TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Deep convolutional network evaluation on the Intel Xeon Phi

where subword parallelism meets many-core

Raina, G.

Award date:
2016

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Technische Universiteit Eindhoven University of Technology

RECORE

Eindhoven University of Technology

Master's Thesis

Deep Convolutional Network evaluation on the Intel Xeon Phi: Where Subword Parallelism meets Many-Core

Author:
Gaurav RAINA
MSc Student
Embedded Systems
0871676
g.raina@student.tue.nl

Committee:
(TU/e)
Prof.Dr. Henk Corporaal
Dr.ir. Pieter Cuijpers
ir. Maurice Peemen
(Recore Systems)
Dr.ir. Gerard Rauwerda

January 25, 2016

Abstract

With a sharp decline in camera cost and size along with superior computing power available at increasingly low prices, computer vision applications are becoming ever present in our daily lives. Research shows that Convolutional Neural Networks (ConvNet) can outperform all other methods for computer vision tasks (such as object detection) in terms of accuracy and versatility [31].

One of the problems with these Neural Networks, which mimic the brain, is that they can be very demanding on the processor, requiring millions of computational nodes to function. Hence, it is challenging for Neural Network algorithms to achieve real-time performance on general purpose embedded platforms.

Parallelization and vectorization are very effective ways to ease this problem and make it possible to implement such ConvNets on energy efficient embedded platforms. This thesis presents the evaluation of a novel ConvNet for road speed sign detection [38], on a breakthrough 57-core Intel Xeon Phi processor with 512-bit vector support. This mapping demonstrates that the parallelism inherent in the ConvNet algorithm can be effectively exploited by the 512-bit vector ISA and by utilizing the many core paradigm.

Detailed evaluation shows that the best mappings require data-reuse strategies that exploit reuse at the cache and register level. These implementations are boosted by the use of low-level vector intrinsics (which are C style functions that map directly onto many Intel assembly instructions). Ultimately we demonstrate an approach which can be used to accelerate Neural Networks on highly-parallel many core processors, with execution speedups of more than 12x on single core performance alone.

Acknowledgments

I would like to express my sincere gratitude and respect to my supervisors at the university, Prof. Henk Corporaal, Maurice Peemen and my supervisors at Recore Systems, namely Dr. Gerard Rauwerda and Dr. Yifan He.

I offer my heartfelt thanks to my colleagues and seniors at Recore Systems in Enschede namely Fasil Taddesse, Jarkko Huijts, Jordy Potman, Timon Ter Braak, Patrick Klok, Ines Nijman, Marcel van de Burgwal, Tauseef Ali, for their insight and guidance, Marco Greven for the much needed IT infrastructure, Frank Litjens, Kim Sunesen, Eduard Fernandez and Nathalie Watermulder for their support. I cherish the opportunity that I had to learn something very valuable from each one of you.

I thank my colleagues and friends at the Electronic Systems group at TU/e for their support and encouragement which kept me going. The enthusiasm and hard work displayed by the ES group members in their work, inspired me to perform my best.

I am indebted to my family in India, without whose love, support and encouragement this could not have become a reality.

Gaurav Raina
Eindhoven, December 2015.

Contents

List of Abbreviations **v**

1 Introduction **1**
1.1 Trends in Computer vision systems 3
1.2 Research Problem 5
1.3 Major contributions 5
1.4 Outline of the thesis report 6

2 Background and Related Work **7**
2.1 Artificial Neural Nets 7
2.2 Convolutional Neural Networks 8
2.2.1 Computations and Data Transfer 10
2.2.2 Parallelism and Data Reuse 11
2.3 The Roofline model: A performance measure 11
2.3.1 Arithmetic Intensity 12
2.3.2 Bandwidth Ceilings 13
2.3.3 An example Roofline 13
2.4 STREAM benchmark 14
2.5 Optimization Approach 14
2.6 Levels of parallelism 16
2.7 Related Work 17

3 ConvNet Speed Sign detection application **19**
3.1 Speed Sign detection Algorithm 19
3.1.1 *C* language implementation of the speed sign ConvNet algorithm 20
3.2 Initial Experiments and Results 23
3.2.1 Dimensioning computational requirements 24

4 Hardware Platforms **26**
4.1 Intel Core i7 - 5930K 26
4.2 Intel Xeon Phi - 31S1P 29
4.2.1 Many Integrated Core micro-architecture 30
4.3 Sub-word parallelism 32

4.4 Summary 33

5 ConvNet Mapping on the Core i7 **35**
5.1 A Brief on the Intel intrinsics format 35
5.2 Method to Vectorize a Convolutional Kernel on the Haswell . 36
5.2.1 Fused Multiply Add Intrinsic 36
5.2.2 Gathering and Arranging data for FMA 38
5.3 Results 40
5.4 Roofline model: Performance evaluation 41
5.4.1 Calculation of Compute Roofline Haswell 42
5.4.2 Calculation of Memory Roofline Haswell 42
5.5 Analysis of Results 42

6 ConvNet Mapping on the Xeon Phi **44**
6.1 Going from Core i7 to Xeon Phi 44
6.1.1 Multiply-Accumulate instruction 45
6.2 Method to Vectorize a Convolutional Kernel 46
6.3 Results 50
6.4 Roofline model: Performance evaluation 51
6.4.1 Calculation of compute ceiling Xeon Phi 51
6.4.2 Calculation of memory ceiling of Xeon Phi 51
6.5 Intel Core i7 (*Haswell*) v/s Xeon Phi (*MIC*) 53
6.6 Challenges faced on the Xeon Phi 54

7 Conclusion and Future Work **55**
7.1 Conclusions 55
7.2 Future Work 56
7.2.1 Multi-core mapping using OpenMP 56
7.2.2 Auto vectorization with OpenMP `#pragma simd` . . . 56
7.2.3 Performance comparison with SIMD accelerator and GPU 57

Bibliography **63**

A Appendix: Core i7 platform details **64**

List of Abbreviations

ANN	Artificial Neural Network
ASIC	Application-Specific Integrated Circuit
AVX	Advanced Vector Extensions
ConvNet	Convolutional Neural Networks
CPU	Central Processing Unit
DDR	Double Data-Rate
FLOP	FLoating-point OPeration
FMA	Fused Multiply Add
FPGA	Field-Programmable Gate Array
GCC	GNU C Compiler
GPU	Graphics Processing Unit
ICC	Intel C Compiler
ILP	Instruction Level Parallelism
IMCI	Initial Many Core Instructions
ISA	Instruction Set Architecture
KNC	Knights Corner (First generation Many Integrated Core architecture)
MAC	Multiply-ACcumulate operation
MIC	Many Integrated Core architecture
MLP	Multi-Layer Perceptron
RAM	Random-Access Memory

SIMD	Single Instruction Multiple Data
SMT	Simultaneous MultiThreading
SSE	Streaming SIMD Extensions
VLSI	Very-Large-Scale Integration

Chapter 1

Introduction

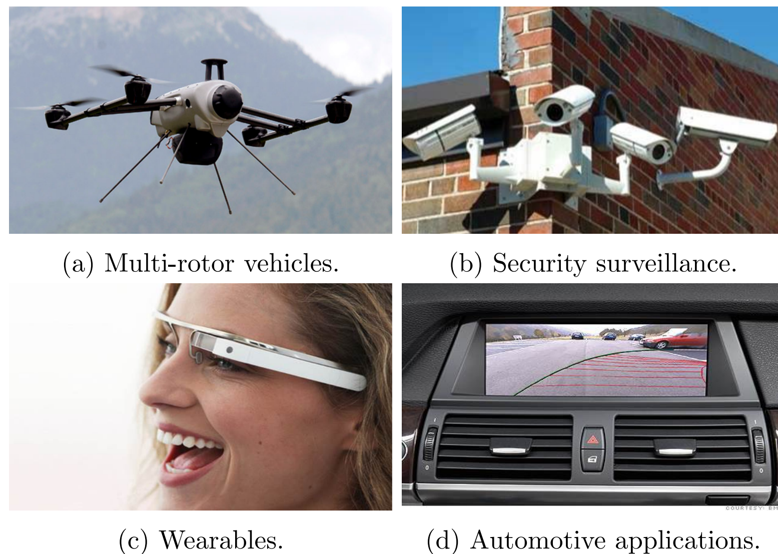
Figure 1.1: [1]

One of the major goals of the "Eyes of Things" EU international project which aims to build the best embedded computer vision platform, is to enable a future with computer vision devices embedded everywhere in the world around us [49]. To a large extent this has already become a reality in the world we live in today with cameras all around us, in smart-phones, laptops, flying drones, surveillance cameras, cameras in automobiles, wearables, industrial vision applications, in robotics, smart homes and so on.

Many of these device are connected to the Internet and fit into the larger Internet of Things trend prevalent in the tech industry. All this has led to new opportunities in the areas of computer vision and image processing, which demand advanced recognition that is as good or even better than humans. Such technologies have the potential to support us in complex, time consuming and dangerous tasks or can even take full control of such systems.

To meet the computational demands of such vision tasks, efficient algorithms and very capable processing hardware are required. There has been a lot of innovation in this area, giving rise to many new computer architectures and vision algorithms have been developed to run such vision tasks. Selecting the right hardware platforms and an optimal software implementations of vision algorithms becomes all the more important with real-time constraints and while running on embedded devices.

This thesis focuses on a very flexible class of image recognition algo-



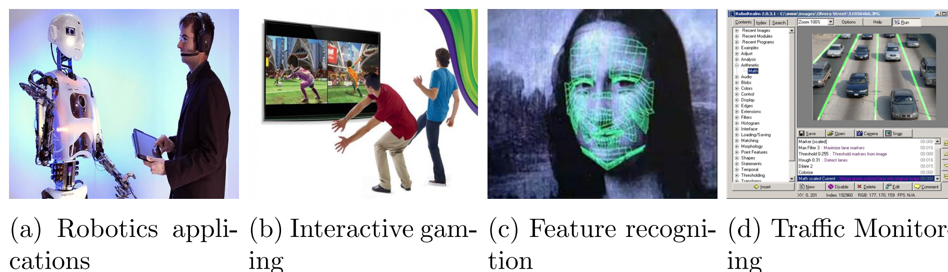
(a) Multi-rotor vehicles. (b) Security surveillance.

(c) Wearables. (d) Automotive applications.

Figure 1.2: The ubiquitous presence of cameras

rithms called Deep Neural Networks, which are biologically inspired networks trained for recognition tasks. More specifically we focus on a subtype of neural network algorithms, namely Convolutional Neural Networks (ConvNet). They are a special type of artificial neural network topology, that is inspired by the animal visual cortex and tuned for computer vision tasks.

There is a lot of research work in this field resulting in many neural network implementations and platforms, but the majority exhibit a problem of high computational complexity and unoptimized implementations. This project address the problem by proposing neural network parallelization approaches to effectively utilize the vector capabilities of modern processor hardware.



(a) Robotics applications (b) Interactive gaming (c) Feature recognition (d) Traffic Monitoring

Figure 1.3: Computer vision applications

This chapter is organized as follows: We start with a discussion on various recent trends in computer vision systems in Section 1.1. Then in

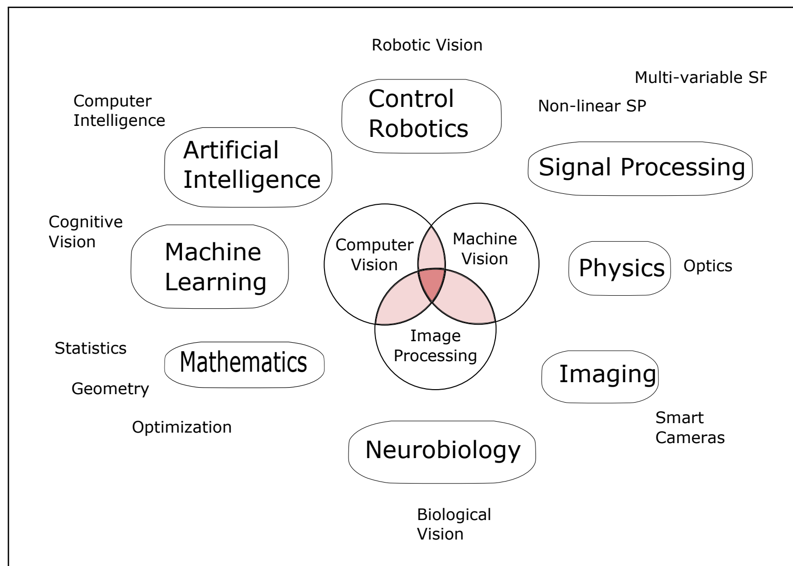


Figure 1.4: Overview of relations between computer vision and other fields [27]

Section 1.2, we introduce the existing research problems in deep neural networks. Finally, the main contributions of this thesis are then addressed in Section 1.3.

1.1 Trends in Computer vision systems

A computer vision system is a system with cameras, an image processing section with compute hardware and custom software which analyses the images to extract relevant data. An overview of the relation between computer vision and other disciplines is shown in fig.1.4.

Convolutional neural nets have revolutionized the computer vision industry in the years since 2012, as seen by the trends in the top computer vision conference Computer Vision and Pattern Recognition (CVPR), IEEE Conference. Neural net algorithms dominate all the top spots in the publications [29, 45]. Deep Neural Networks take the top spots in ImageNet (Large Scale Visual Recognition Challenge) and Microsoft Common Objects in Context (COCO, an image recognition, segmentation and captioning dataset), which are the state-of-the art object recognition challenges [31].

Classical methods are currently being replaced by machine learning approaches based upon deep learning. This is due to the many advantages of deep learning over manually designed algorithms. The quality of classical methods depends on the quality of the hand-crafted front-end while deep learning algorithms depend on the training using quality datasets that are

easy to obtain. Also, the other advantage is that it is simple to modify such deep learning systems, by just adding new trained data. For classical methods, the designer needs to develop a largely new routine for each new problem.

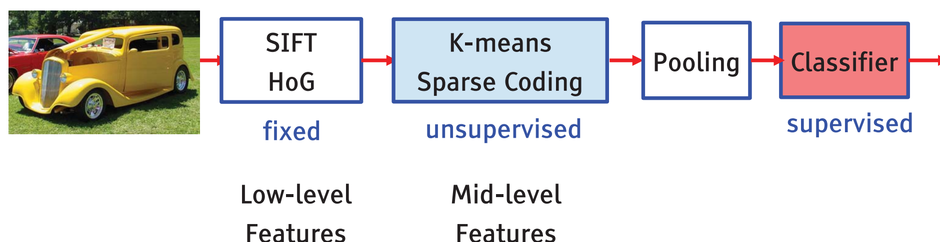


Figure 1.5: Mainstream object recognition pipeline 2006-2012 [29]

Traditional pattern recognition (refer fig.1.5) can be broadly classified in two stages: The first stage low-level feature extraction is often hand-crafted, for eg. Scale-Invariant Feature Transform (SIFT), Histogram of Oriented Gradients (HOG) are some standard ways of extracting low level features from images [35, 16]. The second stage expands the dimension so that the features can be aggregated and sent to a classifier for eg. sparse coding, vector quantization [13].

Figure 1.6: Latest deep learning pipelines are hierarchical and trained [29]

Modern deep learning pipelines can be simply represented as shown in figure 1.6. The structure is made up of a sequence of modules, all stages are trainable, with each stage converting the data into a higher representation. This can be thought of as a hierarchy of representations with an increasing level of abstraction. Some of the applications of deep learning can be seen in the figure 1.3.

Need for a trainable algorithm: Traditional techniques in image recognition use a cascade of specially designed filters which pre-process the image before detecting. Designing such a pipeline of algorithms is very time consuming and must be redone to support new road signs or other objects. The mapping of these computationally complex algorithms to onboard vision platforms is a time consuming task and they are not designed to be updated later during lifetime. This is why the vision system used in this project is based on a fully trainable Convolutional Neural Network (ConvNet).

1.2 Research Problem

As mentioned earlier, the computational requirements of deep convolutional networks are massive, requiring millions of arithmetic operations per input image. Sequential processors do not achieve a high throughput in such embarrassingly parallel pixel processing applications. New platforms that employ massive parallelism need to be evaluated so as to remove the performance bottlenecks.

Parallelization is one of the most effective ways to ease this problem and make it possible to implement such neural networks on energy efficient embedded platforms [32]. In this project, the Intel Xeon Phi co-processor is evaluated as a representative candidate for parallel many-core systems. The problem tackled in this project is the performance evaluation of a convolutional neural network (ConvNet) on the highly-parallel Intel Xeon Phi coprocessor. More specifically it is shown how one could map a ConvNet onto the Intel Haswell and the Many Integrated Core (MIC) architecture, taking into account issues with memory transfer (Roofline model, section 2.3), caches, vector parallelism and multi-core.

1.3 Major contributions

The most important contributions of this project are:

- **Evaluation of ConvNet requirements** (Chapter 2)
  A convolutional network based speed sign detection application is thoroughly studied and tested, to extract the typical requirements in terms of computations and memory accesses.

