MASTER

Embedded platform selection based on the Roofline model applied to video content analysis

Spierings, M.G.M.; van de Voort, R.W.P.M.

Award date:
2012

Eindhoven University of Technology

Department of Electrical Engineering
Electronic Systems Group

Master's Thesis

Embedded platform selection based on the Roofline model

Applied to video content analysis

By
Martien Spierings & Rob van de Voort

Supervisors:
Henk Corporaal
Cedric Nugteren
Tom Goossens
Nick de Koning

Embedded platform selection based on the roofline model

Science Park Eindhoven, December 8, 2011

Course	: Graduation project
Master	: Embedded Systems
Department	: Electrical Engineering - Eindhoven University of Technology
Chair	: Electronic Systems - Eindhoven University of Technology

Committee : *Eindhoven University of Technology:*
Henk Corporaal
Pieter Jonker
Cedric Nugteren

Prodrive B.V.:
Tom Goossens

Authors : Rob van de Voort & Martien Spierings
Science Park Eindhoven 5501
5692 EM SON

Prodrive B.V.
P.O. Box 28030
Science Park 5501
5692 EM Son
The Netherlands
☎ +31 (0)40 2676 200
🖷 +31 (0)40 2676 201
🖱 http://www.prodrive.nl
✉ info@prodrive.nl

Eindhoven University of Technology
P.O. Box 513
Den Dolech 2
5612 AZ Eindhoven
The Netherlands
☎ +31 (0)40 2479 111
🖷 +31 (0)40 2441 692
🖱 http://www.tue.nl
✉ info@tue.nl

Abstract

An *Embedded platform* is an information processing system, designed to perform a dedicated function. These systems are embedded into many electronic products, e.g., watches, cars, ovens, and mobile phones. Embedded systems have become part of our daily life and their number, functionality and complexity increased over the years. For instance, in the past a phone only offered functions like calling and text messaging. Nowadays, these basic functions are extended with video calling, multimedia messaging, internet browsing and gaming. These functionalities of an embedded system are often controlled by software that executes on one or more processing units inside the embedded system.

Selecting hardware is an essential step in the development of an embedded system, especially selecting the appropriate processing units. It is a key step to the success of an embedded system that the processing units are able to meet the required performance. Performance estimations can be done using a cycle accurate simulator or by running benchmarks, however running a simulation is time consuming and running benchmarks may not be possible, this causes a major problem when a combination of processing units needs to be selected in the early phase of a design. Therefore, we present a method for the selection of processing units based on documentation and a high-level application description. This allows for a fast selection which decreases the time to market for embedded systems.

The method focusses on throughput oriented systems and extends the Roofline model to heterogeneous platforms, in order to give an upper bound for the performance of an application on a platform while providing an insightful visualization of the attainable performance. A platform is the composition of multiple processing units. This method enables the modelling of embedded applications on different processing units without requiring detailed knowledge of the application. It consist of decomposing applications into Generic Computational Blocks (GCBs) from which the number of operations and data transfers are extracted. This decomposition is then used to match the GCBs to the available hardware model and to select a set of processing units to form a computational platform for the application. The method is validated by using micro-benchmarks and an real-life application from the video content analysis domain. The theoretical estimates constructed with this method match the real-life performance of an processing unit for 95%. However the memory estimate is only accurate for 40% up to 80% due to the complex implementation of the memory controllers involved.

The method uses an extended version of the Roofline model. These extensions make the method suited for fast selection of processing units for an application. The guideline constructed from this method is usable by a designer to see different design trade-offs.

Contents

List of Figures iv

List of Tables vi

1 Introduction 1

1.1 Related work . . . 2

1.2 Motivation . . . 6

1.3 Thesis outline . . . 7

2 Method for selecting an embedded platform 9

2.1 The Roofline model and its shortcomings . . . 10

2.2 Contributions to the Roofline Model . . . 14

2.3 Roofline estimation errors . . . 17

2.4 Determining a Roofline model . . . 19

2.5 Application utilisation Roofline . . . 23

2.6 Verifying a Roofline Model . . . 26

3 CPU Roofline model 29

3.1 Background . . . 29

3.2 Determining the theoretical Roofline . . . 33

3.3 Determining the practical Roofline . . . 36

3.4 Intel Atom E6xx . . . 39

3.5 Intel Xeon E5540 . . . 43

3.6 Texas Instruments OMAP4430 46

3.7 Performance restrictions 49

3.8 Conclusions 51

4 DSP Roofline model 53

4.1 Background 53

4.2 Determining the Roofline for the Omap 137 55

4.3 Additional properties 57

4.4 Conclusion 58

5 GPU Roofline model 59

5.1 Background 59

5.2 Determining the theoretical Roofline 63

5.3 Determining the practical Roofline 63

5.4 Performance restrictions 67

5.5 Conclusion 70

6 FPGA Roofline model 71

6.1 FPGA basics 71

6.2 A Roofline for the FPGA 72

6.3 The FPGA computational Roofline 76

6.4 Determining the practical Roofline 80

6.5 Performance restrictions 81

6.6 Conclusions 81

7 The decomposition of VCA applications 83

7.1 VCA applications 84

7.2 Describing Generic Computational Blocks 84

7.3 Operations per byte of a GCB 87

7.4 VCA use case 89

7.5 Multiple GCBs on one processing unit 98
7.6 Pipelining GCBs on one processing unit 100
7.7 Platform 102

8 From application to platform, a guideline 107

8.1 Guideline prerequisites 107
8.2 Processing unit selection guideline 108
8.3 Example usage of the guideline 110

9 Conclusions 117

9.1 Future work 119

List of Figures

1.1 Design space exploration 3

1.2 Processor design space, image from [7] 6

2.1 Research Approach 9

2.2 Model assumed by the Roofline model 11

2.3 Original Roofline model 12

2.4 Differences between the original and the extended Roofline model 13

2.5 Different types of operations result in points in the Roofline model 15

2.6 Extra data sources in the Roofline model 17

2.7 Extensions to the original Roofline 18

2.8 Simplified processing unit model 20

2.9 Utilisation Roofline for an application and processing unit 26

2.10 Micro-benchmarking concept 27

3.1 Basic components of a stored instruction digital computer 30

3.2 Basic CPU concept withArithmetic Logic Unit (ALU), memory and CU) 31

3.3 Modern CPUs contain multiple ALUs to process operands in parallel 31

3.4 Intel Atom tunnel-creek micro-architecture instruction ports 40

3.5 Intel E630 benchmark results for integer, single precision float and SIMD 42

3.6 Intel E630 benchmark results for intrinsics, assembly and cache 43

3.7 Intel Nehalem micro-architecture instruction ports 44

3.8 Intel Xeon 5540 benchmark results 45
3.9 Intel Xeon 5540 benchmark results for intrinsics, assembly and cache measurements 46
3.10 Cortex-A9 instruction ports 47
3.11 Texas Instruments OMAP4430 micro-benchmark results 48
3.12 X86 ISA registers 50
3.13 Register of the ARM Instruction Set Architecture (ISA) and extensions to it such as NEON and VFPv3 51

4.1 Early DSP architectures 54
4.2 TI DSP C674x architecture 56
4.3 Measured roof line of DSP 57
4.4 Software pipelining 58

