41. DOI: 10.1007/978-3-322-87863-2_2 (cit. on pp. 51, 113).

[123] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey et al. 'ScaleDeep: a scalable compute architecture for learning and evaluating deep networks'. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM. 2017, pp. 13–26. DOI: 10.1145/3079856.3080244 (cit. on p. 117).

[124] O. Vinyals, A. Toshev, S. Bengio and D. Erhan. 'Show and tell: a neural image caption generator'. In: *ArXiv e-prints* (Nov. 2014). arXiv: 1411.4555 [cs.CV] (cit. on p. 2).

[125] M. Wainwright, T. Jaakkola and A. Willsky. 'MAP estimation via agreement on trees: message-passing and linear programming'. In: *IEEE Transactions on Information Theory* 51.11 (Nov. 2005), pp. 3697–3717. DOI: 10.1109/TIT.2005.856938 (cit. on p. 23).

[126] S. Wang, Y. Song, M. N. Bojnordi and E. Ipek. 'Enabling energy efficient hybrid memory cube systems with erasure codes'. In: *2015 IEEE/ACM International Symposium on Low Power Electronics and Design*. IEEE, July 2015, pp. 67–72. DOI: 10.1109/ISLPED.2015.7273492 (cit. on p. 58).

[127] S. Wang, X. Zhang, Y. Li, R. Bashizade, S. Yang, C. Dwyer and A. R. Lebeck. 'Accelerating Markov random field inference using molecular optical Gibbs sampling units'. In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture*. IEEE, June 2016, pp. 558–569. DOI: 10.1109/ISCA.2016.55 (cit. on pp. 82, 118).

[128] J. Wawrzynek, K. Asanovic, B. Kingsbury, D. Johnson, J. Beck and N. Morgan. 'Spert-II: a vector microprocessor system'. In: *Computer* 29.3 (Mar. 1996), pp. 79–86. DOI: 10.1109/2.485896 (cit. on pp. 114, 117).

[129] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang and J. Cong. 'Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs'. In: *Proceedings of the 54th Annual Design Automation Conference*. ACM Press, 2017, pp. 1–6. DOI: 10.1145/3061639.3062207 (cit. on pp. 3, 4, 118).

[130] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Ł. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes and J. Dean. 'Google's neural machine translation system: bridging the gap between human and machine translation'. In: *ArXiv e-prints* (Sept. 2016), pp. 1–23. arXiv: 1609.08144 [cs.CL] (cit. on p. 2).

[131] M. Wuebbling. *The new NVIDIA TITAN X: The ultimate. Period.* 21st July 2016. URL: https://blogs.nvidia.com/blog/2016/07/21/titan-x/ (visited on 26/09/2017) (cit. on p. 79).

[132] W. A. Wulf and S. A. McKee. 'Hitting the memory wall: implications of the obvious'. In: *SIGARCH Computer Architecture News* 23.1 (Mar. 1995), pp. 20–24. DOI: 10.1145/216585.216588 (cit. on p. 7).

[133] S. Xi, O. Babarinsa, M. Athanassoulis and S. Idreos. 'Beyond the wall: near-data processing for databases'. In: *Proceedings of the 11th International Workshop on Data Management on New Hardware*. ACM Press, 2015. DOI: 10.1145/2771937.2771945 (cit. on p. 110).

[134] X. Xiang, M. Zhang, G. Li, Y. He and Z. Pan. 'Real-time stereo matching based on fast belief propagation'. In: *Machine Vision and Applications* 23.6 (Jan. 2012), pp. 1219–1227. DOI: 10.1007/s00138-011-0405-1 (cit. on p. 78).

[135] L. Xu, D. P. Zhang and N. Jayasena. 'Scaling deep learning on multiple in-memory processors'. In: *3rd Workshop on Near-Data Processing*. 2015 (cit. on p. 110).

[136] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu and M. Ignatowski. 'TOP-PIM: throughput-oriented programmable processing in memory'. In: *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*. ACM, 2014, pp. 85–98. DOI: 10.1145/2600212.2600213 (cit. on p. 110).

[137] 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: *IEEE Transactions on Computers* 50.11 (Nov. 2001), pp. 1117–1132. DOI: 10.1109/12.966490 (cit. on pp. 108, 110).

[138] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. B. Srivastava, R. Gupta and Z. Zhang. 'Accelerating binarized convolutional neural networks with software-programmable FPGAs.' In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*. 2017, pp. 15–24. DOI: 10.1145/3020078.3021741 (cit. on p. 118).

[139] W. Zhao, H. Fu, G. Yang and W. Luk. 'Patra: parallel tree-reweighted message passing architecture'. In: *2014 24th International Conference on Field Programmable Logic and Applications*. IEEE, 2014, pp. 1–6. DOI: 10.1109/FPL.2014.6927506 (cit. on p. 36).

[140] Q. Zhu, K. Vaidyanathan, O. Shacham, M. Horowitz, L. Pileggi and F. Franchetti. 'Design automation framework for application-specific logic-in-memory blocks'. In: *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*. July 2012, pp. 125–132. DOI: 10.1109/ASAP.2012.21 (cit. on p. 108).

[141] Q. Zhu, B. Akin, H. E. Sumbul, F. Sadi, J. C. Hoe, L. Pileggi and F. Franchetti. 'A 3D-stacked logic-in-memory accelerator for application-specific data intensive computing'. In: *2013 IEEE International 3D Systems Integration Conference*. IEEE, Oct. 2013, pp. 1–7. DOI: 10.1109/3DIC.2013.6702348 (cit. on p. 108).