- **Vectorized mappings on the Core i7 and Xeon Phi** (Chapters 5 and 6)
  The speed sign detection algorithm is first optimized on the Intel Core i7 followed by the Intel Xeon Phi. This application is accelerated using Single Instruction Multiple Data (SIMD) vector parallelization approach along with other optimizations. Up to 12 times faster performance is achieved per network layer, while running on a single processor core.

- **Bottleneck analysis with evaluation** (Chapters 5 and 6)
  Here we try to analyze the performance numbers, with the help of the Roofline models. We try to explain the difference between ideal expected performance gains, versus the actual practical gains in performance. This helps us judge the maximum possible performance of the neural network application along with the hardware platform.

1.4 Outline of the thesis report

The remainder of this thesis is organized as follows: Chapter 2 provides an overview of artificial neural networks, the Roofline performance model and presents research works related to the contributions presented in this thesis. In Chapter 3, details about the speed sign detection application used in this project are explained. Chapter 4 briefly presents the relevant hardware architecture details of the two processor platforms used in this project. Chapters 5 and 6, discuss the details of the tested vectorized implementations of the ConvNet application on the Core i7 and Xeon Phi platforms respectively. Finally, the thesis concludes in Chapter 7 along with a discussion on future work.

Chapter 2

Background and Related Work

This chapter begins with an overview of artificial neural networks in Section 2.1. The Section 2.3 introduces the Roofline performance model. By the end of this chapter the reader should have a basic framework to understand the convolutional network used in this project.

2.1 Artificial Neural Nets

Artificial neural networks (ANNs) are biologically inspired networks interconnected in a specific manner as per the application requirement. One of the many advantages of artificial neural nets is that they require minimum or no preprocessing of input data. While the traditional elaborate feature extractors are hand tuned for particular sets of data. A simplistic mathematical model of an artificial neural net is shown in fig. 2.1.

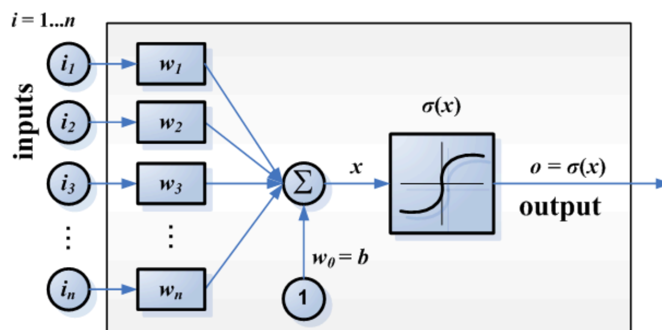


Figure 2.1: An artificial neuron model [43]

Artificial neural network models have the ability to learn and generalize by using examples. This ability to adapt to the recognition task even after design time, makes it unique compared to other training methods.

2.2 Convolutional Neural Networks

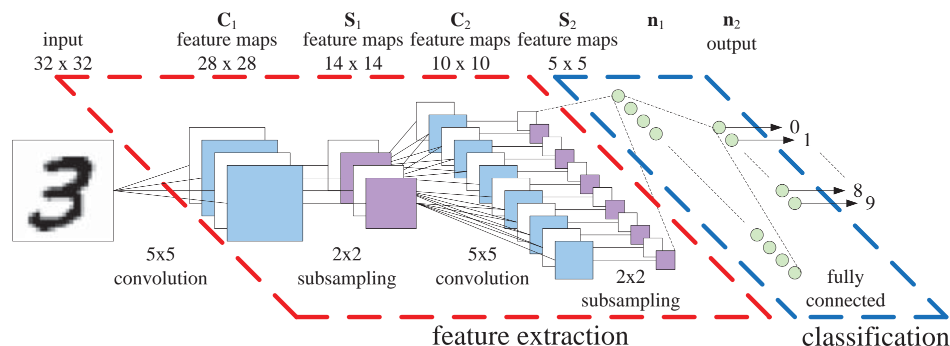


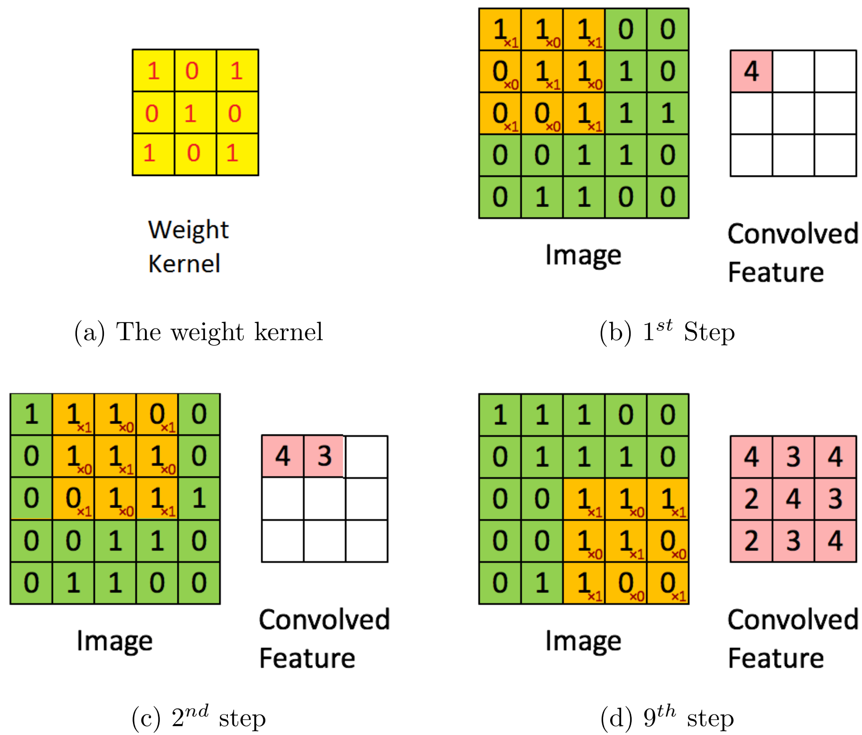
Figure 2.2: A ConvNet for a handwritten digit recognition application [39]

A convolutional neural network is a special type of artificial neural network topology, that is inspired by the animal visual cortex and tuned for computer vision tasks by Yann LeCun in early 1990s [28, 53]. It is a multi-layer perceptron (MLP), which is feed-forward artificial neural network model, specifically designed to recognize two-dimensional shapes. This type of network shows a high degree of invariance to translation, scaling, skewing, and other forms of distortion [20]. The position invariance of the features makes it possible to reuse most of the results of the feature extractor, this makes a ConvNet very computationally efficient for object detection tasks. At each new search location only a single output pixel needs to be computed [41]. The ability to share weight kernels increases data locality and leads to a more efficient implementation in the case of convolutional networks. This is another advantage of ConvNets over fully connected multi-layer perceptrons.

An example of a simple convolution operation with a weight kernel is shown in figure 2.3. Here the 3x3 weight kernel (yellow) is convolved with a 3x3 pixel section of the image (green) and the resulting output pixel is written in the feature map (pink). This operation is done for each row and column of the image, which results in a 3x3 output feature map.

In convolutional networks, any complex new task is learned in a supervised manner by means of a network whose structure includes the following forms of constraints:

Feature extraction: Each neuron takes its synaptic inputs from a local receptive field in the previous layer, thereby forcing it to extract local features. Once a feature has been extracted, its exact location becomes less important, so long as its position relative to other features is preserved.



(a) The weight kernel

(b) 1st Step

(c) 2nd step

(d) 9th step

Figure 2.3: 3x3 convolution calculation example [2]

Feature mapping: Each computational layer of the network is composed of multiple feature maps, with each feature map being in the form of a plane within which the individual neurons are constrained to share the same set of synaptic weights. This second form of structural constraint has the following beneficial effects:

- Shift invariance, is introduced into the operation of a feature map through the use of convolution with a small size kernel, followed by a sigmoid activation function.
- Reduction in the number of free parameters, is achieved through the use of weight sharing.

An input image operated up on by a convolutional kernel produces an output feature map, see fig 2.4.

Subsampling: Each convolutional layer is followed by a computational layer that performs local averaging and sub-sampling, whereby the resolution of the feature map is reduced. This operation has the effect of reducing the sensitivity of the feature maps output to shifts and other forms of distortion.

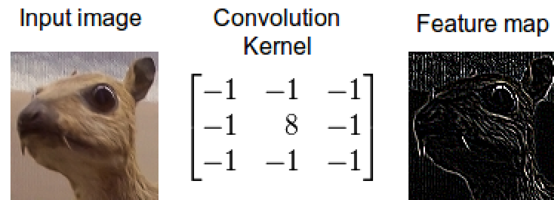


Figure 2.4: Image converted into a feature map by a convolution kernel [17]

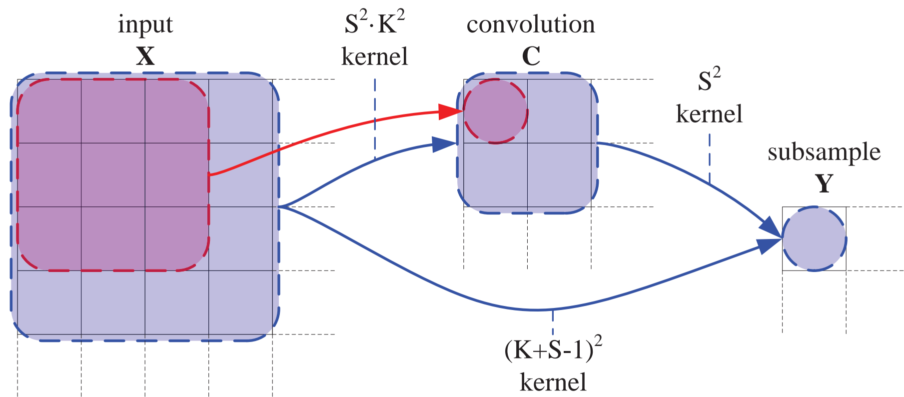


Figure 2.5: An example of a 2D feature extraction layer [39]

In figure 2.5, we see the data dependencies between a 4x4 section of the input image (X), the convolution layer (resulting in C) and the subsampling layer (resulting in Y). Here the convolution kernel is of size 3x3 (K=3), the subsampling factor is 2 (S=2) and the final result pixel is store in Y.

2.2.1 Computations and Data Transfer

The compute requirement of a ConvNet depends greatly on the complexity of a classification task at hand. The main contributor to the compute load are the convolutions. In the case of the speed sign detection application, as we increase the number of speed signs that can be identified, the number of multiply-accumulate operations which represent the convolutions, increase significantly. As the detail in the input images increases, so do the data transfer requirements. In case of high definition images, the intermediate feature maps may become too large for the on-chip main memory to store, which leads to stores to external memories like DDR RAM. This means, that each pixel might be read and written to and from external memory more than once. This can incur a huge time penalty as off-chip DDR RAM access are in the order of 100x slower than accessing on-chip cache memories. An example of the real-world computations and data access numbers

can be seen the table 3.1, which summarizes these numbers for the speed sign detection application.

2.2.2 Parallelism and Data Reuse

It is wise to exploit the spatial correlations (fig.2.6) and temporal reuse (fig.2.7) inherent in images, in developing neural net algorithms for these applications. Individual convolutions should be parallelized to improve the throughput of the ConvNet algorithm.

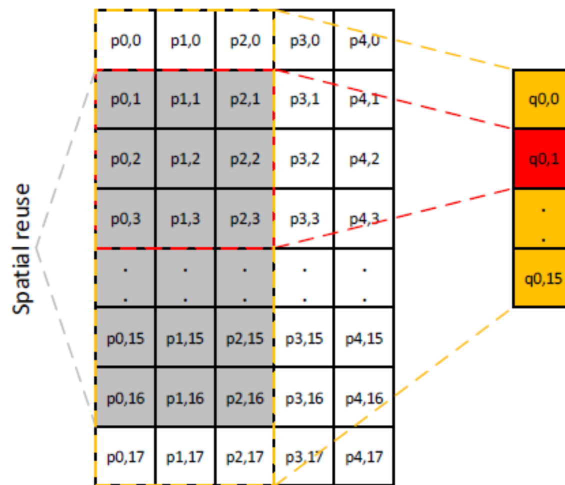


Figure 2.6: Spatial locality as seen at the pixel compute level [42]

Optimally, many neighboring convolutions can be computed together to produce an output array. As seen in fig.2.6 such parallel operations reduce the overall data transfer required by exploiting the spatial locality of the input pixels, shown as gray pixels. When the convolution window moves to the next column, see fig.2.7, the gray pixels are reused in this new set of convolutions, reducing memory accesses substantially.

2.3 The Roofline model: A performance measure

To estimate the performance of the neural network application introduced in previous sections, on a target processor platform in an insightfully visual manner we need the roofline model. Roofline is a performance model which helps us arrive at a performance bound for an algorithm running on multi-core, many-core or accelerator processor architectures.

The Roofline model gives an estimation of the application performance based on the assumption that performance is limited by two factors: memory

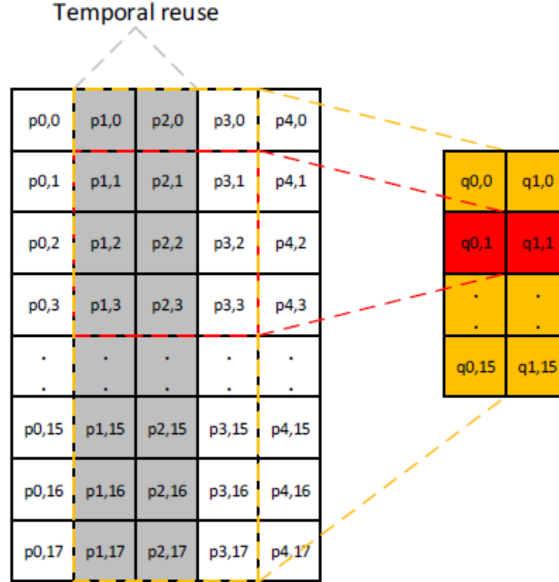


Figure 2.7: Temporal locality as seen at the pixel compute level [42]

bandwidth between off-chip memory and the processor or compute throughput of the architecture. This Roofline figure can be used to assess the quality of achieved performance (with the current implementation) and the inherent performance ceilings of the specific processor [52].

The Roofline model can be applied to different architectures, because it assumes computational elements (e.g. CPUs, cores, function units) and memory elements (e.g. RAM, caches, register) as black boxes interconnected via a network.

2.3.1 Arithmetic Intensity

This is one of the absolute core parameters in the making of the Roofline model. Arithmetic Intensity is the ratio of total arithmetic operations to total data movement in bytes, see equation 2.1.

$$\text{Arithmentic Intensity} = \frac{\text{Total number of operations}}{\text{Total data movement to memory(bytes)}} \tag{2.1}$$

Cache size, cache conflicts and cache misses can increase data movement to off-chip memory significantly, which can reduce arithmetic intensity.

$$\boxed{C_{roof} = \#cores \cdot C_f \cdot C_{ops} \quad \text{operations/sec}} \tag{2.2}$$

$$C_{ops} = \sum_{0}^{q-1} FU_{width} \quad (2.3)$$

As seen in the equation 2.2, to find the computational roofline we have to take the produce of the number of cores, the clock frequency of the cores (C_f) and the number of FLOP per core per clock cycle (C_{ops}). C_{ops} is found using the formula 2.3, which is a summation of the width of Functional Units (FU_{width}) over q, which is the number of functional units (FU) [50].

2.3.2 Bandwidth Ceilings

$$\boxed{B_{roof} = M_f \cdot M_{dt} \cdot M_{width} \cdot \#channels \quad \text{bytes/second}} \quad (2.4)$$

The equation 2.4, can be used to calculate the maximum theoretical bandwidth roofline B_{roof}; where M_f is the frequency at which the memory unit can issue transfers, M_{dt} is the number of data transfers per clock tick, M_{width} is the width of a single meory transfer in bytes and $\#channels$ is the number of channels of memory [50].

Memory bandwith can be found by taking the ratio of the y unit by x unit, i.e. attainable compute performance (Ops/second) by the arithmetic intensity (Ops/byte), which yields bandwidth (bytes/second).

2.3.3 An example Roofline

An example of the Roofline model is shown in figure 2.8, both the axes use a log scale. On the Y axis the maximum attainable compute performance is expressed in Giga Floating Point Operations per second (GFLOP/s) and on the X axis the arithmetic intensity of the kernel under test is expressed in terms of the ratio of Floating Point Operations per byte of data loaded from external memory (FLOP/Byte). The first application (kernel 1), has an arithmetic intensity of 2 Ops/byte and its performance is bounded by the memory bandwidth. The second application (kernel 2), has arithmetic intensity of 8 Ops/byte and it is bounded by the computational performance of the processor.

Performance is bounded by the optimizations which are implemented. In-core optimizations can affect horizontal ceilings, bandwidth optimizations affect the diagonal ceilings and memory traffic optimizations are represented as vertical walls.