5.1 NVIDIA Architectures 60
5.2 AMD VLIW5 architecture 61
5.3 Cuda and OpenCL programming model 62
5.4 NVIDIA FX1700 64
5.5 NVIDIA FX1700 Ceilings 65
5.6 NVIDIA ION and GTX 460 66
5.7 AMD HD6870 67
5.8 GPU memory access 69
5.9 Performance restrictions on GPUs 70

6.1 the main components 72
6.2 Extension to the original Roofline model 73
6.3 Figure of different areas defined in the FPGA in order to estimate the Roofline 74
6.4 Distance between two logic functions 74
6.5 DSP48E1 basic functionality 77
6.6 Resulting Roofline after estimation of computational performance and memory bandwidth 79

7.1 Frame differing 86
7.2 Use case task graph 90
7.3 Mixture of Gaussians example 90
7.4 Erosion filter 92
7.5 Connected component labelling 94
7.6 Basic blob tracker 95
7.7 Region extrapolation 96
7.8 GCB overview 97
7.9 Combining GCBs in the Roofline model 99
7.10 Example of pipelining GCBs on one processor 101
7.11 platform 102
7.12 Example of a CPU-GPU platform 103
7.13 The Utilisation Roofline for the CPU-GPU platform 105
7.14 The platform modelled on processing unit level 106

8.1 Research Approach 108
8.2 Pareto space of BOM vs Risk of the configurations 116
8.3 Pareto space of Risk, Power and BOM of the configurations 116

List of Tables

1.1 The partitioning of chapters 7

2.1 Example of operations per byte measure 14

2.2 parameters for calculating the utilisation Roofline 25

3.1 Roof and Ceilings of the Intel Atom E630 41

3.2 Roofs and ceilings of the Intel Xeon E5540 44

3.3 Roofs and ceilings of the Cortex-A9 part of the OMAP4430 47

4.1 Properties of the C674x 56

5.1 Properties of the NVIDIA Quadro FX1700 64

5.2 Properties of the NVIDIA ION and GTX460 65

6.1 Xilinx Virtex-6 XC6VLX240T available and required amounts of logic 75

6.2 Number of LUTs available for computational purposes 78

6.3 Number of FFs available for computational purposes 78

6.4 FPGA resource usage 78

7.1 GCB analysis approaches 89

7.2 Quantisation of the GCB variables and parameters 90

7.3 GCBs A and B combined on one processing unit 98

7.4 Example application for platform 104

8.1 List of processing units that are available for selection in this example 111

8.2 Generic Computational Block (GCB) properties used for the heuristic 112

8.3 Processing unit capabilities 113

8.4 List of processing units selection result 113

8.5 GCB requirement of bandwidth Gbyte/sec and performance Gops/sec 113

8.6 Risk of running a combination of GCBs ($comb$) on processing unit A or D 115

8.7 Different configurations of processing units and GCBs 115

Chapter 1

Introduction

An *Embedded platform* is an information processing system, designed to perform a dedicated function. These systems are embedded into many electronic products, e.g., watches, cars, ovens, and mobile phones. Embedded systems have become part of our daily life and their number, functionality and complexity increased over the years. For instance, in the past a phone only offered functions like calling and text messaging. Nowadays, these basic functions are extended with video calling, multimedia messaging, internet browsing and gaming. These functionalities of an embedded system are often controlled by software that executes on one or more processing units inside the embedded system.

Electronics designers design embedded systems according to the requirements specified by their customers. Such requirements specify criteria regarding the embedded system, e.g., power consumption, size, cost, response time, processing speed. In some cases, determining whether the embedded system meets the requirements is straightforward. For example, if the price is above the budget then the cost requirement is not met. In other cases it may be more difficult to determine if the requirements are met, especially for performance requirements like processing speed that depend on the combination of processing unit and software performance.

Already in the specification phase an embedded system designer wants to be confident that the system can meet the performance requirements. This performance mainly depends on the combination of processing units and software, because these process the information going through the system. Verifying that performance requirements are met can be done by testing all software tasks on every processing unit and selecting the combination of processing units and software that meets the requirements. However, this requires too much time, because a vast number of processing units is available and because software development takes a lot of time. In practice, an embedded system has a time to market constraint and because of this design time is limited. Moreover, the availability of processing units and software is an issue. In the case of software it is often not implemented or confidential and cannot be accessed in the specification phase. Furthermore, processing units might only exist on paper. Despite these limitations one or more processing units must be selected in limited time to form an embedded system for a given algorithm, task or application.

This raises the question, how can an embedded system designer quickly estimate which processing units are needed for a set of tasks and system requirements? In the next section related studies are addressed that can help to answer this question.

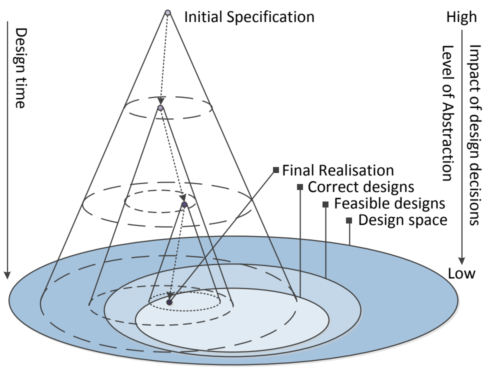
1.1 Related work

Selecting processing units based on system requirements can be classified as part of the Design Space Exploration (DSE) problem of an embedded system. DSE is the process of discovering and evaluating design alternatives during system development. A common view on DSE of embedded systems is to distinguish a number of consecutive design phases. These design phases focus on refining the amount of detail in order to realise the required functionality. Figure 1.1a illustrates how decisions taken at consecutive levels may lead from the system requirements to a realisation. First, an overview of approaches to traverse a design space is discussed, followed by methods to evaluate a design point (one configuration within the design space) and a discussion of processing unit classifications.

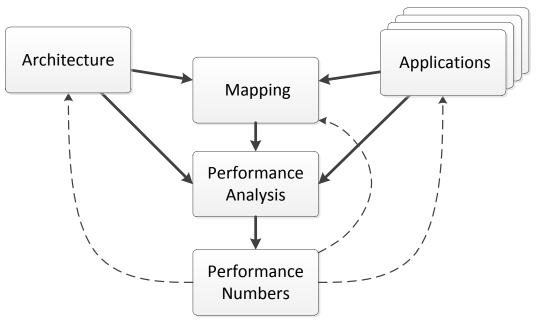
1.1.1 Design Space Exploration

The challenge in DSE originates from the vast design space that must be explored. Typically, an embedded system has millions, if not billions, of design possibilities [14]. The exploration of these possibilities is often based on the Y-chart [23] which is shown in figure 1.1b. The Y-chart is an iterative exploration of the design space, where one or several descriptions of the application and one architecture specification are kept separately. A mapping step binds the application to the architecture. This is followed by the evaluation of the mapping, e.g., in terms of performance, area, power, or other requirements. This evaluation may require synthesis of the architecture, adapting application code, or compilation of the application onto the architecture. The evaluation can trigger further iterations of the Y-chart by adapting the architecture, application, or the mapping strategy itself. Givargis [13] proposes a Y-chart based method to automate the exploration. The design space is explored to find all Pareto-optimal configurations of parametrized System-On-a-Chip (SoC) architectures. Pareto-optimal configurations are configurations that are not dominated by any other configurations from the space. This is elaborated by Geilen et al.[12] who propose a Pareto Calculator, a tool for compositional computation of Pareto points. The calculator is based on the algebra of Pareto points and used to construct a Pareto optimal system from different Pareto sets.