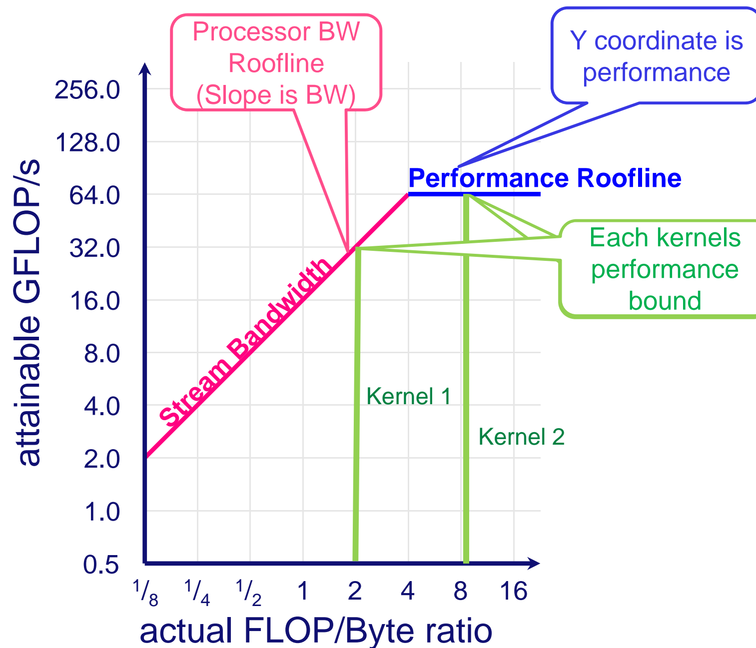


Figure 2.8: An example Roofline model

2.4 STREAM benchmark

The STREAM benchmark is an industry standard benchmark to measure sustainable memory bandwidth. It measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels [34]. The benchmark is a very useful indicator for real-world application performance. This benchmark is repeatedly referred to in many high performance computing benchmarks, for eg. the High-Performance Conjugate Gradient Benchmark (HPCG) on the Xeon Phi [37]. In this paper, the net bandwidth for HPCG is reported to be about 65 to 80 percent of the STREAM benchmark, hence this shows that it is a good indicator for the bandwidth performance. We use the results of STREAM benchmark as a cross check for our practical result from the experiments.

2.5 Optimization Approach

Optimizations on the ConvNet application are performed in the following order:

1. Compiler optimization flags
2. Loop unrolling
3. Vectorization using SIMD intrinsics

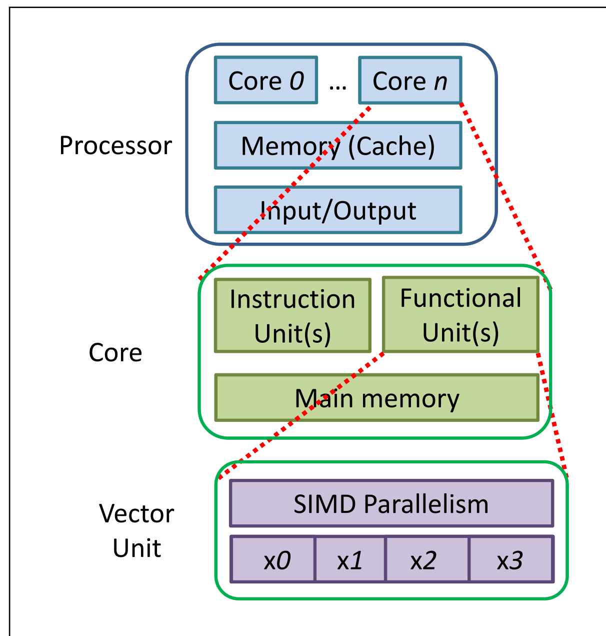


Figure 2.9: A big picture of the different levels of parallelism in computer architecture

It is common to see enormous speedups being reported from porting existing CPU applications to GPUs and other accelerators like the Intel Xeon Phi, but often very little thought is given to the baseline chosen for the comparison. The performance of the original code is often directly compared against code produced by experts on the chosen accelerator. The resulting speedup, while factually correct, may give a misleading result [33].

To make a fair comparison, we implement basic software optimization before we check for what speedups are achieved. We implement the same speed sigh detection algorithm, on all versions of the code.

We analyze the order of execution and remove unnecessary work in critical loops. Then we iterate on a number of measure-optimise-measure cycles. We compare performance with both un-vectorized and auto-vectorized versions of the application code.

We focus on execution time to find the bottlenecks, remove them and measure again. We try to make the best use of the features of the hardware and the accompanying instruction set. The CPUs include many opportunities to boost application performance including vector units, special instruction sets, caches and multi-core, if used correctly.

2.6 Levels of parallelism

In figure 2.9, we can see a birds eye view of the different levels of parallelism available in computer architecture. At the top we start with a computer running on a multi-core processor, then we focus on a single processing core, inside the core among all the functional units we focus on the vector unit, here we see the SIMD sub-word parallelism capabilities. In this thesis we will focus on the lowest level of parallelism, i.e. SIMD sub-word parallelism which is found inside the vector unit of each core in a multi-core processor.

As seen from the figure 2.10, Intel intrinsics used in this project is the lowest level and most powerful method to vectorize an application. The figure gives an overview and classification of all the programming models available for the Many-Integrated Core architecture, which will be introduced in Chapter 4. The approaches are ordered according to the ease of implementation versus the degree of precise control available.

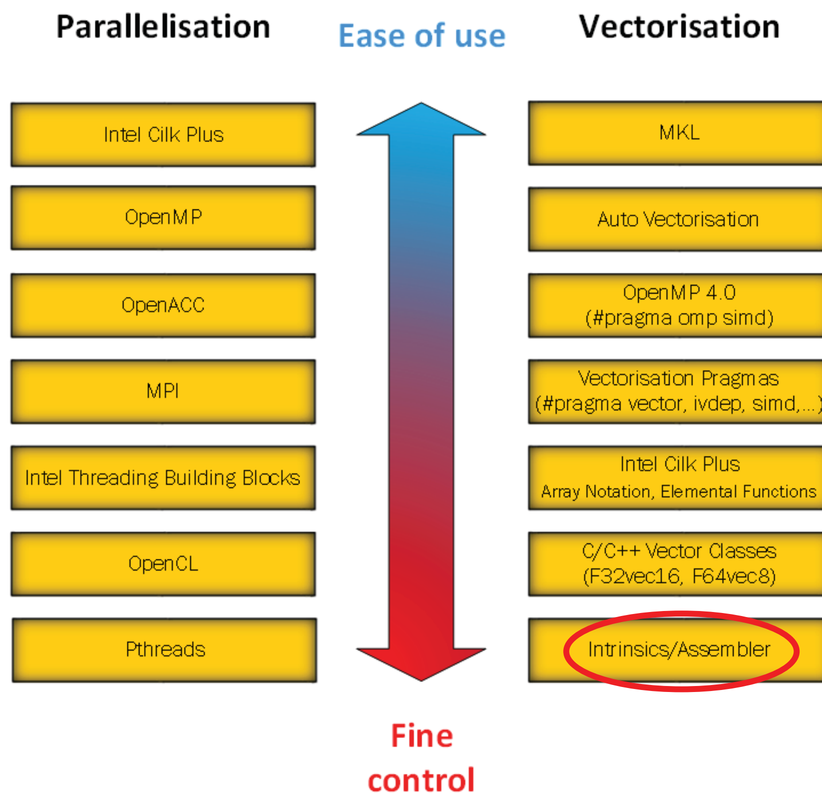


Figure 2.10: Many Integrated Core architecture programming models [18]

2.7 Related Work

Much work has been done on this topic of mapping complex image processing applications on the Intel Xeon Phi [26, 51, 15]. In addition, a number of performance comparison studies have been done comparing the MICs to the GPUs and CPUs [47]. Very few neural network algorithms in research are suited for achieving real-time performance on embedded platforms. Many of the FPGA based neural network accelerators have been unsuccessful due to the high design cost, poor flexibility and the competing increase of CPU and GPU performance [41].

The speed sign detection ConvNet has been implemented on a Nvidia GeForce GTX460 GPU at the PARSE group at TU Eindhoven, achieving 35 frames per second with an HD 720p video stream input [38]. Training of the total classification system is done off-line with the help of the error back-propagation algorithm.

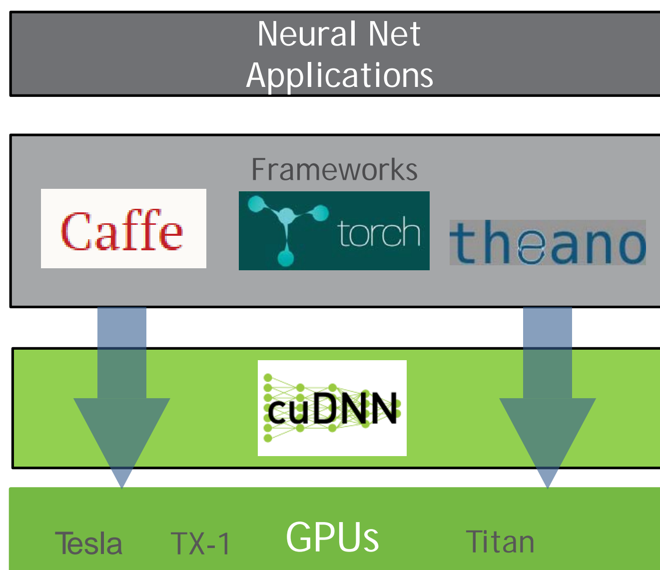


Figure 2.11: cuDNN is a library of primitives for deep learning applications on NVIDIA GPUs

Other approaches available in the computer vision community to accelerate neural net based applications include the use of deep learning frameworks such as Caffe [25], which offer an open-source library, public reference models and working examples for deep learning. Torch7 is another useful numeric computing framework and machine learning library[14], which is an environment to learn and experiment with machine learning like Matlab is for numerical computing. Theano is a python software package developed by machine learning researchers at the University of Montreal, which com-

piles symbolic code for different architectures including the CPU and GPU. It has performance comparable to hand-crafted C implementations on the GPU, for complex machine learning tasks like deep convolutional networks, which benefit from parallelization [11]. These frameworks can be better understood using the diagram 2.11, which shows their position in the overall image processing system; with the application at user level, the frameworks using the underlying cuDNN libraries to fully utilize the compute potential of the GPUs underneath.

Many of these frameworks use NVIDIAs cuDNN library in the back-end to implement many neural network specific operations. This enables these frameworks to accelerate their code on GPUs very efficiently, but they lack in performance on vector enabled CPUs and other new many-core architectures.

Chapter 3

ConvNet Speed Sign detection application

The computational workload in a neural networks prevents software implementations from achieving real-time recognition. The two main bottlenecks are computational workload and data transfer. Exploiting the available parallelism in ConvNets is essential to remove the computational scaling problems. In addition memory locality should be heavily exploited to reduce the excessive memory bandwidth which is also a bottleneck [41].

To solve the above problem, an optimized implementation of a speed sign detection ConvNet is needed. The application used in this project has some unique characteristics: First, it uses a modified algorithm to reduce the computational workload in CNN layers. Secondly, fixed point data types are used instead of float data type. This ConvNet has been trained using Error Back-Propagation and as per the results, it needs 65-83% less MACs (multiply-accumulate operations) than the original implementation, without loss of recognition quality [41].

In Section 3.1 we introduce the speed sign detection algorithm used in this project, followed by its C language implementation in Section 3.1.1. Finally, we analyze the applications computation and memory requirements.

3.1 Speed Sign detection Algorithm

The structure of the convolutional network used for speed sign detection can be seen in figure 3.1. The layers from Layer 1 to Layer 3 function as a trainable feature extractor. The last Layer 4 of this architecture are a fully connected feed forward Artificial Neural Network (ANN) layers. These first 3 layers are ANN layers with constraints to extract specific position invariant features from 2D images.

An example output of the speed sign detector ConvNet is shown in fig. 3.2. Here we can see the yellow bounding boxes marked in the image, around

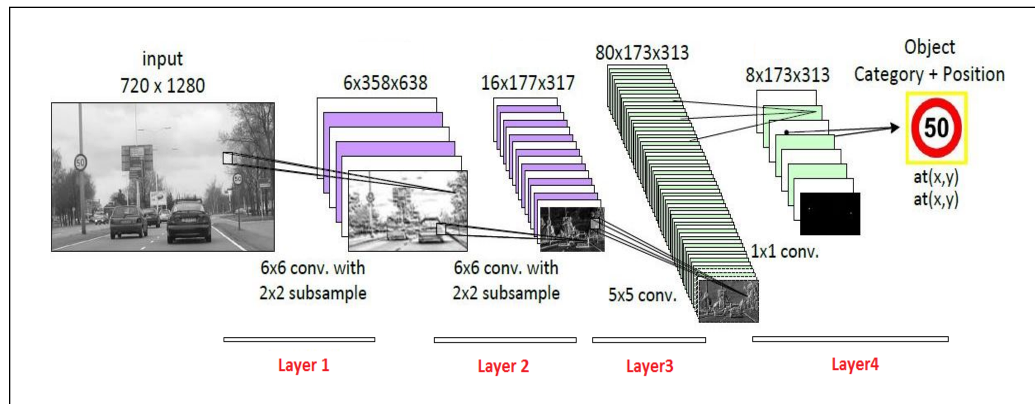


Figure 3.1: ConvNet layers representation [40]

Figure 3.2: Detector output for a 50 km/h speed sign [38].

the speed signs on the street and the correctly detected speed overlay on top of the image.

3.1.1 *C* language implementation of the speed sign ConvNet algorithm

The figure 3.3 shows pseudo code of a C language implementation the convolutions of the speed sign detection algorithm. One can relate this to the artificial neural network seen in Chapter 1 figure 2.1. The compute line, which is the multiply accumulate operation, is that part of the artificial neuron where there is a product of the each input ($in_layer[i]$) with the corresponding weights ($weight[i]$) and then summation (acc) of all these is passed through a sigmoid function ($fixact[j]$) and output as a detection is

```
1.  for(r=0; r<6; r++){
2.              acc = bias[r];
3.      for(m=0; m<YL1; m++){
4.          for(n=0; n<XL1; n++){
5.              for(k=0; k<6; k++){
6.                  for(l=0; l<6; l++){
7.  Compute ⇨           acc += in_layer[m*2,n*2,l]*weight[r,k,l];
8.                      }
9.                  }
10.             index=saturate_shift(acc); //10bit fixedpoint format
11.             out_layer[r,m,n]=fixact[index];  ⇦ Store
12.             }
13.     }
14. }
"r" = o/p feature maps (6)      "k*l" = 6*6 convolution kernel
"n" = Neuron outputs            fixact = sigmoid activation function
```

Figure 3.3: C kernel implementing the convolution operations

determined for every `n` neuron and stored collectively in ($out_layer[k]$) for each layer.

The k and l `for` loops determine the size of the convolution operation window. If `k` and `l` are equal to 6, then it is a 6x6 convolution operation, as seen in the fig. 3.3. As we go up to the outer `for` loops, `n` represents each individual neuron output. The `for` loop of `m` shifts the input window over input image and the `for` loop of `r` is use to jump to the next output feature map.

The call graph of the application is shown in fig. 3.4, which gives an idea of the structure of the code. The application uses a 720p HD video frame as input. As visualized in figure 3.1 there are three convolution and sub-sampling layers followed by a fourth layer for detection. The input for Layer 1 is a 1280x720 pixel resolution image, the input for Layer 2 are six, 637x357 pixel feature maps, input for Layer 3 are sixteen, 317x177 pixel feature maps and inputs for Layer 4 are eighty, 312x172 pixel feature maps. The Layer 4 is the detection layer, which outputs the detected speed of the speed sign.

In the algorithm, convolution and sub-sampling are merged as shown in

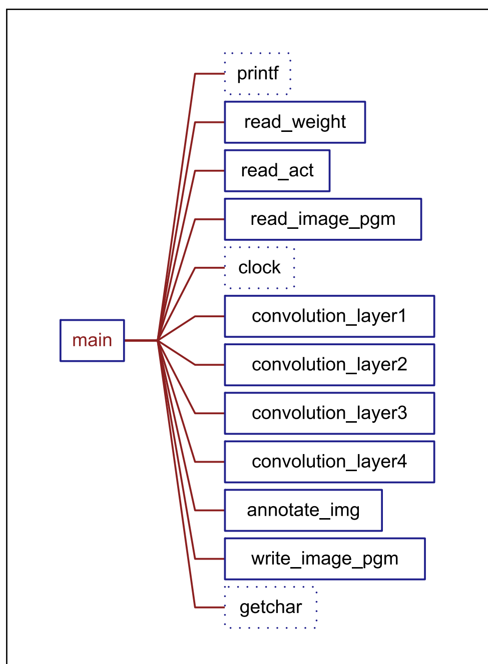


Figure 3.4: Calls graph of the speed sign application

equation 3.1, as derived in the paper [41].

$$y[m,n] = \phi(b + \sum_{q \in Q} \sum_{k=0}^{K-1} \sum_{l=0}^{K-1} v_q[k,l] x_q[mS+k, nS+l]) \tag{3.1}$$

In the above equation 3.1, y denotes the neuron output, ϕ denotes the activation function, b denotes the initial bias value required, v denotes the kernel and x is each input value. The set Q contains the indexes of input feature maps, which are connected to this feature map. The set of connected input feature maps can vary for different output feature maps. The constant K denotes the size of the convolution kernel and S represents the subsample factor, which is implemented as the step size of the input window, on an input feature map. This is illustrated in the figure 3.5, with **X** representing the input image, **v[k,l]** being the weight kernel and **y** representing the example output.

This equation is implemented in C code as follows:

```
acc = acc + in_layer[(m*2+k)*XIN+n*2+l] * weight[6*(r*6+k)+l]
```

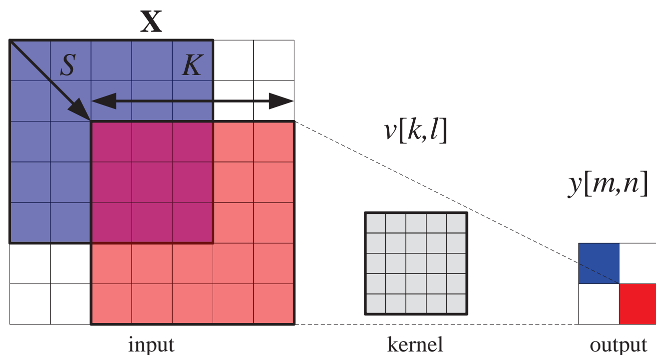


Figure 3.5: An example computation of feature map neurons [41]

3.2 Initial Experiments and Results

The CNN application code was profiled on an Intel x86 64 bit platform seen in fig. 3.6 (Core0 of Intel Xenon X5677 @ 3.47Ghz 24GB RAM). No x86 architecture specific optimizations performed. In the newer gcc 4.9.2 and clang LLVM 3.5.0 compilers unrolling of inner loops is automatically implemented at the `-O3` flag optimization level.

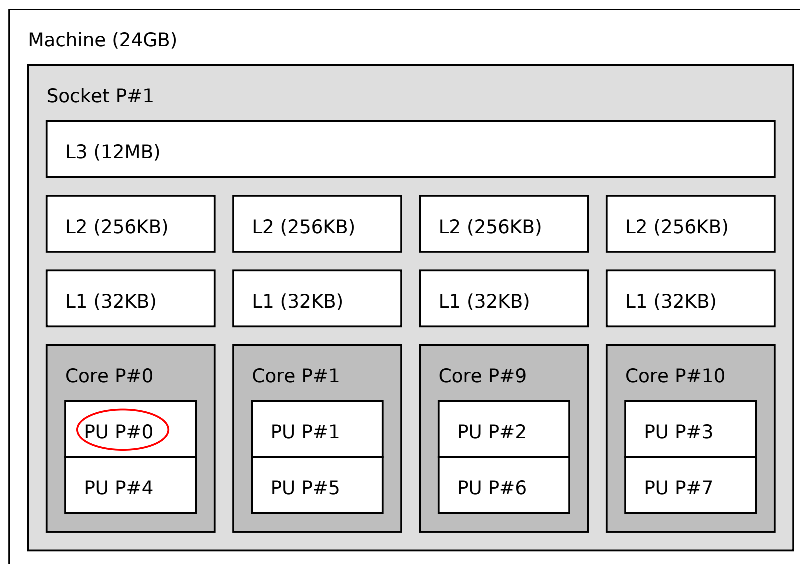


Figure 3.6: CPU topology of the initial x86 platform used for testing

The resulting split-up of the Multiply-Accumulate (MAC) operations per frame of the road speed sign video processed is shown in fig.3.7. This shows that Layer 3 is responsible for more than 80% of the MACs computation, hence this layer should be optimized at the highest priority.

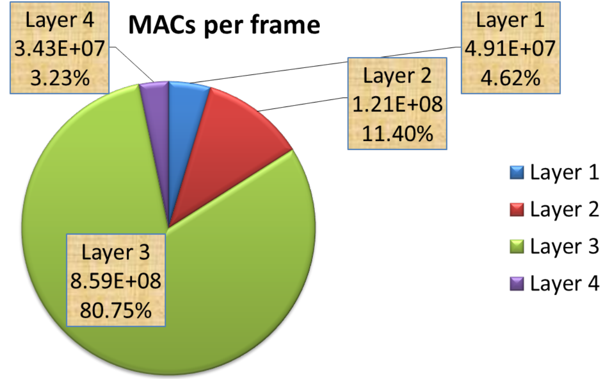


Figure 3.7: MACs split-up of the ConvNet application

3.2.1 Dimensioning computational requirements

Methodology followed to measure computations and data accesses:

```
acc = acc + in_layer[i_index] * weight[w_index]
```
(3.2)

Equation 3.2 is counted as one Mutiply Accumulate operation and two Read operations.

```
out_layer[o_index] = fixact[index]
```
(3.3)

Equation 3.3 is counted as one Read and one Write operations.

Computations and Memory accesses per frame

All the computation and memory requirements of the speed sign detection algorithm are summarized in table 3.1.

A video demonstration of this road speed sign detection application can be found at [48]. The second video demonstrates an easily extended version of the original algorithm which can now detect road crossings in addition to road speed signs. This demonstrates one of the advantages of having a trainable front end (feature extractor), which enables quick additions to the functionality by simply updating with newly trained weights. This expands the scope of its applications in the area of car navigation systems, as the feature extractor can use up to the minute information on road conditions to ensure driver safety.

Table 3.1: MAC and data transfer requirements of ConvNet speed sign detection (per frame)

Layers	**Parameters**				**MACs**	**Data transfer (MB)**
	No. of feature maps	**Feature map size**	**Kernel size**	**Sub-sample**		
L1	6	637x357	6x6	2x2	49.1M	2.25
L2	16	317x177	6x6	2x2	121.2M	2.17
L3	80	312x172	5x5	1x1	858,6M	4.98
L4	1	312x172	1x1	1x1	4.3M	4.1
TOTAL					**1.03G**	**13.5**

Chapter 4

Hardware Platforms

It is an interesting research question to see the suitability of a Xeon Phi platform for neural networks, as it might offer a better alternative to the GPU model most prevalent today. Possible benefits might include energy savings, easy portability of the optimized code to any Intel CPU, backward compatibility to x86 CPUs and a viable test platform for new many-core embedded platforms. As the programming model is same across these two Intel processors, the Core i7 was a good test bench for getting a baseline for the speed sign detection ConvNet.

In this chapter we introduce the hardware platforms used in this project. First, we introduce the Intel Core i7 processors in Section 4.1, followed by the Xeon Phi co-processor in Section 4.2. From this chapter you will have better knowledge of how the hardware capabilities of the processor influence a mapping and performance of a neural network.

4.1 Intel Core i7 - 5930K

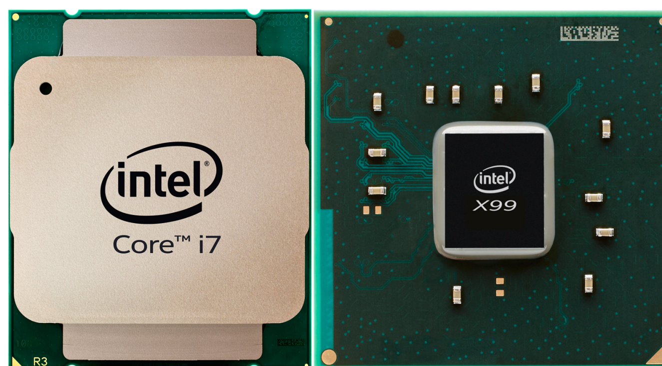


Figure 4.1: Intel Haswell-E Core i7 processor [6]

The Intel Core i7 5930K CPU is used as a baseline for testing opti-

Table 4.1: Superscalar architecture: Operations can be dispatched on different ports [10]

Port 0	Port 1	Port 5	Port 6	Port 4	Port 2	Port 3	Port 7
ALU	ALU	ALU	ALU	Store_data	Load_Addr	Load_Addr	Store_addr
Shift	LEA	LEA	Shift		Store_addr	Store_addr	AGU
	BM	BM	JEU				
SIMD_Log	SIMD_ALU	SIMD_ALU					
SIMD_Shifts	SIMD_Log	SIMD_Log					
SIMD misc							
FMA/FP_mul	FMA/FP_mul	Shuffle					
Divide	FP_add						
JEU	slow_int	FP mov					

LEA : Load Effective Address, BM : Bit Manipulation, JEU : Jump Execution Unit (Branch Unit)
SIMD_Log : Vector Logical instructions, FMA: Fused Multiply Add, AGU : Address Generation Unit
FP: Floating point

mizations before moving on to the Intel Xeon Phi. The Intel literature recommends this approach of optimizing for high end Intel processors before moving on to the Intel Xeon Phi as most of the optimizations scale well to the many-core Xeon Phi.

The Core i7 5930K is a 64bit CPU with Haswell-E micro-architecture. It has 6 physical cores with 2 Threads per core (12 logical cores), including 32K bits Level 1 (L1) data and instruction cache each, 256K bits of Level 2 (L2) cache and 15360K bits (15MB) of Level 3 (L3) cache. It has a 256 bit wide vector unit capable of running the AVX2 instruction set extensions along with AVX, FMA3, SSE, SSE2, SSE3, SSSE3, SSE4, SSE4.1, SSE4.2 x86-64 and Intel 64 instructions.

The Haswell-E architecture is wider than the previous generation Ivy Bridge architecture, with four arithmetic logic units (ALU), three address generation units (AGU), two branch execution units, higher cache bandwidth, improved memory controller with a higher load/store bandwidth. The hardware functional units are visualized in the Table 4.1, which represents the superscalar parallelism in this processor. To give an idea of the capabilities of each functional unit, some example instructions for each functional unit are listed in Table 4.2.

It has a maximum of 32 parallel SIMD lanes of 8-bits each, as seen in orange in the fig. 4.2. All the data-types supported by the Haswell architecture and AVX2 instruction set combination are shown in this figure. This is one of the important decision metrics when selecting the optimal instructions for implementing the ConvNet kernel onto this architecture.

Figure 4.2: SIMD Vector Data Types

Table 4.2: Haswell microarchitecture Execution Units and example instructions [10]

Execution Unit	No. of Ports	Instructions
ALU	4	add, and, or, xor, mov, (v)movdqu
SHFT	2	sal, shl, rol, adc, sarx, (adcx, adox)
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep
SIMD Log	3	(v)pand, (v)por, (v)movq, (v)blendp
SIMD_Shift	1	v)psl*, (v)psr*
SIMD ALU	2	(v)padd*, (v)pabs, (v)pavgb, (v)pmax
Shuffle	1	(v)shufp*, vperm*, (v)pack*, vbroadcast, (v)pblendw
SIMD Misc	1	(v)pmul*, (v)pmadd*, (v)bendv
DIVIDE	1	divp*, vdiv*, sqrt*, vsqrt*, rsqrt*, idiv
FP Add	1	(v)addp*, (v)cmpp*, (v)max*, (v)min*,
FP Mov	1	(v)movap*, (v)movup*, (v)movsd/ss, (v)movd gpr
BM	2	andn, bextr, blsi, blsmsk, bzhi, etc

4.2 Intel Xeon Phi - 31S1P

Intel announced the Xeon Phi brand at the International Supercomputing Conference, Hamburg in June 2012 [5]. Here the first supercomputer with Xeon Phi co-processors named "Discovery" made the TOP500 list [21].

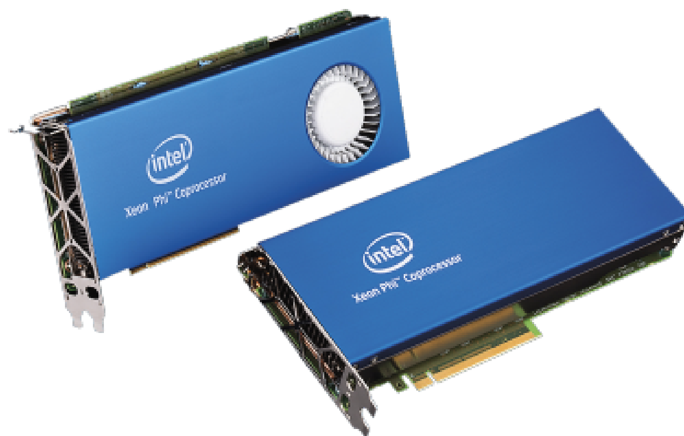


Figure 4.3: Intel Xeon Phi family of coprocessor [4]

In November 2012, the Green500 list named an Intel Xeon Phi based system as the world's most energy-efficient supercomputer [23]. This indicates that the Xeon Phi has good potential in accelerating compute intensive tasks like neural networks algorithms, in an energy efficient manner.

The Intel Xeon Phi Co-processor is in the form factor of a PCIe-16x extension card, which runs its independent Linux operating system and can be used by a host system either as a separate processing system or via offloading sections of the code to the co-processor.

Intels' first commercial product using the Many Integrated Core architecture (MIC) was named Knights Corner. In this architecture Intel introduced a very wide 512-bit SIMD unit to the processor architecture, a cache-coherent multiprocessor systems connected to the memory via a ring bus, with each core supporting 4-way multi-threading. This can be seen in fig.4.4, with 4 of 8 graphics double data rate (GDDR) memory controllers (MC), tag directories (TD) which look up cache data distributed among the cores, a ring interconnect, and the cores with separate L2 caches. Intel MIC KNC processors have 32 native registers, 60+ in-order low power IA cores (similar to Intel Pentium) on a high-speed bi-directional ring interconnect, which also enables fully coherent L2 cache. It has two pipelines with dual issue on scalar instructions, it's scalar unit is based on Pentium processors upgraded to Intel 64 ISA, 4-way SMT, new vector instructions, and increased cache sizes. The scalar throughput after pipelining is one-per-clock cycle. It has a 512bit SIMD Vector Processing Engine, with 4 hardware threads per core and 512KB coherent L2 Cache per core. It has 8 memory controllers

with 2 channels each to the GDDR5 memory on-board.

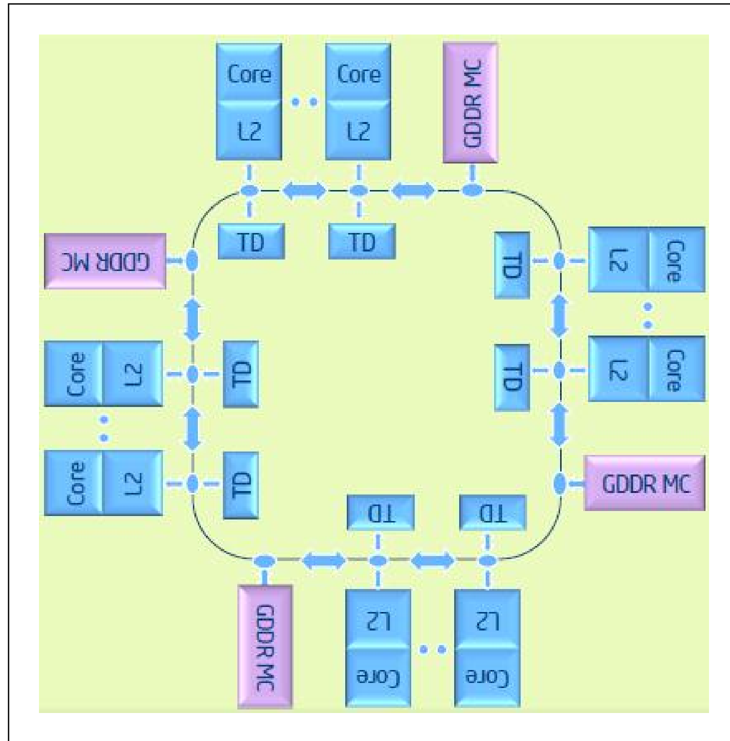


Figure 4.4: Intel Xeon Phi micro-architecture with interleaved memory controllers [9]

The Intel Xeon Phi 31S1P used during in this project is capable of 1 TeraFLOPs double precision performance with its 57 cores running at 1.1GHz each, it has 8GB GDDR5 memory with a speed of 5 GT/s, with a total bandwidth of 320GB/s and has a total cache size of 28.5MB [24]. One thing lacking is L2 to L1 cache auto pre-fetching. See fig. 4.5 for some detailed specifications. Due to a simple Instruction Decoder it needs two threads per core to achieve full compute potential for the same instruction.

The Intel Xeon Phi has the potential to save time and resources compared to other coprocessors in the market, because it uses familiar programming languages, parallelism models and tools used for existing Intel processors. This facilitates easy migration of code developed for older Intel architectures to this new MIC architecture [36].

4.2.1 Many Integrated Core micro-architecture

At the heart of the MIC architecture is a combination of a scalar and vector processor, as seen in fig.4.6. The scalar part of the core is mainly to maintain the backward compatibility and easy porting of operating systems.