The previous discussed work depicts how the design space can be explored. Another important aspect of DSE is the evaluation of design points. Design points can be evaluated on many metrics as suggested in the example of the Y-chart. Performance is considered to be one of the important metrics for an embedded system. Gries [14] defines three categories of methods to determine the



(a) Abstraction pyramid: successive exploration at increasing levels of detail, image redrawn from [35]



(b) The Y-chart [23] approach where one or several descriptions of the application and architecture specification are kept separately. An explicit mapping step binds application tasks to architecture building blocks. Followed by the evaluation of the mapping which may trigger further iterations. Image redrawn from [23]

Figure 1.1: Design space exploration

performance of a point in the design space, i.e., benchmarking, simulation and analytical modelling. These categories will be discussed in the following paragraphs.

1.1.2 Benchmarking

Benchmarking is defined as ”measuring a product according to a standard in order to compare it with other products”. In many performance analysis cases a software application is the benchmark used to compare processing units. The application is executed on processing units to obtain the metrics of interest. These metrics can be compared to find the processing unit that is compliant with the requirements. Examples of benchmarks that can be used to determine a measure of performance are, Whetstone(WMIPS), Linpack(Mflops), Drystone(DMIPS) and CPU2006. Whetstone uses a mix of integer and floating point operations that operate on shared variables, which is not a good representation for software applications according to Weicker [39]. Linpack operates on large matrices using only floating point operations. Linpack contains a number of variants, i.e., single or double precision numbers, rolled or unrolled loops and different matrix sizes. It is therefore important to know which version is used when comparing Linpack results. Drystone is a benchmark that is based on system-type programs this contains, less numeric operations, fewer loops, more branch statements and procedure calls. A more recent benchmark is the CPU2006 benchmark of Standard Performance Evaluation Corporation (SPEC), which includes a set of 29 benchmarks to test the floating point and integer performance of a CPU. However, these benchmarks are specially designed for the CPU, and in the ideal case a benchmark is usable for every type of processing unit. The work of Asano et al.[3] uses defined algorithms instead of application to target different processing unit types, i.e., CPU, GPU, FPGA. This benchmark should show which processing unit is most suited for the parallelism in image processing. The FPGA achieves the best performance in two out of the three cases, however the implementations on GPU and CPU are not fully tuned. This gives a biased view of the suitedness of each processing unit for the algorithms. When comparing the benchmarks results one should keep in mind under what conditions the benchmark is performed. The used compiler, libraries, programming language and cache size can have a big influence on the results[39]. Benchmarking can be a useful approach to determine the performance of a processing unit. However, executing the benchmarks on every candidate processing unit can be a time consuming process, and benchmarks can only be performed when the processing units are at hand.

1.1.3 Simulation

Simulation is defined as the imitative representation of the functioning of one system or process by means of the functioning of another. A processing unit simulator processes the model or source code of the application on a executable model of a processing unit. The Texas Instruments (TI) C64x+ simulator [9] is an example of a simulator which uses source code. This simulator is not applicable in general because the performance results only apply to C64x+ based processing units. Bammi et al. [4] propose a performance estimation based on the simulation of a virtual instruction set. Source code is compiled to virtual instructions that are architecture independent. These instructions are executed on processing unit models which issue a certain delay for every instruction.The in the work of Caarls [6] the simulation time is shortened by benchmarking each operation in isolation, and combine them in a high-level multiprocessor simulation. In this work different networks are taken into account for simulating the multiprocessor system. The obtained results are used to create a Pareto volume of area, energy, processing time, which shows optimal trade-offs. However, these methods are often unusable in the specification phase of a design because the source code of the application must be available. This is not the case in the work of Florescu et al.[11], which defines a formal modelling language, called Parallel Object-Oriented Specification Language (POOSL), which is used to model the performance of soft real-time systems. This approach is also able to detect deadline misses as well as deadlocks. The simulation is based on the instruction count of scenarios and can handle jitter in the activation of scenarios as variation of the load of a scenario. The disadvantage of this approach is that the execution time of the scenarios on the simulated architectures must be known. A disadvantage of simulation in general is the time required to create and execute a model, in particular cases where the simulation has to reach a steady state.

1.1.4 Analytical models

An alternative to benchmarking and simulation is the use of analytical methods to generate performance numbers of a processing unit. These methods provide a relatively high abstraction of the system but are also less accurate compared to benchmarking and simulation according to Gries et al.[14]. However, analytical methods are multiple orders faster than a simulation approach according to Eeckhout et al.[10]. Therefore, analytical models can offer a fast evaluation of design points and ease early design decisions. The work of Lattuada et al.[22] proposes an analytical method for performance estimation without requiring knowledge of the platform architecture. The information needed for the estimation is obtained by building a linear regression model from instruction sequences for which the execution time is known. These execution times are obtained by running a set of applications. This method is not usable because the processing units must be available to obtain the execution times. The work of Shabbir et al. [29] uses architecture aware Synchronous Data Flow (SDF) to analyse the performance of applications on a communication assist based multi processor system, given that the mappings of the tasks on the processors is already provided. The SDF graph is used as an input for iterative probabilistic performance prediction which uses profiled application code to obtain the execution times and calculates the throughput for soft real-time systems. The automated design flow proposed in this work is also able to generate the required soft- and hardware for the required functionality. This can be very useful for FPGA designs, however, it cannot be applied to all types of processing units. Moreover, to apply this method, an implementation of the application is required.

Another method that looks promising for performance estimation is the Roofline model proposed by Williams et al.[40]. The model aims to provide visual insights in the performance of parallel architectures for floating point computations. The model ties together floating-point performance, memory performance and application performance in a single graph. The peak performance can be found using the hardware specification, and it looks like this implies that a model can be created from documents and data sheets of the processing unit. Although this method has benefits it also features some shortcomings: **1)** The model assumes floating-point operations while applications might consist of a mix of different data types. **2)** The memory performance is modelled by the performance of the external memory, while a processing unit might use different data sources like an interconnect and internal memory. **3)** The roof, which is the highest attainable performance of the processing unit, is not always a relevant upper bound for an application. **4)** The model is based on issued instructions, therefore it is unknown whether it can be used to model a Field Programmable Gate Array (FPGA).

1.1.5 Processing units classification

Processing units exist in various forms, contributing to a large design space. Despite the differences there are also common properties between processing units. These common properties are used to classify processing units. A well known classification is Flynn's taxonomy described in [15]. Flynn defines four classes which are based upon the number of concurrent instruction and data streams available in the architecture, i.e., Single Instruction Single Data (SISD), Multiple Instruction Multiple Data (MIMD), Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD).