SKU #	Form Factor, Thermal	Board TDP (Watts)	Max # of Cores	Clock Speed (GHz)	Peak Double Precision (GFLOP)[1]	GDDR5 Memory Speeds (GT/s)	Peak Memory BW	Memory Capacity (GB)	Total Cache (MB)
5110P	PCIe Card, Passively Cooled	225	60	1.053	1011	5.0	320	8	30
31S1P	PCIe Card, Passively Cooled	270	57	1.1	1003	5.0	320	8	28.5
3120P	PCIe Card, Passively Cooled	300	57	1.1	1003	5.0	240	6	28.5

Figure 4.5: Intel Xeon Phi 31S1P specs (Co-processor used for testing) [24]

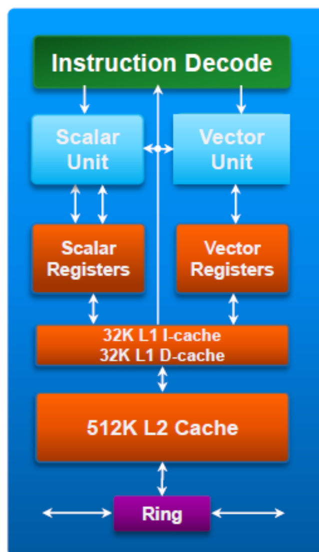


Figure 4.6: Intel Knights Corner core architecture [7]

The actual compute power is in the vector processing part of the core. The vector processor has a 512bit SIMD unit with capability to process 16 words per operation.

We can see the detailed block diagram of the Intel MIC architecture core in the figure 4.7. We can clearly see that the standard scalar IA (Intel Architecture) core has been extended with a 512bit vector unit. The vector unit contains 16 SP (single precision) ALUs and 8 DP (double precision) ALUs. Most of the vector instructions have a latency of 4 cycles and throughput of 1 cycle and are issued in the u-pipeline. Some vector instructions like mask, store, prefetch are issued in v-pipe [7].

The Xeon Phi's IMCI instruction set supports 16x 4-bytes elements or 8x 8-bytes elements. As illustrated in the figure 4.8, the Knights Corner MIC architecture does not natively support 8 bit or 16 bit data types, which

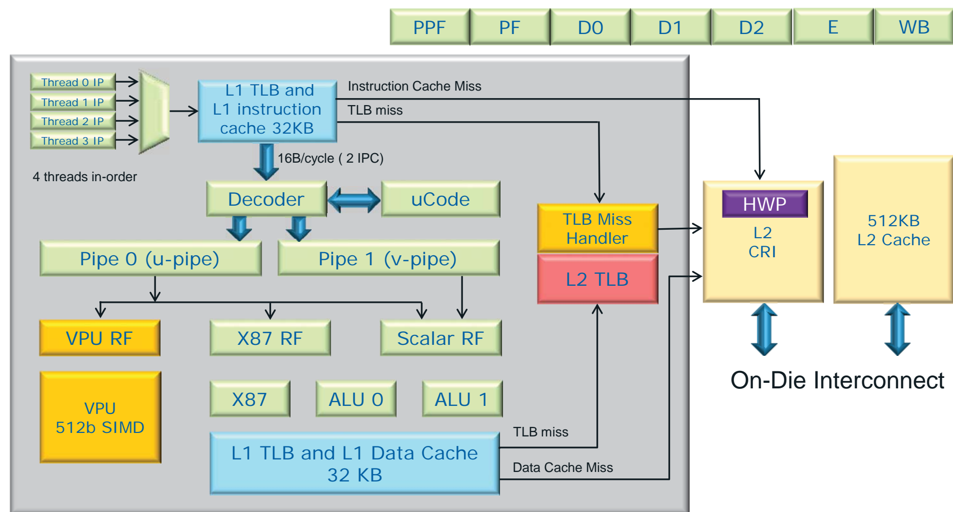


Figure 4.7: MIC single core architecture [7]

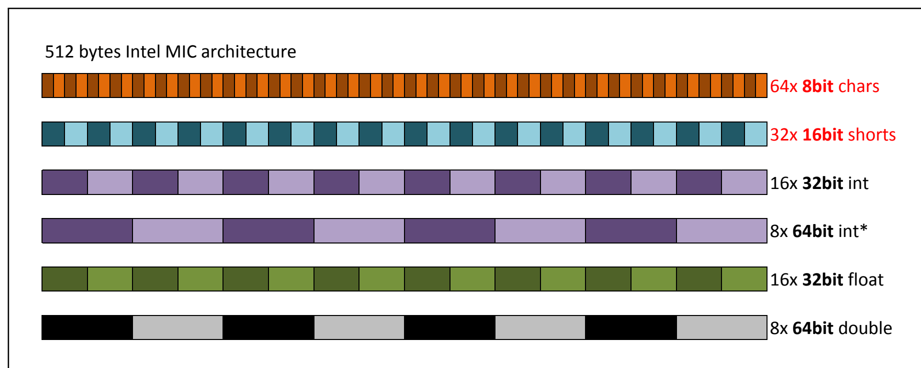


Figure 4.8: Xeon Phi MIC 512-bit vector data-types of IMCI

are supported in the Haswell architecture found in Core i7. This puts the KNC MIC at a slight disadvantage for algorithms requiring byte operations (which most pixel processing algorithms need), as it has to up-convert such data types to 32bit before it can operate on them. This is a hardware limit as there are only 16 x 32 bit lanes in the SIMD vector unit in the KNC MIC architecture.

4.3 Sub-word parallelism

Sub-word parallelism can simply be understood as Single Instruction Multiple Data (SIMD) within a register [19]. Sub-word parallelism enables us to work with lower-precision data on architectures which have word-sized data paths. It is a form of SIMD parallelism at a smaller scale and at a

cheaper hardware cost as one register can hold multiple data operands [30].

Vector instructions are extensions to the x86 instruction set architecture for microprocessors from Intel and AMD. Advanced Vector Extensions (AVX) were first supported by Intel with the Sandy Bridge processor in Q1 2011 and later on by AMD with the Bulldozer processor in Q3 2011. AVX2 expands most integer commands to 256 bits and introduces Fused Multiply Add (FMA).

Vector SIMD registers

The AVX-512 (ZMM0-ZMM31) register scheme is an extension of the AVX (YMM0-YMM15) registers and SSE (XMM0-XMM15) registers, as visualized in the table 4.3. We can see that the SSE vector instructions started with 128-bit registers, which were extended to 256-bits by the AVX instruction set, then the IMCI instruction set further extended 16 registers to 512-bits but did not maintain backward compatibility to AVX and finally the latest AVX-512 instructions added 16 more registers while bringing back backward compatibility to old vector extensions including AVX and SSE.

4.4 Summary

The Xeon Phi co-processor with it's MIC architecture has been designed for highly parallel workloads, while Core i7 with Haswell architecture is a general purpose CPU. The table below indicates which type of workload is suitable for each type of processor.

In the next two chapters 5 and 6, we evaluate the platforms and programming approaches used to map the ConvNet application. In the case of processor platforms, the layers of speed sign detection application are mapped onto the two processors natively and the execution time is used as the unit of performance. Similarly, for the programming approaches, the speed sign detection application is written using Intel intrinsics and compared with the automatically generated executable from the original C implementation of the application.

Table 4.3: AVX-512 (ZMM0-ZMM31), AVX (YMM0-YMM15) and SSE (XMM0-XMM15) registers

511 .. 256 bits	**255 .. 128** bits	**127 .. 0** bits
ZMM0	***YMM0***	**XMM0**
ZMM1	***YMM1***	**XMM1**
ZMM2	***YMM2***	**XMM2**
ZMM3	***YMM3***	**XMM3**
ZMM4	***YMM4***	**XMM4**
ZMM5	***YMM5***	**XMM5**
ZMM6	***YMM6***	**XMM6**
ZMM7	***YMM7***	**XMM7**
ZMM8	***YMM8***	**XMM8**
ZMM9	***YMM9***	**XMM9**
ZMM10	***YMM10***	**XMM10**
ZMM11	***YMM11***	**XMM11**
ZMM12	***YMM12***	**XMM12**
ZMM13	***YMM13***	**XMM13**
ZMM14	***YMM14***	**XMM14**
ZMM15	***YMM15***	**XMM15**
ZMM16	YMM16	XMM16
ZMM17	YMM17	XMM17
ZMM18	YMM18	XMM18
ZMM19	YMM19	XMM19
ZMM20	YMM20	XMM20
ZMM21	YMM21	XMM21
ZMM22	YMM22	XMM22
ZMM23	YMM23	XMM23
ZMM24	YMM24	XMM24
ZMM25	YMM25	XMM25
ZMM26	YMM26	XMM26
ZMM27	YMM27	XMM27
ZMM28	YMM28	XMM28
ZMM29	YMM29	XMM29
ZMM30	YMM30	XMM30
ZMM31	YMM31	XMM31

Table 4.4: Prediction for the two processors

Workload type	**Xeon Phi**	**Core i7**
Sequential	Low	High
Data-parallel	Medium	High
Memory intensive	High	Medium

Chapter 5

ConvNet Mapping on the Core i7

In this chapter, the performance of a Core i7 is evaluated, as a baseline for further tests. This is done by porting the speed sign detection algorithm elaborated in Chapter 3 to a computer with the Core i7 processor. This application is accelerated using platform specific hardware features to measure the peak performance that can be achieved. The performance of these software optimizations are plotted on a Roofline model, which gives us a clear comparison with the theoretical limits of the platform.

In Section 5.3, the resulting performance of the ConvNet algorithm is discussed. By the end of this chapter you will learn some of the tricks and methods which can be used to accelerate neural networks on wide SIMD path platforms.

5.1 A Brief on the Intel intrinsics format

Intel intrinsic instructions, are C style functions that allow the programmer to easily integrate assembly instructions into C language source code, without the need to write assembly code. They provide access to many Intel instructions - including SSE, AVX, AVX-512, IMCI etc.

There are six main vector types which are used in the AVX, AVX2 and IMCI instruction sets, as shown in Table 5.1. Xeon Phi supports int32, int64, float32 and float64 elements. Normal C language data types like `int` and `float` are also used in some intrinsics [44].

Table 5.1: Vector Data types

Data Type	Description
__m256i	256-bit vector containing integers (8, 16, 32 bit)
__m256	256-bit vector containing 8floats (32 bit)
__m256d	256-bit vector containing 4doubles (64 bit)
__m512i	512-bit vector containing integers (32, 64 bit)
__m512	512-bit vector containing 16floats (32)
__m512d	512-bit vector containing 8doubles (64 bit)

5.2 Method to Vectorize a Convolutional Kernel on the Haswell

The first step, when we decide to vectorize an application using SIMD intrinsics is to survey the instruction set and computational capabilities of the specifice processor. In this case it is the Core i7 with AVX2 instruction set supporting FMA operations. The we search for the most efficient way to implement our target computation, in this case multiply-accumulate.

As we know from Chapter 4 the Core i7 has Haswell microarchitecture which supports AVX2 256-bit vector intrinsics. In this instruction set the most efficient instruction for a multiply-accumulate operation is the Fused Multiply Add instruction. As doing a multiply takes 5 clock cycles, an add takes 1 clock cycle while a multiply-accumulate instruction takes 5 clock cycles, thereby saving the cycle for add operation, the data transfer and reducing dependency between different hardware units.

The next step is to find the most efficient way to gather data required for this MAC operations. There are many load instructions available but we must aim to increase the number of computations achieved for every byte of data loaded. Thus vector load instructions are the most efficient.

After loading the data, we need to arrange it in the particular order required for each iteration of the loop. This can be done using the permute and shuffle intrinsics. There is a significant exploration, as there are different ways to achieve the same result, but we must aim to minimize the number of cycles.

5.2.1 Fused Multiply Add Intrinsic

The throughput of this FMA instruction is 1 clock cycle, which means that we can have a new FMA result every clock cycle, effectively doing two operations (multiply and add) per clock cycle. FMA instructions are not only limited to convolution kernels but are used extensively in matrix-vector

and matrix-matrix product operations.

```
Intrinsic:__m256i _mm256_madd_epi16 (__m256i a, __m256i b)

Instruction: vpmaddwd ymm, ymm, ymm
```

Multiply packed signed 16-bit integers in a and b, producing intermediate signed 32-bit integers. Horizontally add adjacent pairs of intermediate 32-bit integers, and pack the results in dst. This operation is visually explained in the figure 5.2 and the pseudo code of this operation is presented in illustration 5.1.

```
FOR j := 0 to 7
    i := j*32
    dst[i+31:i]:= a[i+31:i+16]*b[i+31:i+16] + a[i+15:i]*b[i+15:i]
ENDFOR
dst[MAX:256] := 0
```

Figure 5.1: `madd()` operation explained with pseudo code

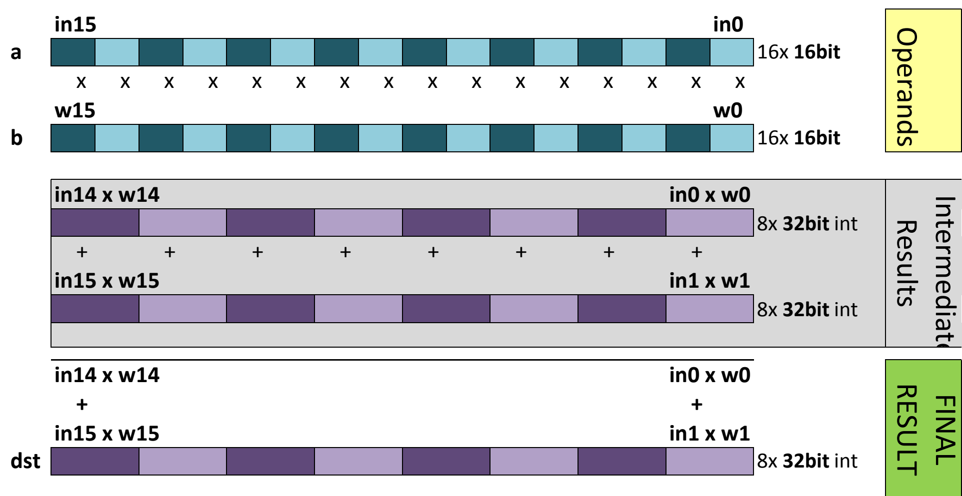


Figure 5.2: Details of `madd()` Fused Multiply-Add operation

On the Haswell architecture this instruction has a Latency of 5 cycles and a Throughput of 1, which means that ideally there can be one MAC (multiply-accumulate) result produced every cycle.

5.2.2 Gathering and Arranging data for FMA

First we have to convert `in_layer[]` from unsigned 8 bit data-type to signed 16 bit data-type. `weight[]` array is already of the required data-type of 16 bit signed integer.

- **Naive approach**
  Using `set` intrinsic present in the AVX instruction set, as seen from the pseudo code in illustration 5.3, we can manually provide each operand for the vector operation.

  The pseudo code for this instruction is shown in fig. 5.3, which shows how packed 16-bit integers are store in destination address `dst` with the supplied values. This is a sequential operation of loading each value individually and hence very inefficient and wasteful considering the large amount to memory operations required.

```
dst[15:0] := e0
dst[31:16] := e1
dst[47:32] := e2
dst[63:48] := e3
dst[79:64] := e4
dst[95:80] := e5
dst[111:96] := e6
dst[127:112] := e7
dst[145:128] := e8
dst[159:144] := e9
dst[175:160] := e10
dst[191:176] := e11
dst[207:192] := e12
dst[223:208] := e13
dst[239:224] := e14
dst[255:240] := e15
dst[MAX:256] := 0
```

Figure 5.3: `set()` operation explained with pseudo code

- **Optimized approach**
  Using `loadu` intrinsic, followed by permute and shuffle intrinsics to arrange the data in the vector register, as per the algorithm requirements. This method does much more efficient loads as it can loads 256-bits of integer data from memory into destination vector register.

```
dst[255:0] := MEM[mem_addr+255:mem_addr]
dst[MAX:256] := 0
```

Figure 5.4: `loadu()` operation explained with pseudo code

Layer 1 optimized convolution kernel steps:

1. Broadcast initial bias value to all 16 entries of a vector register (macresult0)
2. Load the 32 bytes of raw data of the input image into a vector register (in0) using *loadu* intrinsic
3. Load the 32 bytes of raw data of the weights into a vector register (w0) using *loadu* intrinsic
4. Permute the data loaded in in0 register to an intermediate format.
5. Permute the data loaded in w0 register to an intermediate format.
6. Shuffle data in in0 register as per pre-calculated indexes and the required pattern of input image data is ready in the register.
7. Shuffle data in w0 register as per pre-calculated indexes and the required pattern of the weights is ready in the register.
8. Perform the multiply-accumulate operation on registers in0, w0 and macresult0
9. Sum the result of the current operation with the previous result, in the first iteration it is the bias values.
10. Extract the resulting eight 32-bit accumulated value (`acc`) one by one.
11. Saturate each value to fit within 10-bit precision and calculate corresponding activation function index using if conditions.
12. Look up activation function value in the table and write to output pixel.

In the table 5.2, we can see that the ratio of actual compute operations to the total number of operations needed for this operation is 0.26. Which means that only 26% of the cycles are spent in actual compute and the rest 74% are overhead. This is one of the major factors which prevents the vector SIMD implementation from reaching the roofline. Admittedly, this is a simplistic was of looking at the problem, an there are many factors that play a role here, pipelining, hardware unit throughput, data dependency etc.

Table 5.2: Analysis of one AVX intrinsic kernel implementation

Intrinsic	**Cycles**	**Description**
loadu	3	Load 32 bytes
loadu	3	
permute	3	Permute data in a register
permute	3	
shuffle	1	Shuffle data within a register
shuffle	1	
madd	**5**	Multiply-accumulate intermediate
add	**1**	Accumulate final result
extract	3	Extract results from vector register
TOTAL	23	
Ratio Compute/Total		0.26087

5.3 Results

The results of the code optimizations using vector intrinsics, loop unrolling and compiler optimizations are shown in this section. Tables 5.5 and 5.6 document the actual speedup achieved in execution time after using SIMD intrinsics over the default non-vectorized code generated by the compilers.

Table 5.3: ICC execution times of the speed sign detection application

ICC	Execution time (ms)		
	Vectorization off	Auto-vectorized	Manually optimized
Layer 1	20.90	21.86	**4.43**
Layer 2	46.06	91.24	**8.06**
Layer 3	243.62	244.13	**50.21**
Layer 4	7.67	2.87	**1.17**
Complete	318.24	360.09	**63.87**

Table 5.4: GCC execution times of the speed sign detection application

GCC	Execution time (ms)		
	Vectorization off	Auto-vectorized	Manually optimized
Layer 1	20.39	20.39	**4.33**
Layer 2	57.48	57.61	**8.38**
Layer 3	322.73	322.90	**48.32**
Layer 4	2.25	1.28	**2.49**
Complete	402.85	402.19	**63.52**

Note:
This is the only case where an auto-vectorizer successively improves throughput of the complete application. Also this is the only case where in Layer 4 the manually optimized code is slower than compiler generated code.

Table 5.5: Speedup achieved with Intel C Compile

Layers	**Speedup** over Un-vectorized code	**Speedup** over Auto-vectorized code
Layer 1	4.7x	4.9x
Layer 2	5.7x	6x
Layer 3	4.8x	4.8x
Layer 4	6.6x	2.5x
Complete Application	5.6x	5x

5.4 Roofline model: Performance evaluation

Actual multiply accumulate operation in the C code:

```
acc = acc + in_layer[i]*weight[j]        (5.1)
```

The `add` and multiply accumulate (`madd`) intrinsics are used to implement this operation as follows: `add(acc, madd(in_layer,weight))`. The bytes of data loaded from memory for the operands in this operation are as follows: `in_layer[i]` = 1byte ; `weight[j]` = 2bytes. The operational intensity of these loads versus the two arithmetic operations of multiply and add can be calculate as: 2ops/3bytes = 0.67 Ops/Byte

Table 5.6: Speedup achieved with GCC compiler

Layers	**Speedup** over Un-vectorized code	**Speedup** over Auto-vectorized code
Layer 1	4.7x	4.7x
Layer 2	6.8x	6.8x
Layer 3	6.7x	6.7x
Layer 4	0.9x	0.5x
Complete Application	6.34x	6.3x

5.4.1 Calculation of Compute Roofline Haswell

Theoretical:

```
2 Ops/cycle (FMA) * 8 (32bit operands) * 3.5GHz
          * 1 (SIMD unit for FMA) = 56 GOps/s        (5.2)
```

5.4.2 Calculation of Memory Roofline Haswell

Aggregate Bandwidth (BW) to DDR4 RAM

Theoretical:

```
(2133MHz*64bits*4channels)/8 = 68.256 GBytes/s        (5.3)
```

Read bandwidth to L1 cache

Theoretical:

```
3.5GHz *32byte *(2 Read units) = 224 GBytes/s        (5.4)
```

Write bandwidth to L1 cache

Theoretical:

```
3.5GHz*32byte*(1 Write unit) = 112GBytes/s        (5.5)
```

The Roofline model for a single core of the Core i7 processor is shown in figure 5.5.

5.5 Analysis of Results

As seen in fig. 5.5, the best mapping on the Core i7 processor is hand-optimized Layer 3 of speed sign detection, which is able to achieve 35.54 GOps/sec. This still leaves a 3.2x performance gap to the theoretical limit

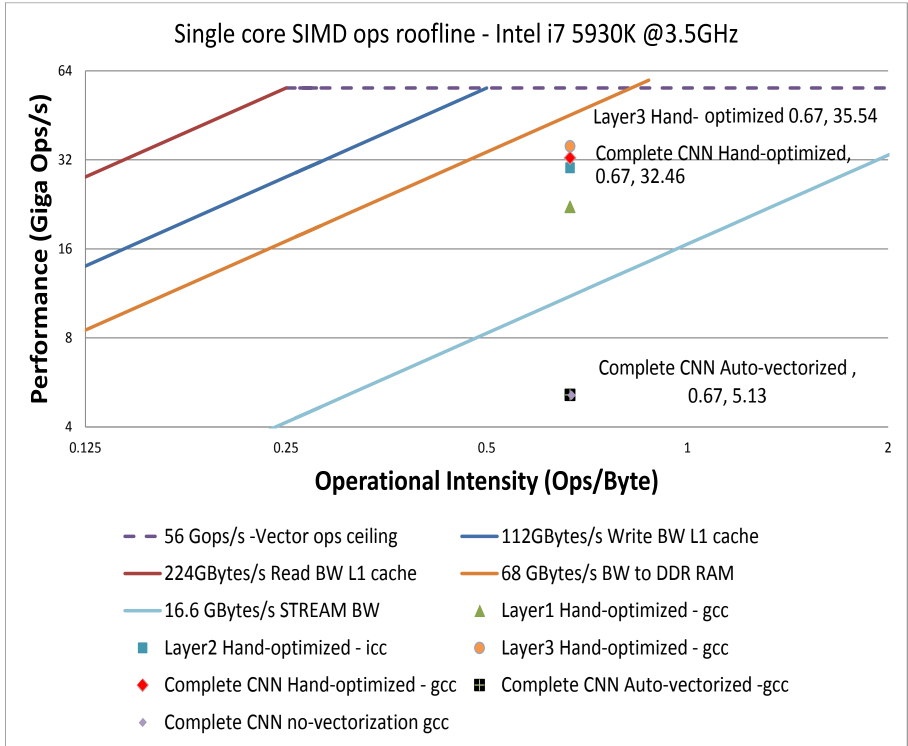


Figure 5.5: Core i7 single core Roofline: Complete ConvNet application

of the platform, which is 112 GOps/sec. This gap is wider for other mappings ranging from 3.7x to 5x less than theoretical performance.

One of the reasons for this performance gap is the ratio of effective computational instructions versus the total instructions. The actual computation instructions are only about 30% of the total instructions in the kernel, the rest of the instructions are for loading, arranging, extracting and storing back the data. The data dependencies between the operations, further exacerbates this problem. For eg. the add instruction used to accumulate the product, has to wait for the multiply units to provide the results.

In the figure 5.5 we can observe an irregularity that the manually optimized application points lie above the measured STREAM Triad bandwidth (represented by the blue line). This is because the operational intensity is based upon the instructions and not the complete application. If we use the applications operational intensity all points will move towards the right in the roofline fig. 5.5. If we calculate the operational intensity of the application from table 3.1, it would be about 76 Ops/Byte which is far more than the 0.67 used in the Roofline now.

Chapter 6

ConvNet Mapping on the Xeon Phi

This chapter goes into depth of the porting the convolutional network application onto the Intel Xeon Phi co-processor. Towards the end of this chapter the resulting performance of the ConvNet algorithm is discussed and reasons are put forward. By the end of the chapter you will know some tricks and methods which can be used to accelerate neural networks on wide SIMD path platforms.

6.1 Going from Core i7 to Xeon Phi

As illustrated in the figure 6.1, the Knights Corner MIC architecture does not natively support 8 bit or 16 bit data types, which are supported in the Haswell architecture found in Core i7. This puts the KNC MIC at a slight disadvantage for algorithms requiring byte operations (which most pixel processing algorithms need), as it has to up-convert such data types to 32bit before it can operate on them. This is a hardware limit as there are only 16 x 32 bit lanes in the SIMD vector unit in the KNC MIC architecture.

Table 6.1: An example of corresponding vector instructions in Core i7 and Xeon Phi

Core i7 - Haswell (AVX) 256 bit SIMD vector	**Xeon Phi - MIC KNC (IMCI)** 512 bit SIMD vector
_mm256_loadu_si256	_mm512_load_si512
_mm256_permute2x128_si256	_mm512_mask_permute4f128_epi32
_mm256_shuffle_epi8	_mm512_shuffle_epi32
_mm256_madd_epi16	_mm512_fmadd_epi32
_mm256_extract_epi32	_mm512_kextract_64
_mm256_mulhi_epi16	_mm512_mulhi_epi32

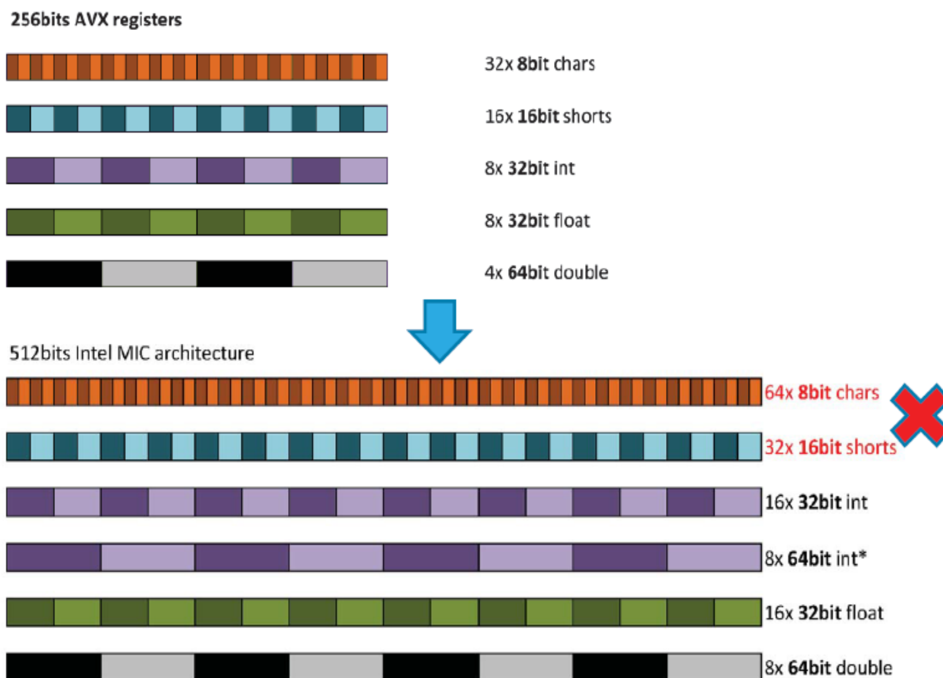


Figure 6.1: Going from Core i7 to Xeon Phi (Haswell to KNC)

6.1.1 Multiply-Accumulate instruction

The Intel Initial Many Core Instruction set (IMCI) provides a very efficient instruction which combines the multiply and add operations into a single instruction cycle. This hybrid instruction is called Multiply-Accumulate corresponding assembly instruction is "`vpmadd231d`" which can be accessed using the intrinsic "`fmadd()`"

```
Intrinsic:
__m512i _mm512_fmadd_epi32 (__m512i a, __m512i b, __m512i c)

Instruction:  vpmadd231d zmm {k}, zmm, zmm
```

Multiply packed 32-bit integer elements in **a** and **b** and add the intermediate result to packed 32 bit elements in **c** and store the results in **dst** destination register. This operation is visually explained in the figure 6.4 and the pseudo code of this operation is presented in illustration 6.3.

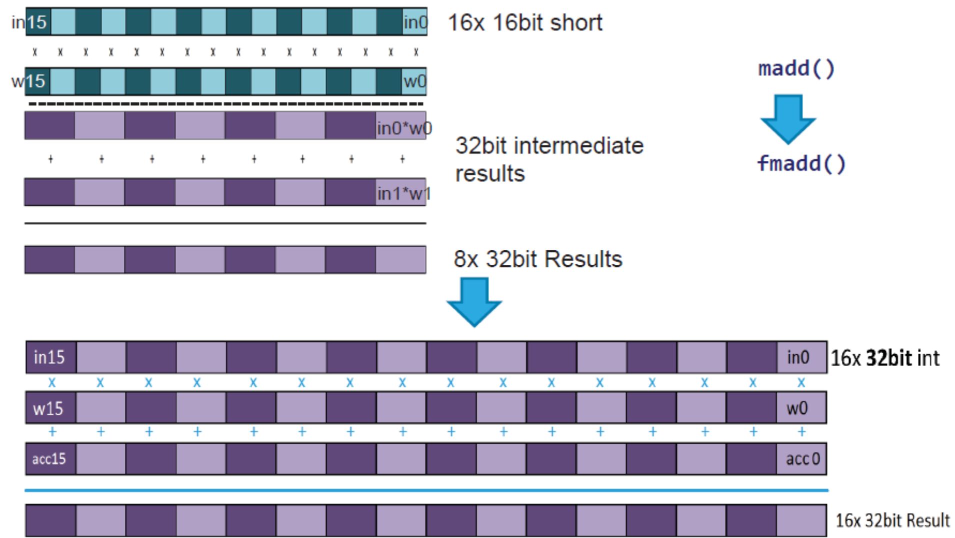


Figure 6.2: Going from AVX to Initial Many Core Instruction set

```
FOR j := 0 to 15
    i := j*32
    dst[i+31:i] := (a[i+31:i] * b[i+31:i]) + c[i+31:i]
ENDFOR
dst[MAX:512] := 0
```

Figure 6.3: `fmadd` operation pseudo-code

6.2 Method to Vectorize a Convolutional Kernel

Single core SIMD vectorized mappings of the ConvNet algorithm on the Xeon Phi are discussed in the following sections. These optimizations are done using the Initial Many-core instruction (IMCI) set intrinsics.

Three main steps to vector operations are Gather, Compute and Scatter. We try to minimize the overhead in gather and scatter operations as they are just the overhead necessary to make the vector computations possible.

Layer 1 optimized convolution kernel steps (see figure 6.5):

1. Broadcast initial bias value to all 16 entries of a vector register (macresult0)

2. Load the 16 values of the input image into a vector register (in0) using *gather* intrinsic (`_mm512_i32extgather_epi32`)

3. Load 16 values of weights into a vector register (w0)

4. Perform the multiply-accumulate operation on registers in0, w0 and macresult0
5. Extract the resulting 32-bit accumulated value (`acc`) one by one.
6. Saturate each value to fit within 10-bit precision and calculate corresponding activation function index using if conditions.
7. Look up activation function value in the table and write to output pixel.

Table 6.2, shows a simplistic analysis of the ratio of total intrinsics required, versus the actual multi-ply accumulate intrinsic (`fmadd`). This ratio is very poor at 0.08 which is just 8%. This shows that even in a manual optimized implementation of the kernel, there is massive overhead required to gather and scatter the data before and after the compute operation. This points to a clear inefficiency in the IMCI instruction set which does not provide the required smaller data-types (byte and short) for this kernel.

As the Xeon Phi does not have an out-of-order execution core, we interleave load and multiply-accumulate operations manually, in order to reduce stalls in the pipeline, where the computation is waiting for operand data to arrive.