A more fine-grain classification is prosed by Corporaal [7]. In this classification each processing unit can be specified as a 4-tuple *(I,O,D,S)* where *I* is the number of instructions per cycle, *O* the number of operations per instruction, *D* the number of operands or operand pairs to which the operation is applied, and *S* is the superpipelining degree. This results in a four dimensional processor design space, as illustrated in Figure 1.2. In this work four classes are defined, i.e., Central Processing Unit (CPU) also called a general purpose processor, Digital Signal Processor (DSP), Graphical Processing Unit (GPU), Field Programmable Gate Array (FPGA) which are based on commercially used names.

CPUs can be classified as SISD or MIMD and often has some SIMD parts. CPUs are very diverse and exist in variations in each dimension of *(I,O,D,S)*

DSPs can be classified as SISD or MIMD and often has someSIMD parts. The DSP can be very similar to the CPU and is often characterized by a large *O*.

GPUs can be classified as SIMD and can be characterized by a large *D* and *S*.

FPGAs cannot be classified using the previous methods because it is not based on the execution of instructions in a fixed data path.

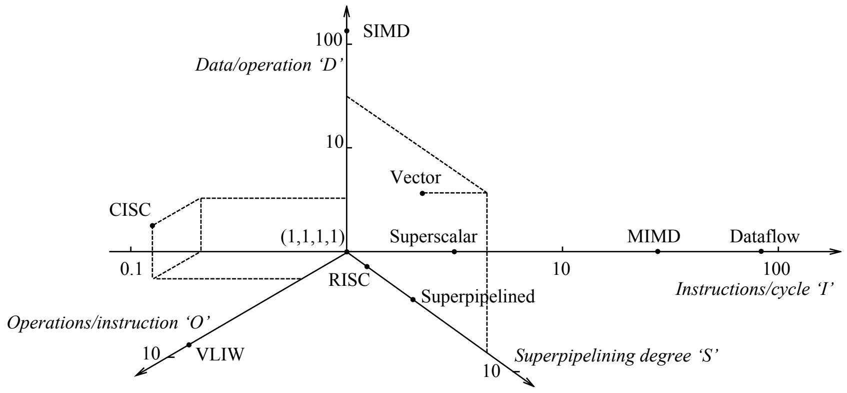


Figure 1.2: Processor design space, image from [7]

1.2 Motivation

The related work shows that there are many methods to traverse the design space of an embedded system. However, no comprehensive answer for quick processing unit selection for an application can be found in prior work. Prior work assumes the design points in the design space are evaluated according to requirements of the embedded system. A major requirement of such an embedded system is its performance, which can be evaluated by simulation, benchmarking or using an analytical method. However, most of these methods assume the availability of application source-code and processing hardware or require a considerable amount of time and effort.

From all of these methods the Roofline model by Williams et al.[40] is the most promising: it can model an application and a processing unit in a single insightful model. It can model application peak performance based on a small set of hardware specifications. The shortcomings of the model which are discussed in the related work need to be addressed in this thesis to make it applicable for a processing platform selection.

Therefore, this thesis extends the Roofline model and integrates it into a larger method that is able to select processing units for a processing platform. This method must be able to analyse the performance of an application and processing units to be able to quickly select a set of suited processing units for this application. Moreover, this selection should be based on processing unit documentation and description of the algorithms inside the application. Additionally, this method should be able to give an upper bound on the required and available performance inside the application, processing units and processing. Finally, the method must be suited for different kinds of processing units i.e. CPU, DSP, GPU, and FPGA.

The process of going from documentation and algorithm description to an insightful performance estimation that matches real-life performance must be verified in this thesis.

1.3 Thesis outline

This thesis is organized as follows. Chapter 2 gives an overview of the proposed platform selection method and presents contributions to solve the short comings in the Roofline model. Chapter 3 to 6 discusses the different types of processing units respectively CPU, DSP, GPU, and FPGA. These chapters aim to verify the theoretical performance estimated from the documentation against the practical reachable performance for the different processing units. In this way these chapters verify the usability of the Roofline model as a performance estimation method. After introducing the model and processing units, properties of applications, algorithms and GCBs are examined in chapter 7. This chapter addresses the software side of the method by a detailed reprise of the fist steps of the approach. Chapter 7 also presents a use case where an application is modelled. After depicting the properties of an application the complete method is applied to another use case in which example processing units are selected for the application from chapter 8. Chapter 9 concludes this thesis and suggests some possibilities for future research concerning this work.

Name	Chapters written
Rob van de Voort	2,3,6,8
Martien Spierings	1,4,5,7,9

Table 1.1: The partitioning of chapters

Chapter 2

Method for selecting an embedded platform

Defining a platform, a combination of hardware Processing Unit (PU) and peripherals, that meets the requirements and performance of the application is not trivial. Processing units exist in different sizes, types, and brands. However not every processing unit is suited for the requirements of the application.

In this chapter a method is proposed to extract information from an application and processing units and estimate which part of the application can be executed on which processing unit. This information can then be used to select which processing units can be used on the platform. Figure 2.1 depicts the proposed method. The key part of this method is step number three in which a pre-selection is done with a heuristic and the final selection is based on the Roofline model. How a Roofline model can be created from documentation of the processing unit is explained in section 2.1 and onwards.

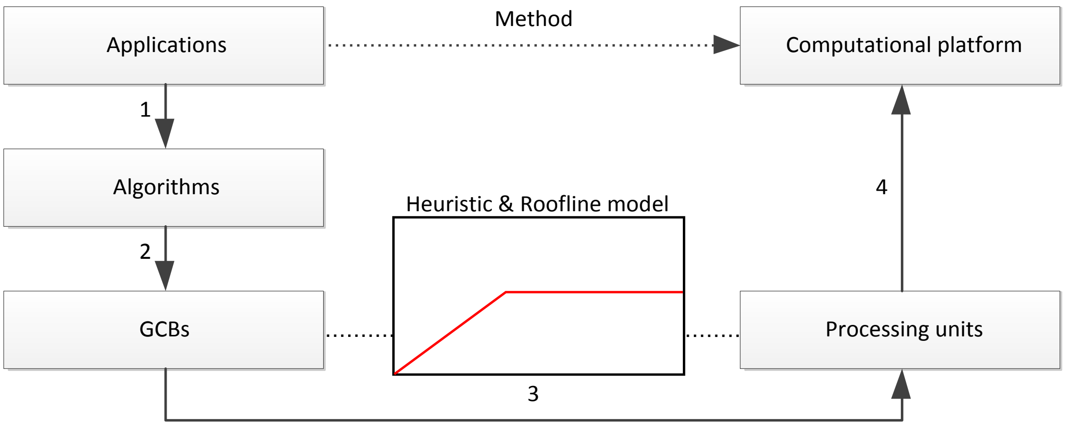


Figure 2.1: The proposed method: 1) Analyse the application. 2) Extract the GCBs. 3) Match GCB requirements with processing unit models, and 4) Define the required computational platform

Selecting one or more PUs for the platform requires information about the application. In step one of the method (figure 2.1) the application is analysed and the algorithms that are used within the application are identified. Algorithms define the kind of processing that is done inside an application, e.g., a JPEG image decoder consists of a number of algorithms: run-length decoder, quantization decoder, inverse discrete cosine transformation .

Most Algorithms contain one or more smaller functions to produce their output. These smaller functions are defined as GCBs. GCBs are basic computation blocks which could be used in more than one algorithm. For example, Huffman decoding is a GCB, which is used in MP3 and PKZIP algorithms. Identifying and analysing the GCBs is performed in step 2 of figure 2.1.

Step three of figure 2.1 depicts the process of selecting processing units for GCBs. A GCB is matched with one or more processing units based on the properties of the GCB and properties of the processing unit. First an heuristic is applied that performs an pre-selection based on the requirements of the GCB and requirements of the embedded platform. Afterwards a modified version of the Roofline model is used to model the performance of a GCB in conjunction with the available performance in a processing unit. Based on this model one or more processing units are selected that are suited for the given GCB. Chapter 8 contains an in depth explanation of this process. While the next sections show how the Roofline model can be constructed from knowledge of the internal architecture of a processing unit.

The fourth step is used to combine the selected processing units into a platform. The last part of chapter 7 depicts how the performance of such a platform can be estimated for a GCB.

2.1 The Roofline model and its shortcomings

The Roofline model[40] can model the memory and computational performance of a processing unit in combination with an application. It is an analytical model that can be visualised with a graph. The model illustrates the computational performance of a processing unit in terms of floating point operations per second versus the computational intensity (number of floating point operations per byte retrieved from the memory). The model assumes that the bandwidth of the external (off chip) memory or the average peak performance of a processing unit is the limiting factor on the performance of an application. According to the Williams et al [40], data needed to construct this model can be extracted from the hardware specification of a processing unit.

The Roofline model describes two types of performance boundaries, a " *roof* " which is the physical performance limit of a processing unit and " *ceilings* " which are boundaries that can be crossed if a program or application is programmed efficiently by making better use of available resources. A Roofline model of a processing unit has one roof, while it can have multiple ceilings. The model is visualized with a graph, which depicts the memory and computational roofs and ceilings of a processing unit (figure 2.3). Two types of roofs and ceilings are present in the Roofline model. These are the memory bandwidth roof and ceilings and the computational roof and ceilings. The

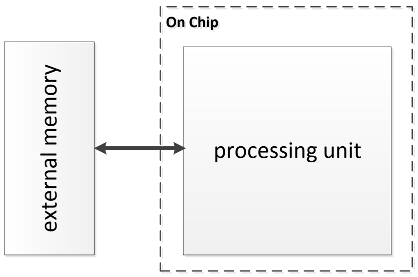


Figure 2.2: The Roofline model assumes the external memory and average peak performance are the limiting factors on performance of an application. This assumption is used to calculate an upper bound on performance. This bound is called the Roofline of a processing unit. In this assumption external memory is off chip memory which in many cases is DRAM

computational part of the Roofline model is bounded by the internal processing performance of the processing unit while the memory bandwidth performance is bounded by the available bandwidth to and from the external memory.

Applications are depicted as vertical bars in the Roofline model (figure 2.3), giving an indication of the maximum attainable performance of an application. The application is situated somewhere on the line (one point) depending on how well the resources of the processing unit are used. Whether an application used all the resources can depend on a number of things e.g. the time budget of the programmer or the properties of the application. Figure 2.3 depicts an example of the original Roofline with two applications

However, the Roofline model cannot be used to do proper performance estimations and comparisons because of a few shortcomings.
1)Only a single data type: The Roofline model considers floating point operations on data, whereas many algorithms use a mix of data types and sizes.
2)Only a single memory bound: The Roofline model considers the external memory as the only data source. This results in a strange (unwanted) side effect when using other data sources in an application. For example applications move along the flops/byte axis when data is fetched from a different location e.g. local memory. Figure 2.4a illustrates this side effect. This movement of an application within the Roofline model is not intuitive for a designer, and makes it hard to compare the performance of processing units and applications. Furthermore a designer did not change the complexity of the application but only improved the performance, still the application moves along the flops/byte axis.
3)Application only bounded by roof: The Roofline model assumes that the upper bound on performance is the roof. Better programming lets the application reach the roof of the model. However ceilings can also bound the performance of an application and introduce a lower upper bound on performance. For example the roof of a processing unit is positioned at 1000 operations per second for addition operations. A ceiling of the same processing unit is multiply operations at 100 operations per second. If an application only performs multiply operations it cannot perform better then the 100 operations per second ceiling, while the model depicts the roof is 1000 operations per second. The

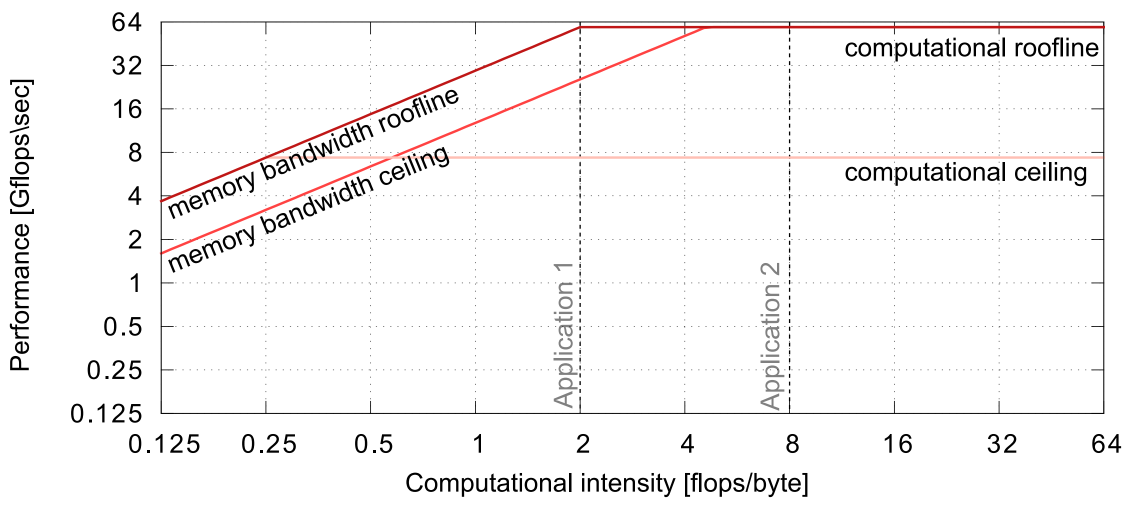
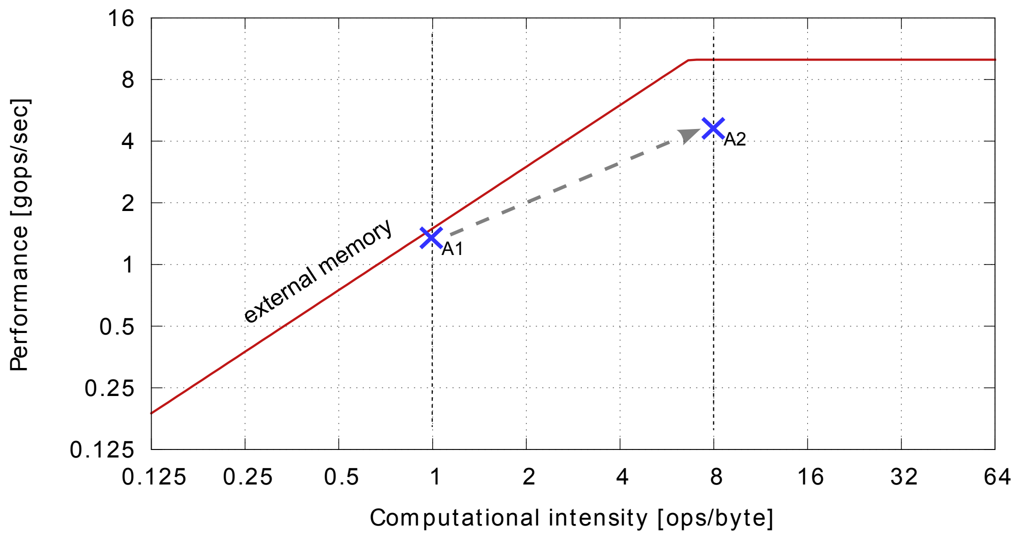