Table 6.2: Simple analysis of actual compute intrinsics in a kernel

Intrinsic	**Cycles**	**Description**
extload	7 (load) + 6 (convert)	load and up-convert operands
extgather	8(load) + 6 (convert)	load scattered data and up-convert
extload	7 (load) + 6 (convert)	load and up-convert operands
fmadd	4	multiply-accumulate
reduce	6	extract result from vector register
Total	**50**	
Ratio	Compute/Total	**0.08**

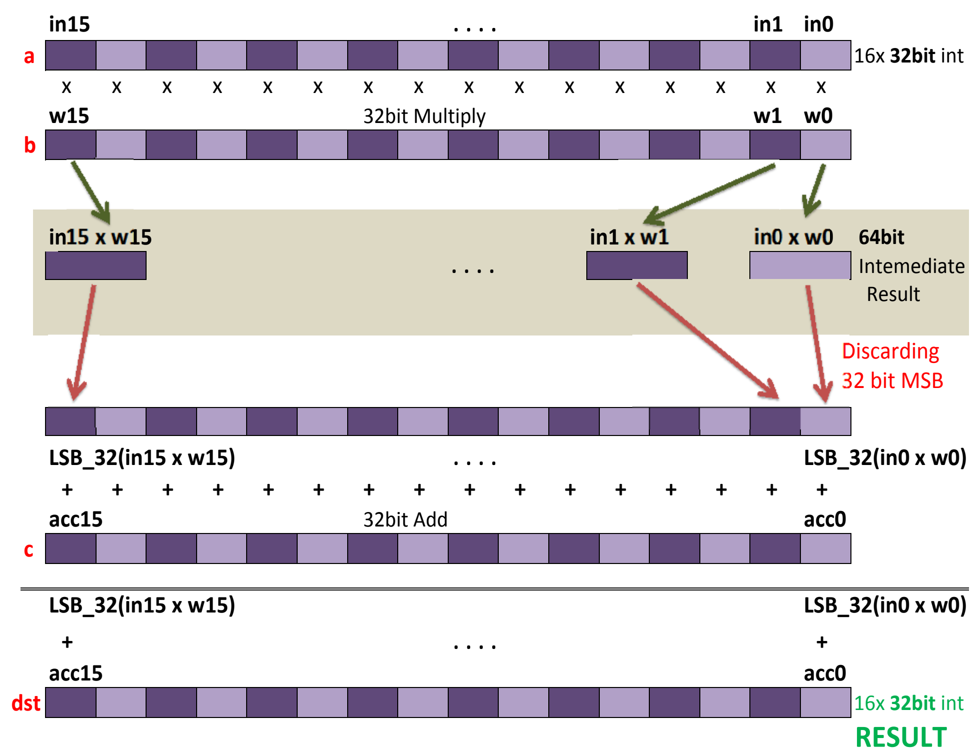


Figure 6.4: Details of `fmadd()` Multiply-Add operation

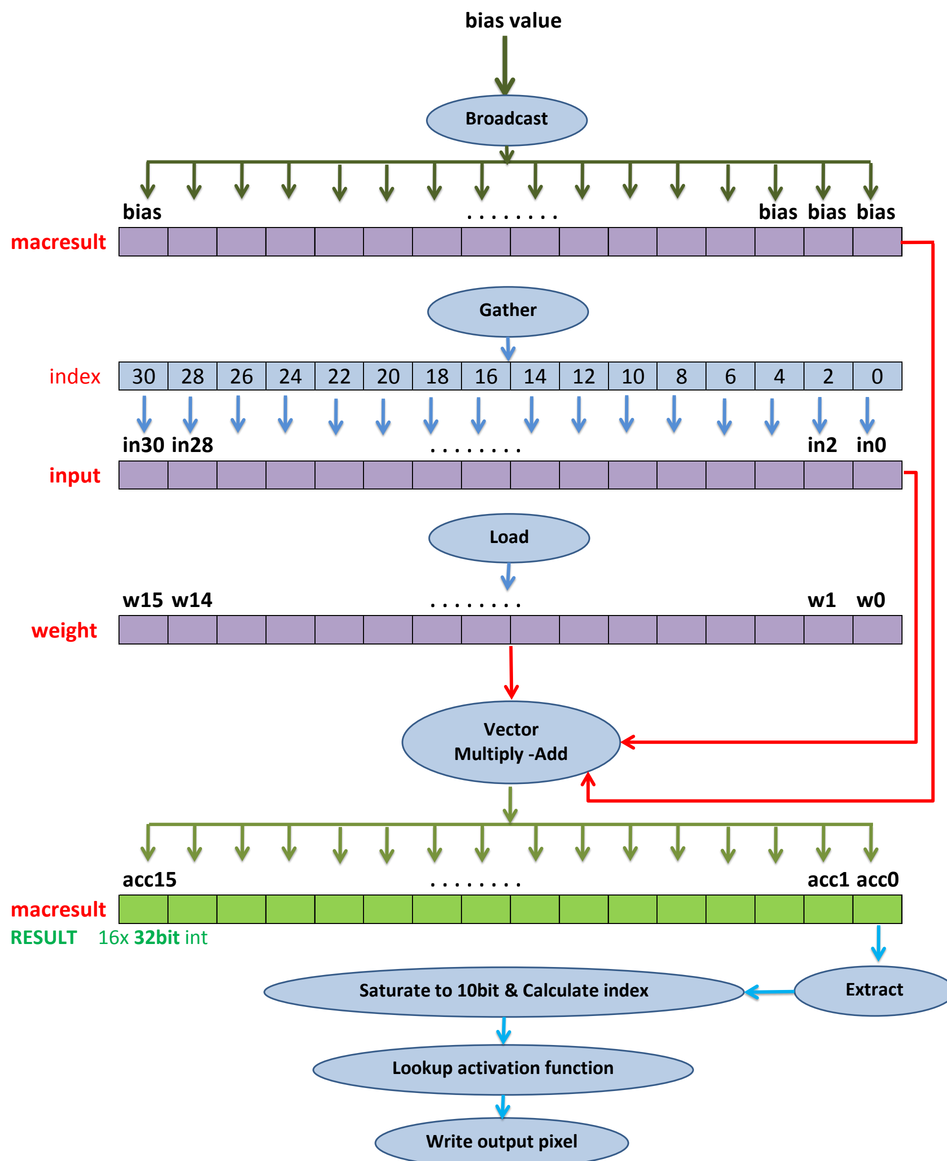


Figure 6.5: Layer 1 kernel block diagram

6.3 Results

The results of the code optimizations using vector intrinsics, loop unrolling and compiler optimizations are shown in this section.

Table 6.4 documents the actual speedup achieved in execution time after using vector SIMD intrinsics over the default non-vectorized code generated by the compiler.

The complete execution time of the speed sign application after optimization on a single core is 1.3197 seconds. This is same as saying that about 0.75 of a frame can be processed in one second. If we naively extrapolate this to 57 cores of a Xeon Phi, we can predict that the this ConvNet speed sign application can achieve a frame rate of 43 frames per second (fps) on the Xeon Phi 31S1P.

Table 6.3: ICC Execution times of the speed sign detection application

ICC	Execution time (ms)		
	Vectorization off	Auto-vectorized	Manually optimized
Layer 1	610.89	601.09	**107.97**
Layer 2	1,763.23	1,097.64	**173.26**
Layer 3	12,001.63	10,381.03	**970.94**
Layer 4	160.74	37.46	**67.57**
Complete	14,536.48	12,117.22	**1,319.73**

Table 6.4: Speedup achieved using the Intel C Compiler (ICC)

Layers	**Speedup** over Un-vectorized code	**Speedup** over Auto-vectorized code
Layer 1	5.7x	5.6x
Layer 2	10.2x	6.3x
Layer 3	12.4x	10.7x
Layer 4	1.0x	1.0x
Complete Application	11.01x	9.2x

6.4 Roofline model: Performance evaluation

6.4.1 Calculation of compute ceiling Xeon Phi

The equations 6.1 and 6.2, show the calculation for the number of Floating Point Operations (FLOP) that the Xeon Phi co-processor can compute in one second, using all of it's 57 cores or using only 1 core respectively. These are the absolute maximum theoretical numbers and are difficult to reach in practice.

Theoretical: 57 core

$$\texttt{57cores*1.1GHz*16(32bit SIMD lanes)*2ops(FMA) =2TFLOP/s} \quad (6.1)$$

Theoretical: 1 core

$$\texttt{1core*1.1GHz*16(32bit SIMD lanes)*2ops(FMA) =35.2GFLOP/s} \quad (6.2)$$

Practical: Scalar Roofline for a single core, calculated using the results from STREAM benchmark is `0.48 GFLOP/s`.

6.4.2 Calculation of memory ceiling of Xeon Phi

Here we calculate the available bandwidth between a processor and the on-chip Level 1 only accessible per core, the shared on Level 2 cache which is accessible to all 57 cores and bandwidth to the main DDR5 RAM on the Xeon Phi PCI card. These are the absolute maximum theoretical numbers and are difficult to reach in practice.

Bandwidth to Level 1 Data cache

Theoretical:

$$\texttt{64bytes/cycle/core @1.1GHz = 70.4GBytes/second/core} \quad (6.3)$$

Bandwidth to Level 2 cache

Theoretical:

$$\texttt{(1/2)*L1 BW[8] = 35.2GBytes/second/core} \quad (6.4)$$

Aggregate Bandwidth to DDR5 RAM

Theoretical:

$$\texttt{8 mem controllers * 2channels/controller * 32bits/channel * 5.0GT/s= 320GB/s} \quad (6.5)$$

This is the aggregate bandwidth of the platform, but as we are working on a single core, it may have access to only 1 out of the 8 memory controllers. This gives us a ballpark figure of 40 GB/s theoretical bandwidth per core.

Using the STREAM benchmark introduced in section 2.4, we find the actual bandwidth available to simple vector kernels like with multiply-add operation.

Practical: STREAM benchmark - 57 core

$$\texttt{Bandwidth = 145 GB/s} \text{ (best reported is } \texttt{179GB/s}[22]) \tag{6.6}$$

Practical: STREAM benchmark - 1 core

$$\texttt{Bandwidth = 5.8 GB/s} \tag{6.7}$$

Roofline figure

The Roofline model for a single core of the Xeon Phi co-processor is shown in figure 6.6.

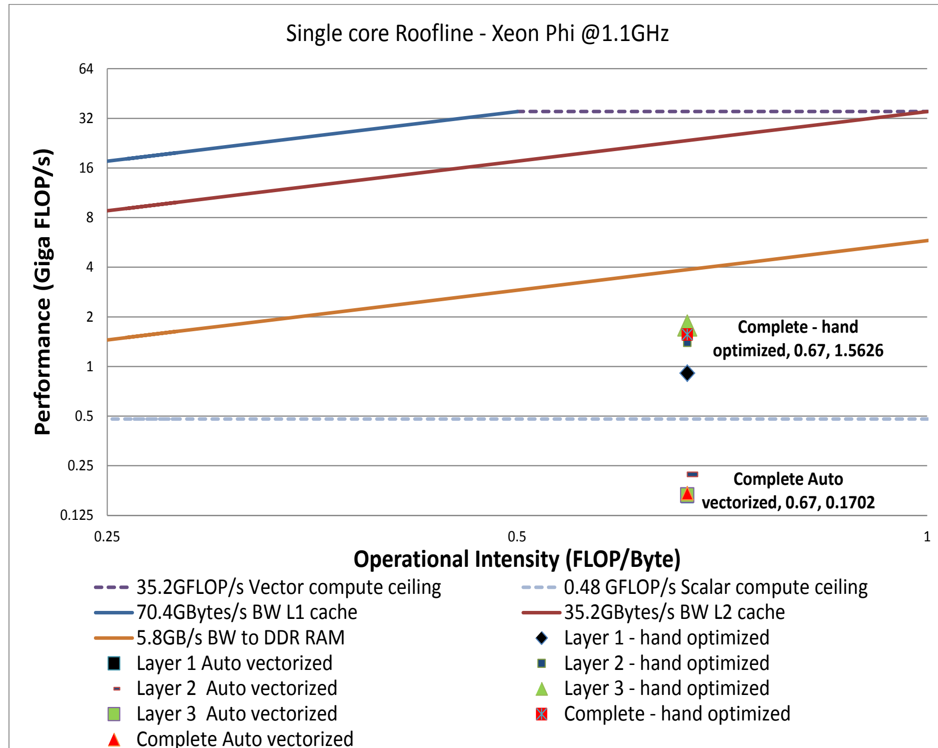


Figure 6.6: Xeon Phi single core Roofline: ConvNet application

6.5 Intel Core i7 (*Haswell*) v/s Xeon Phi (*MIC*)

Intel Core i7 has a very flexible instruction set for sub-word parallelism, whereas Xeon Phi has wider 512-bit instructions but more restricted. The number of data types supported is less, especially for pixel processing applications like image processing which require byte level operations. Also in terms of granularity of operation in the IMCI is only 32 bit, while the granularity is 8 bit in AVX instruction set. This impacts the performance by a factor of 2x slowdown for the speed sign application, as the datatype of operands is 16bit, hence the upper 16 bits of the available 32bit operand are unused.

The MIC has a simple core design versus the advanced out-of-order core in the Core i7 with Haswell architecture. This means that the sequential paths in the program will take longer to execute on a Xeon Phi as compared to a Core i7. Thus the speedup is limited by the serial part of the program in accordance with Amdahl's law. The SIMD vector unit in the Xeon Phi has a 4 clock latency and cannot issue the same instruction back to back in the same thread. The can achieve a max throughput of 1 cycle by utilizing all 4 threads to make a round-robin schedule. There are sequential sections in the kernel which limit the speedup which is achieved with parallelization. The `if, else if, else` statements for index calculation, the sigmoid activation function, and initialization of the bias variable `biasvar = bias[r]<<8`. This hurts the Xeon Phi even more because of a very inferior scalar unit.

One more thing to note is that the memory hierarchy on the two platforms are different. In case of multi-core performance the Intel Xeon Phi would have a larger bank of Level 2 caches (57x 512K bytes) compared to the six 256K bytes Level 2 caches of the core i7.

The Xeon Phi core lacks automatic L2 to L1 cache pre-fetching as found on the Haswell. Also L2 cache access times are 23 cycles on the Xeon Phi MIC architecture whereas they are just 11 cycles on the Core i7 Haswell [10, 7]. This put the Xeon phi core at a disadvantage compared to the Core i7.

One of the reasons for under-utilization of the available ILP, is that the compiler does not produce optimized assembly code. The compiler cannot auto-vectorize nested `for` loops. To alleviate this problem, all convolution operations have been implemented using Intel low-level intrinsics which increased ILP.

Hardware accelerators like Intels Xeon Phi, GPUs and FPGAs are best suited for embarrassingly parallel configurations. For eg., a Monte-Carlo pricing application, the best performance is achieved when we choose several million paths [33]. This shows the best performance gain for accelerators over CPUs. This might not be representative of the real-world scenario if your algorithm does not use this number of paths.

Table 6.5: Memory address `mt` alignment boundary [3]

	broadcast: 1x16	**broadcast: 4x16**	**broadcast: none**
conversion: uint8, sint8	1 byte	4 byte	16 byte
conversion: 16-bit types	2 byte	8 byte	32 byte
conversion: none	4 byte	16 byte	64 byte

6.6 Challenges faced on the Xeon Phi

First restriction in the MIC architecture is that it can only operate on 32bit data-types. Then a restriction of the `extload()` intrinsic, used to load data from memory, is that the starting memory address can only be certain alignment, as shown in the table 6.5. The two deciding factors are whether you would like to convert the loaded data into a data-type other than 32-bit `int`, and the other factor is whether you would like to broadcast the first 32-bits of the loaded data to all of the positions in the vector register or broadcast first four 32-bit values to four positions each. This leads us to use the more cycle expensive gather intrinsic to implement the loading of operands. A large portion of the time was spend debugging memory segmentation faults due to these memory restrictions.

The lack of backward compatibility of the IMCI instruction set of the Knights Corner Xeon Phi chip, with the older AVX2 instruction set on the Core i7 Haswell chip is another hurdle as the entire intrinsics code had to be rewritten. Having this backward compatibility in the new AVX-512 instruction set of the latest Knights Landing Xeon Phi processor is a boon, as it will enable early porting and testing of legacy code.

Having a co-processor instead of a CPU also has it's complications in terms of loading specific libraries and copying over relevant data for native execution on the Xeon Phi. Now the latest Knights Landing Xeon Phi processor offers a CPU form factor, which will be easier to natively optimize and would have access to a larger DDR RAM, not being restricted to the on board memory as in case of current co-processor.

Chapter 7

Conclusion and Future Work

With the widespread adoption of convolutional networks for various signal processing tasks, the computational demands have become a performance bottleneck, especially for general purpose embedded systems. Even though there have been some very efficient custom hardware accelerators, but the prohibitive complexity of programming and inflexible architectures have stalled market adoption. With the proliferation of multi-processor platforms in the market, it is essential for the research community to accelerate neural networks on highly-parallel resource constrained multi-processor hardware. Parallelization using SIMD vector intrinsics is one of the surest ways to move towards this goal.

7.1 Conclusions

The main contributions presented in this thesis are:

- Evaluation of a ConvNet speed sign detection application requirements
- Vectorized mappings on the Intel Core i7 and Xeon Phi
- Bottleneck analysis with evaluation

This thesis work presents highly efficient ways to utilize the subword SIMD vector capabilities of modern processors, by using the intimate knowledge of the neural net algorithm at hand. Using these techniques one can cut down the execution times by a factor of 12x, while increasing energy efficiency and resource utilization.

Based on the results, one can say that using SIMD vector intrinsics is an effective way to exploit the data-level parallelism inherent in convolutional network applications. This extra vectorization effort can mean the difference between real-time performance guarantees and non-real-time performance of an application. Also, in the area of dynamic power savings it has significant implications, as the shorter execution times leads to less power consumption.

This work has resulted in speedups ranging from 5.6x to 12.3x in different layers of the ConvNet speed sign application, using SIMD (Single Instruction Multiple Data) vector intrinsics, as compared to default compiler generated code.

The experiments conducted in this thesis serve as a precursor to judging the suitability of massively parallel many-core processors for running ConvNet applications in real-time and with increased power efficiency.

7.2 Future Work

Further work needs to be conducted in order to compare the performance of this ConvNet application on other architectures like Nvidia GPUs, ARM processors and custom hardware accelerators, like the one developed at the Eindhoven University of Technology for ConvNets [46].

7.2.1 Multi-core mapping using OpenMP

The next logical step is utilizing the power of all the 57 cores of the Intel Xeon Phi using the OpenMP library. Each of the 57 cores in the Xeon Phi supports 4 hardware threads. This makes it is possible to vary the number of threads from 1 Thread per core (57 Threads) to 4 Threads per core (228 Threads). Multi-threading helps increase the hardware utilization of the processor especially in case of lighter workloads, which do not occupy all functional units of the processor.

We can divide work among the cores using OpenMP directives such as "`#pragma omp for`" which is written at the beginning of for loops. This instructs the OpenMP library to parallelize this for loop among threads.

There are certain environment variable which can instruct the type of distribution of threads on the cores. Varying thread distribution on Cores using "`KMP_AFFINITY`" environment variable, can significantly boost the performance on the Intel Xeon Phi (refer fig.7.1). This has to be tested to fit the particular algorithm under consideration.

This type of optimization will be suited to shared memory systems, as this is the basis for software threading model. Nevertheless, useful hints can be derived about optimal task distribution of the particular neural network; which can be used to split the work on distributed memory multi-processor systems.

7.2.2 Auto vectorization with OpenMP `#pragma simd`

In the OpenMP 4.0 specifications, "`simd`" directives were introduced. The `simd` construct can be used to indicate that a loop is capable to be transformed into a SIMD loop. This means that multiple iterations of the loop can be executed concurrently using SIMD instructions.

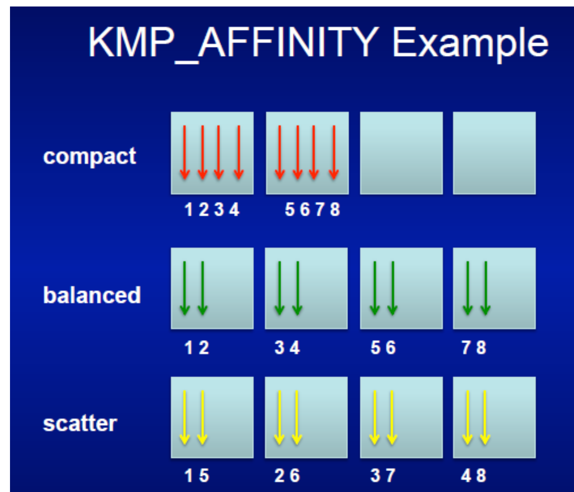


Figure 7.1: OpenMP thread distribution [12]

	Intel I7 4 cores	neuFlow Virtex4	neuFlow Virtex 6	nVidia GT335m	NeuFlow ASIC 45nm	nVidia GTX480*
Peak GOP/sec	40	40	160	182	320	1350
Actual GOP/sec	12	37	147	54	300	294
FPS	14	46	182	67	364	374
Power (W)	50	10	10	30	0.6	220
Embed? (GOP/s/W)	0.24	3.7	14.7	1.8	490	1.34

Figure 7.2: Example of desired comparison as performed by Yann LeCunn for the NeuFlow ASIC processor [29]

7.2.3 Performance comparison with SIMD accelerator and GPU

Comparing performance numbers of this ConvNet speed sign detection algorithm on the Intel Xeon Phi, a custom SIMD accelerator and a GPU implementation will give us much insight into the suitability of different architectures for running neural network based applications. Trade-offs can be made based on the requirements of power consumption, design time, flexibility and cost. Such comparisons can help put the performance numbers in perspective and help researchers design better suited architectures for neural networks.

A good example for such a comparison performed for neural networks performed by Yann LeCun is shown in figure 7.2. Here GOP/sec (Giga Operations/second) represents the of billions of operations that can be per-

formed on an architecture per second. FPS is the Frame rate Per Second which is achieved as a result of running a video processing application on this architecture. The last row GOP/s/W, shows the ratio of computation done to power consumed, which hints at the suitability of a platform to be used in a power constrained embedded device, like a cell phone.

We can see that the custom ASIC accelerator is the clear winner, with reconfigurable Virtex 6 FPGA accelerator coming second. This indicates that ASICs with a component of reconfigurability might be the best way ahead for fixed type of ConvNet algorithms.

Bibliography

[1] http://eyesofthings.eu/. [Online; accessed December 2015].

[2] Feature extraction using convolution. http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution. [Online; accessed 2015].

[3] User and Reference Guide for the Intel C++ Compiler 15.0. https://software.intel.com/en-us/node/523489. [Online; accessed 2015].

[4] Intel Xeon Phi Coprocessor family. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html, 2012.

[5] Intel Names the Technology to Revolutionize the Future of HPC - Intel Xeon Phi Product Family. http://goo.gl/5ku0gT, 2015.

[6] Intel Haswell-E (5960X, 5930K, 5820K). http://www.overclock.net/t/1510106/various-intel-haswell-e-5960x-5930k-5820k-reviews, August 2014.

[7] PRACE Summer School - Enabling Applications on Intel MIC based Parallel Architectures - Bologna, Italy). https://events.prace-ri.eu/event/181//, Jul 2013.

[8] *Intel Xeon Phi Coprocessor System Software Developers Guide*, March, 2014.

[9] Intel Xeon Phi Coprocessor - the Architecture. https://goo.gl/iyBWgx, November 12, 2012.

[10] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, September 2015.

[11] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yoshua Bengio. Theano: new features and speed improvements. *CoRR*, abs/1211.5590, 2012.

[12] Carlos Rosales (Texas Advanced Computing Center). Introduction to Intel Xeon Phi Coprocessors. https://goo.gl/WnKWc6. [Online; accessed 2015].

[13] Adam Coates and Andrew Y Ng. The importance of encoding versus training with sparse coding and vector quantization. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 921–928, 2011.

[14] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[15] Tim Cramer, Dirk Schmidl, Michael Klemm, and Dieter an Mey. OpenMP Programming on Intel R Xeon Phi TM Coprocessors: An Early Performance Comparison. 2012.

[16] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 886–893. IEEE, 2005.

[17] Tim Dettmers. Deep Learning in a Nutshell: Core Concepts. http://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-core-concepts/, November 3, 2015.

[18] LRZ Dr. Volker Weinberg. Introduction into Intel Xeon Phi Programming. https://www.lrz.de/services/compute/courses/x_lecturenotes/MIC_GPU_Workshop/micworkshop-micprogramming.pdf, April 28, 2015.

[19] Randall James Fisher. General-purpose SIMD within a register: Parallel processing on consumer microprocessors. 2003.

[20] Simon S Haykin. *Neural networks and learning machines*, volume 3. Prentice Hall, 2008.

[21] Raj Hazra. Intel Xeon Phi coprocessors accelerate the pace of discovery and innovation. http://blogs.intel.com/technology/2012/06/intel-xeon-phi-coprocessors-accelerate-discovery-and-innovation/, June 18, 2012.

[22] Karthik Raman (Intel). Optimizing Memory Bandwidth on Stream Triad. https://software.intel.com/en-us/articles/optimizing-memory-bandwidth-on-stream-triad. [Online; accessed 2015].

[23] Radek (Intel). Green500 List Names Intel Xeon Phi Coprocessor-based System "Beacon" the World's Most Energy-efficient Supercomputer. http://goo.gl/PBmx8A, Nov 14, 2012.

[24] Colfax International. Special Promotion for Developers: Intel Xeon Phi Coprocessor 31S1P. http://www.colfax-intl.com/nd/xeonphi/31s1p-promo.aspx, February 6, 2015.

[25] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[26] Lei Jin, Zhaokang Wang, Rong Gu, Chunfeng Yuan, and Yihua Huang. Training Large Scale Deep Neural Networks on the Intel Xeon Phi Many-Core Coprocessor. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 1622–1630. IEEE, 2014.

[27] KYN. https://en.wikipedia.org/wiki/File:CVoverview2.svg. [Online; accessed 2015].

[28] Y. LeCun, L. D. Jackel, B. Boser, J. S. Denker, H. P. Graf, I. Guyon, D. Henderson, R. E. Howard, and W. Hubbard. Handwritten digit recognition: Applications of neural net chips and automatic learning. *IEEE Communication*, pages 41–46, November 1989. invited paper.

[29] Yann LeCun. Convolutional Networks: Unleashing the Potential of Machine Learning for Robust Perception Systems. http://goo.gl/cCF20G, May 2014.

[30] Ruby B. Lee. Subword Parallelism with MAX-2. *IEEE Micro*, 16(4):51–59, August 1996.

[31] Allison Linn. Microsoft researchers win ImageNet computer vision challenge. http://blogs.microsoft.com/next/2015/12/10/microsoft-researchers-win-imagenet-computer-vision-challenge/, December 10, 2015.

[32] Junjie Liu, Haixia Wang, Dongsheng Wang, Yuan Gao, and Zuofeng Li. Parallelizing convolutional neural networks on intel many integrated core architecture. In *Architecture of Computing Systems ARCS 2015*, volume 9017 of *Lecture Notes in Computer Science*, pages 71–82. Springer International Publishing, 2015.

[33] Jrg Lotze. Benchmarker Beware! http://blog.xcelerit.com/benchmarker-beware/, October 2015.

[34] John D McCalpin. Memory bandwidth and machine balance in current high performance computers. 1995.

[35] Krystian Mikolajczyk and Cordelia Schmid. A performance evaluation of local descriptors. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 27(10):1615–1630, 2005.

[36] Intel Newsroom. Intel Delivers New Architecture for Discovery with Intel Xeon Phi Coprocessors. http://goo.gl/Dy6Oba, Nov 12, 2012.

[37] Jongsoo Park, Mikhail Smelyanskiy, Karthikeyan Vaidyanathan, Alexander Heinecke, Dhiraj D Kalamkar, Xing Liu, Md Mosotofa Ali Patwary, Yutong Lu, and Pradeep Dubey. Efficient shared-memory implementation of high-performance conjugate gradient benchmark and its application to unstructured matrices. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 945–955. IEEE Press, 2014.

[38] Maurice Peemen, Bart Mesman, and C Corporaal. Speed sign detection and recognition by convolutional neural networks. In *Proceedings of the 8th International Automotive Congress*, pages 162–170, 2011.

[39] Maurice Peemen, Bart Mesman, and Henk Corporaal. Efficiency optimization of trainable feature extractors for a consumer platform. In *Advanced Concepts for Intelligent Vision Systems*, pages 293–304. Springer, 2011.

[40] Maurice Peemen, Arnaud AA Setio, Bart Mesman, and Henk Corporaal. Memory-centric accelerator design for Convolutional Neural Networks. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pages 13–19. IEEE, 2013.

[41] M.C.J. Peemen. Mapping convolutional neural networks on a reconfigurable FPGA platform . http://repository.tue.nl/710846, 2010.

[42] W. Pramadi. Automatic Mapping of Convolutional Networks on the Neuro Vector Engine. Master's thesis, Eindhoven University of Technology, 2015.

[43] Prof. Dr. Riko afari, Asis. Dr. Andreja Rojko, University of Maribor. . http://www.ro.feri.uni-mb.si/predmeti/int_reg/Predavanja/Eng/2.Neural%20networks/_05.html, December 2006.

[44] Matt Scarpino. Crunching Numbers with AVX and AVX2. http://www.codeproject.com/Articles/874396/Crunching-Numbers-with-AVX-and-AVX, Feb 2015.

[45] Pierre Sermanet. Deep ConvNets: astounding baseline for vision. http://cs.nyu.edu/~sermanet/papers/Deep_ConvNets_for_Vision-Results.pdf, April 6, 2014.

[46] Runbin Shi, Zheng Xu, Zhihao Sun, Maurice Peemen, Ang Li, Henk Corporaal, and Di Wu. A locality aware convolutional neural networks accelerator. In *Digital System Design (DSD), 2015 Euromicro Conference on*, pages 591–598. IEEE, 2015.

[47] George Teodoro, Tahsin Kurc, Jun Kong, Lee Cooper, and Joel Saltz. Comparative Performance Analysis of Intel Xeon Phi, GPU, and CPU. *arXiv preprint arXiv:1311.0378*, 2013.

[48] Maurice Peemen (TU/e). Speed Sign Recognition by Convolutional Neural Networks. https://youtu.be/kkha3sPoU70, https://youtu.be/UrWAeukkLdk. [Online; accessed 2015].

[49] Noelia Vallez, Jose Luis, Oscar Deniz, Daniel Aguado-Araujo, Gloria Bueno, and Carlos Sanchez-Bueno. The Eyes of Things Project. In *Proceedings of the 9th International Conference on Distributed Smart Cameras*, ICDSC '15, pages 193–194, New York, NY, USA, 2015. ACM.

[50] R.P.M. van Doormaal. Parallel Training of Large Scale Neural Networks: Performance Analysis & Prediction. http://repository.tue.nl/741149, 2012.

[51] Andre Viebke and Sabri Pllana. The Potential of the Intel Xeon Phi for Supervised Deep Learning. *arXiv preprint arXiv:1506.09067*, 2015.

[52] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[53] Yoshua Bengio (Deep Learning Tutorials). Convolutional Neural Networks (LeNet). http://deeplearning.net/tutorial/lenet.html. [Online; accessed 2015].

Appendix A

Appendix: Core i7 platform details

The figure A.1 shows a simplified internal block structure of the Haswell micro-architecture in the Intel Core i7 processor. This figure gives us a better understanding of the capability and limitations of the processor to execute different type of instructions simultaneously. As instructions on different ports can execute at the same time, at the same clock cycle.

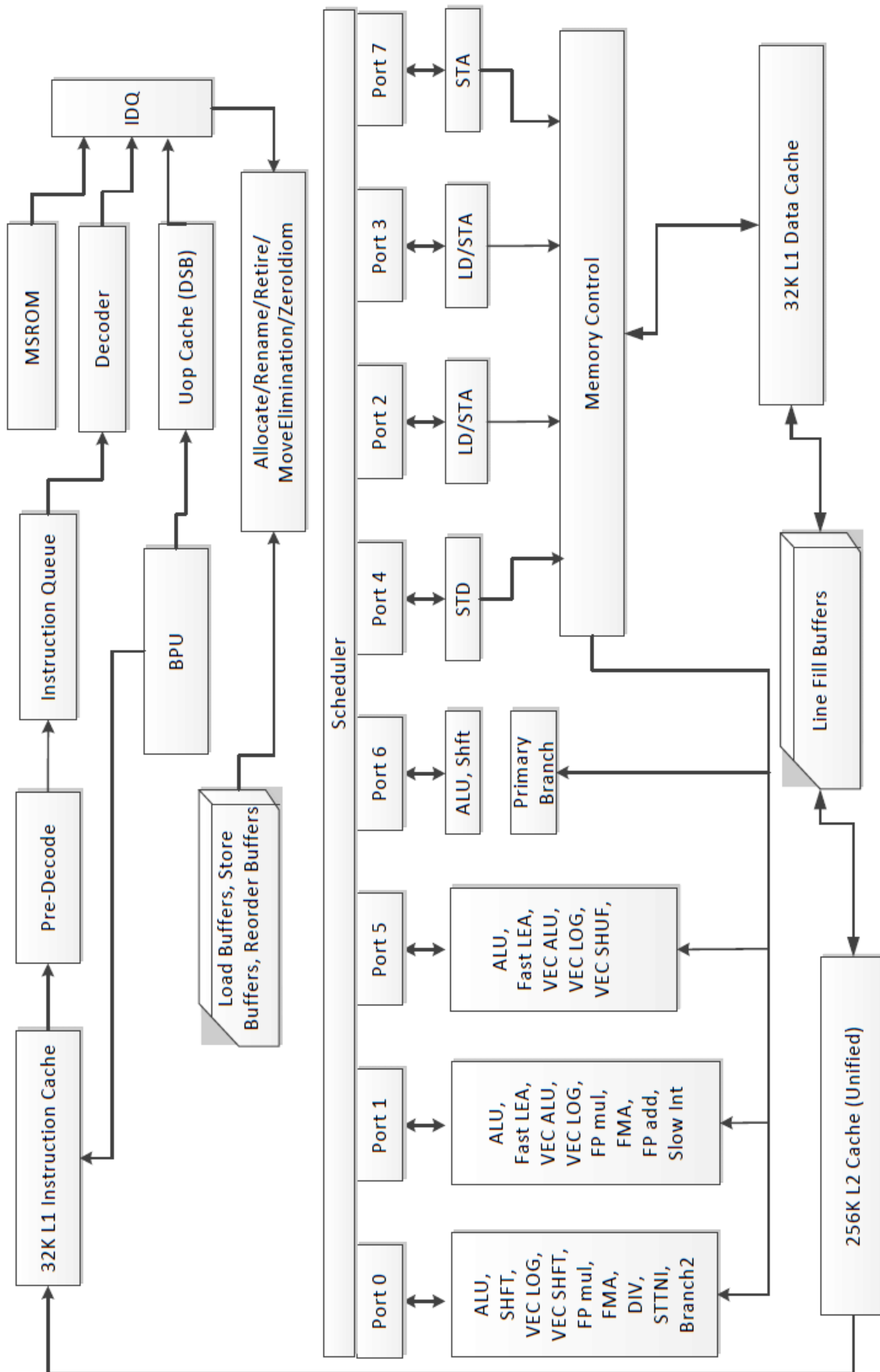


Figure A.1: CPU Core Pipeline Functionality of the Haswell Microarchitecture [10]