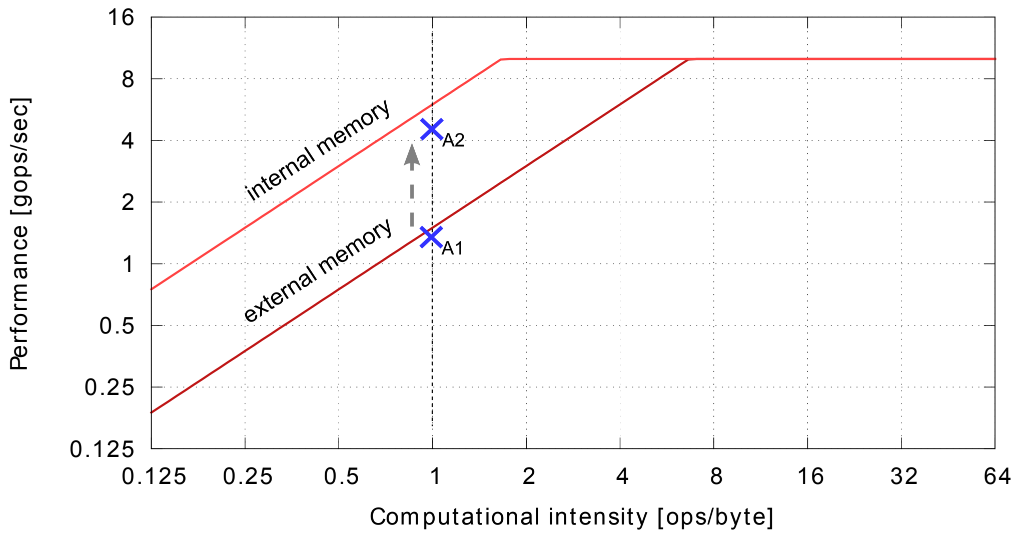
Figure 2.3: This Roofline model consist of two roofs a memory bandwidth roof limited by the external memory, and a computational roof limited by the computations of the processor. A memory ceiling is shown which can be "broken" by more effective accesses the external memory, for example by using software pre-fetching. The computational ceiling can be "broken" by using the computational resources more effective, for example by using vector instructions that are able to process more data in one cycle. Applications are modelled on a vertical line, more effective usage of resources leads to higher performance and therefore a higher position on the line. When an application moves on the flops/byte axis its computational intensity changes, more flops/byte means more computations on data.

effect of the lower upper bound is visible in figure 2.4a where application A2 does not reach the roof of the model while intuitively this should be possible. An explanation of this lower upper bound in the model can be found in section 2.5.

4)No modelling of the Field Programmable Gate Array (FPGA): The model is based on issued instructions, therefore it is unknown whether it can be used to model a FPGA.



(a) *Original:* In the original model the external memory accesses are regarded when calculating the operations per byte of an application. This results in a change in operations per byte when using internal memory or interconnects. for example, application A1 has 8 operations and applies them on 8 bytes from the external memory (1 ops/byte). By improving A1 it becomes application A2 and uses 7 bytes from the internal memory and on from the external memory, this results in a change in ops per byte, 8 operations on 1 byte from the external memory (8 ops/byte), and an increased performance



(b) *Contribution:* The addition of extra data sources to the Roofline model, e.g internal memory , interconnect etc. This means multiple data sources are taken into account when calculating the bandwidth Roofline and ceilings of a processing unit. This minor change has a benefit when plotting applications in the Roofline model because the computational intensity does not change when using bytes from other sources. The same application A1 and its improvement A2 are plotted in this figure. Performance increases in the same manner as the original model

Figure 2.4: Differences between the original and the extended Roofline model

type	bits	ops	bytes	ops per byte
$char$	8	1	3	$\frac{1}{3}$
int	32	1	12	$\frac{1}{12}$
$long$	64	1	24	$\frac{1}{24}$
$float$	32	1	12	$\frac{1}{12}$
$double$	64	1	24	$\frac{1}{24}$
$char16[vec]$	$8 \cdot 8$	16	16	1
$int4[vec]$	$32 \cdot 4$	4	16	$\frac{1}{4}$

Table 2.1: Additions performed on a number of data types, operations per bytes are calculated according to the number of operations that can be executed in parallel and the number of required bytes per data type. A vector operation $[vec]$ performs multiple additions on a number of data elements contained within a single operand. For example a $char16[vec]$ operand contains 16 operands of type char.

2.2 Contributions to the Roofline Model

Solutions to the shortcomings of the Roofline model are presented in this section. The solutions are contributions to the original model and make the model better suited for the proposed processing unit selection method. These contributions improve the model to make it general applicable to a variety of processing units. To do a proper performance estimation and comparison across different processing units a measure is needed that is suited for different processing units and data types. The original Roofline model is based on floating point operations, however floating point operations are not supported by all processing units (e.g an FPGA has limited support for these operations). An operation is defined as a arithmetic function on one or more operands. Table 2.1 depicts the influence of different data types for an addition operation which uses two input operands and produces one result. The first column from the left depicts the data size in bits of one operand. The second column depicts the number of operations that are executed when performing one addition on this operand. The third column depicts the number of bytes that are accessed from the memory when performing the addition (e.g. an addition on a char data type results in 2 input bytes and one result byte so in total 3 bytes are accessed), and the last column depicts the operations per byte ratio of one addition of this data type. Besides the known data operations such as char (character) and int (integer), other data operations are listed in the table, namely vector operations marked with $[vec]$. A vector operation $[vec]$ contains multiple data elements of the corresponding type. For example $char16[vec]$ means one vector operand contains 16 $char$ elements. Therefore one addition of this type results in 16 operations for the implicit $char$ type in the vector. Table 2.1 lists additions on different data types and how this results in different operations per byte ratios.

2.2.1 Contribution 1 - Mixing data types

The Roofline model is adjusted to support any data type instead of solely using floating point data. To accomplish this both axes of the Roofline model are changed to list operations instead of floating point operations. These operations can be performed on different data types. The chosen type

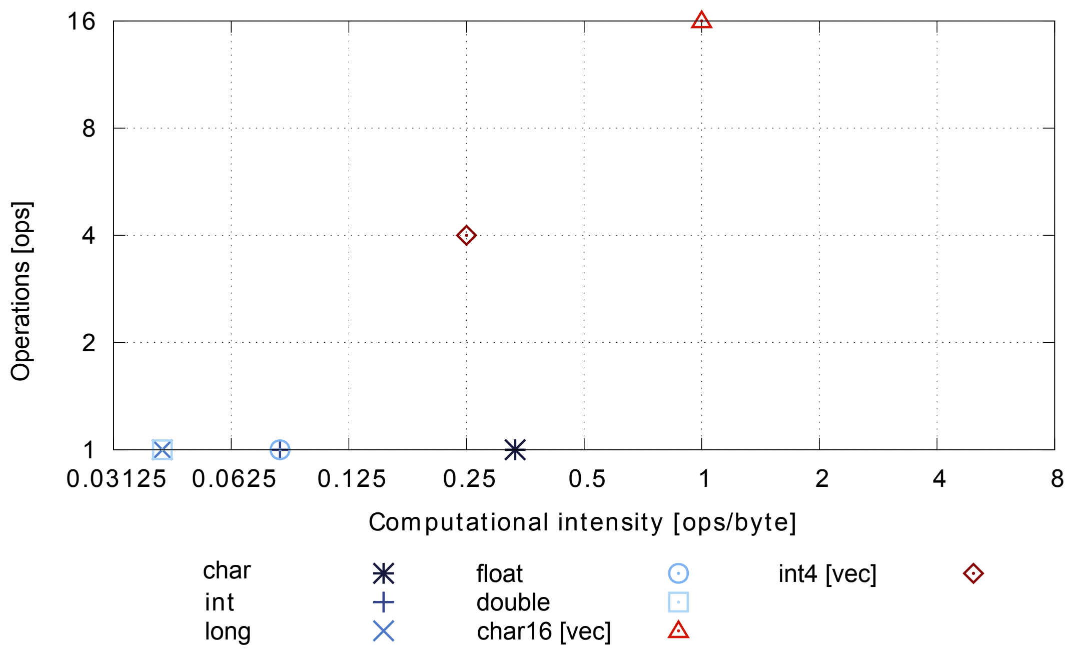


Figure 2.5: An application contains 10^9 *additions executed on the data types of table 2.1 The different data types from table 2.1 are plotted in the Roofline model, this results in points within the model.*

influences the speed at which operands are processed. With this adjustment a mix of instructions in an application is allowed whereas the original model without adjustments can only model a single data type and the corresponding vector (Single Instruction Multiple Data (SIMD)) operations.

The processing units are selected based on the required performance of application, algorithm and GCB. Making a good comparison between different processing units based on their performance is key for a good selection. In this thesis processing units are compared by executing 32 bit operations or vector variants of them. For example int, float, int4[vec] from table 2.1 are valid operations to use for comparison. More data types could have been selected, however these operations are supported by all processing units that are compared in this document. Depending on the application the different data types can be used to model applications and processing units e.g. operations on 8 bit data types. The operation per byte ratio can be different per data type, as is shown in figure 2.5.

2.2.2 Contribution 2 - Extra data sources

The requirement of an application in the Roofline model is expressed in operations per byte and operations per second. This is explained in depth in chapter 7. These requirements result in a point within the Roofline model as depicted in figure 2.4b and 2.4a. Assume the Roofline model is used (figure 2.4a) and application A1 has the following properties, it executes 8 operations on 8 bytes from the external memory. A1 is limited by the external memory bandwidth of the PU.

Improving the performance of the application by using the available internal memory results in A2, which reads 1 byte from the external memory and uses 7 bytes from the internal memory, note that the same amount of operations (8) are executed on 8 bytes. In the original model this results in an increased performance of the application (A2) depicted in figure 2.4a. By reading less bytes from the external memory an increase in $[ops/byte]$ ratio is observed in the original model. The performance of A2 $[ops/sec]$ also increases because the internal memory is faster and does not bound the computational performance of the application.

Changing the data source of bytes should not change the computational intensity, the algorithm does not perform more or less computations on data, the only thing that changed is the location of the data. That is why changing the usage of different data sources should not influence the computational intensity (8 operations are still executed on 8 bytes). The second contribution to the Roofline model adds extra data sources such as internal memory and interconnect which solves this problem. Adding extra data sources makes sure the $[ops/byte]$ ratio does not change when multiple data sources are used in an algorithm. This contribution removes the assumption of the original Roofline model that the data bottleneck is formed by the external memory. By changing this assumption, algorithms can read bytes from different sources, e.g. external, internal memory, interconnect. These extra data sources are added to the processing unit model as depicted in figure 2.6. The bandwidth of these data sources are then added in the Roofline model as a bandwidth ceiling or roof. This is depicted in figure 2.4b and 2.7 that show the internal memory as well as the external memory bandwidth. In this thesis the interconnect (e.g PCI-e), internal memory (e.g. cache or local memory) and external memory (e.g. DRAM) are used while extra data sources can be added if required by the application or processing unit. Examples are hard disk Input-Output (IO), flash memory, Multi Processor System On Chip (MPSoC) interconnect etc.

To show the advantage of this contribution the previous mentioned application A1 is considered again. A1 executes 8 operations on 8 bytes from the external memory which places A1 on 1 operation per byte in the Roofline model, depicted by A1 in figure 2.4b. A1 is still limited by the external memory bandwidth of the PU. The performance of the application is improved by using the available internal memory, this results the same application A2 as depicted in figure 2.4a. A2 uses 1 byte from the external and 7 bytes from the internal memory, and applies 8 operations. Note that because multiple data sources are modelled, the operation per byte ratio remains 1 and the performance increase is identical to the original model. Figure 2.4b depicts the original application A1 and the improved version A2 in the Roofline model applying the this contribution with extra data sources.

2.2.3 Contribution 3 - A tighter upper bound on performance

The last contribution to the Roofline model is the calculation of a tighter upper bound on the computational and memory performance roof. This tighter bound is created by analysing the operations and memory accesses of an application in conjunction with the available processing inside a processing unit. This analysis results in the utilisation Roofline and depicts how well the application uses the parallelism and available resources inside a processing unit. Section 2.5 explains how this utilisation

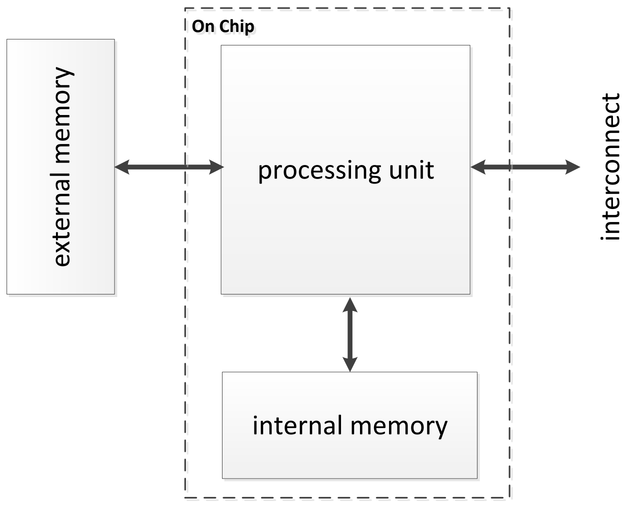


Figure 2.6: The original Roofline model is extended with extra bandwidths to better model a real processing unit, which transfers data from and to different sources, e.g., internal memory, external memory or an interconnect. This also makes sure the $[ops/byte]$ ratio does not change when changing the source of data in an algorithm. Data sources can be of different types as long as the bandwidth of this source is known. The processing units in this thesis assume the external memory and interconnect are off chip data sources while the internal memory is on chip. Examples of external memory are, flash memory, DRAM, examples of interconnect are PCI-e, USB and examples of internal memory are cache and scratch pad memory.

Roofline is calculated. For the rest of this document the extended Roofline model is defined as the original Roofline model with contributions 1,2 and 3 added to it. This solves three out of four short comings of the original Roofline model.

2.2.4 Contribution 4 - A Roofline model for an FPGA

The last hurdle of the Roofline model is that it is based on issued instructions which imposes difficulties when modelling the FPGA architecture. An FPGA has an flexible internal architecture in which the data path and instructions are not fixed or present, any architecture can be created in this kind of processing unit. In chapter 6 an example translation is done from the internal architecture of the FPGA onto the Roofline model to make it possible to compare the attainable performance of this unit with other processing units.

2.3 Roofline estimation errors

Estimating the operations and bytes that an application requires is not a trivial task e.g. compilers add and remove operations that an designer expected to be present. This introduces an error in the

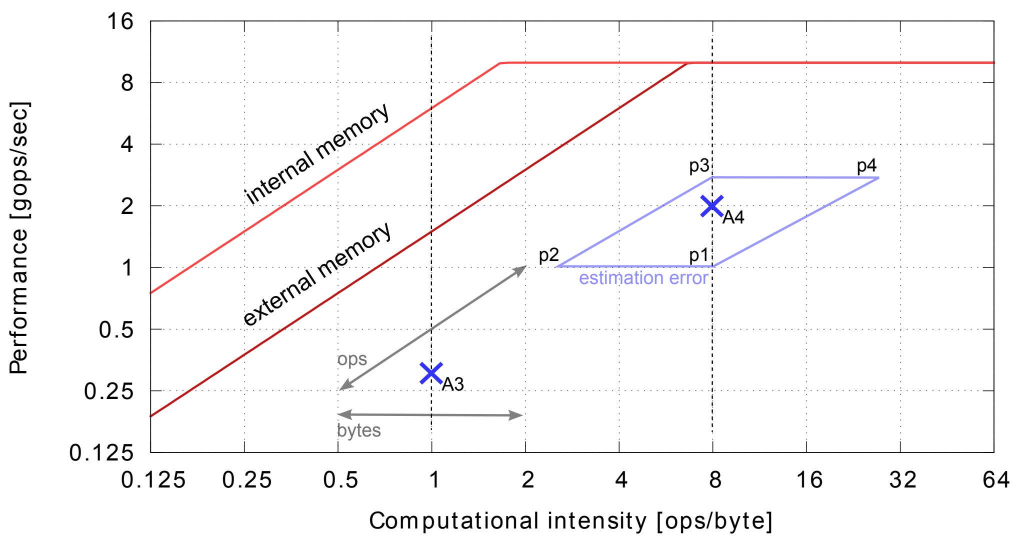


Figure 2.7: The requirements of applications A3 and A4 are estimated in terms of operations per byte and operations per second. Not every aspect of the application can be determined on forehand, therefore an estimation error is assumed. Making an error estimating the number of bytes results in an horizontal displacement and an error in estimating the number of operations results in a diagonal displacement depicted by the arrows around A3. The estimation error spans a plain around the estimated application. This is depicted by A4 which is estimated at 8 ops/byte and 2 gops/sec with an estimation error of 50%, note that an estimation error on the number of operations results in an diagonal displacement.

number of operations and or bytes. An estimation error results in a displacement of the application within the Roofline model. The application moves in the horizontal direction within the Roofline model when an estimation error is made in the number of bytes the application processes. A diagonal displacement is observed when the number of operations is estimated incorrectly. This is caused by the operations having influence on both performance and computational intensity. These displacements are depicted in figure 2.7. An application with an incorrect estimation can give a false impression in the Roofline model. By defining the estimation error as ε, a plane can be added in the model. This plane depicts the error plane of the application. If a designer wants to have more guarantees on performance he/she must make sure the plane is situated below the Roofline of the model. Figure 2.7 depicts an estimation error of 50% on both operations and bytes for application A4. The four points of the plain and the corresponding operations and bytes $(Eops, Ebytes)$ are defined as follows. In this definition $\varepsilon = 0.5$ for an estimation error of 50%, and ops and $bytes$ are the operations and bytes of the estimation of the requirements of application A4.

$$p_1 \rightarrow Eops = ops - \varepsilon \cdot ops,\ Ebytes = bytes + \varepsilon \cdot bytes$$
$$p_2 \rightarrow Eops = ops - \varepsilon \cdot ops,\ Ebytes = bytes - \varepsilon \cdot bytes$$
$$p_3 \rightarrow Eops = ops + \varepsilon \cdot ops,\ Ebytes = bytes - \varepsilon \cdot bytes$$
$$p_4 \rightarrow Eops = ops + \varepsilon \cdot ops,\ Ebytes = bytes + \varepsilon \cdot bytes$$

To reduce the estimation error estimations should be automated by tooling like code analysis programs. This tooling must be able to extract operations and data accesses from an application while anticipating on compiler or programming optimizations. More information about analysing applications can be found in chapter 7 and in particular the operations of an application in section 7.3.

2.4 Determining a Roofline model

In section 2.1 and 2.2 the Roofline model and contributions to the model are introduced. This section discusses the process of creating a Roofline model for a processing unit with external memory, internal memory and an interconnect. In this way the Roofline model is usable for a Central Processing Unit (CPU), a Digital Signal Processor (DSP), a Graphical Processing Unit (GPU) and an FPGA as will be shown in the next chapters.

By analysing the internal architecture of a processing unit its computational roof and memory roof can be calculated. The calculations used to determine the Roofline model are explained using an abstraction of a processing unit. This processing unit model is based on assumption that every processing unit is connected to an interconnect, and has external memory outside the processing unit to store data and has internal memory to store temporary results and smaller portions of the data, figure 2.6 illustrates this assumption.

The computational roof and ceilings, and the memory roof and ceilings are defined by a set of equations. The parameters of these equations can be visualised in a more detailed model. This more detailed model is called the simplified processing unit (SPU) model and is depicted in figure 2.8.

2.4.1 Computational ceilings

The computational ceilings are based on the internal architecture of a processing unit. This section explains how the ceilings are calculated using the SPU model. The SPU model assumes p cores, with the same internal architecture (i.e. homogeneous). The internal architecture of the cores consists of q functional units. Every functional unit (FU) has different functionalities that can be grouped into three types:

- Memory functional units which fetch data from the memory.
- Operational functional units that perform operations on data.
- Combined functional units at can access data or perform operations.

All functional units that can perform operations are taken into account when calculating the operations of a processing unit core (C_{ops}). Functional units are controlled by an clock signal called the core clock that runs at a frequency C_f. This clock defines at which speed operations can be executed. In the SPU model the assumption is made that all cores are connected to the same core clock, with frequency C_f.

Core operations of type i (C^i_{ops}), are defined as arithmetic or logical calculation on 4 bytes (32 bits) of data. This definition allows floats and integers to be used in the same Roofline model, and is necessary to be able to compare different processing units in the next chapters. Every function unit

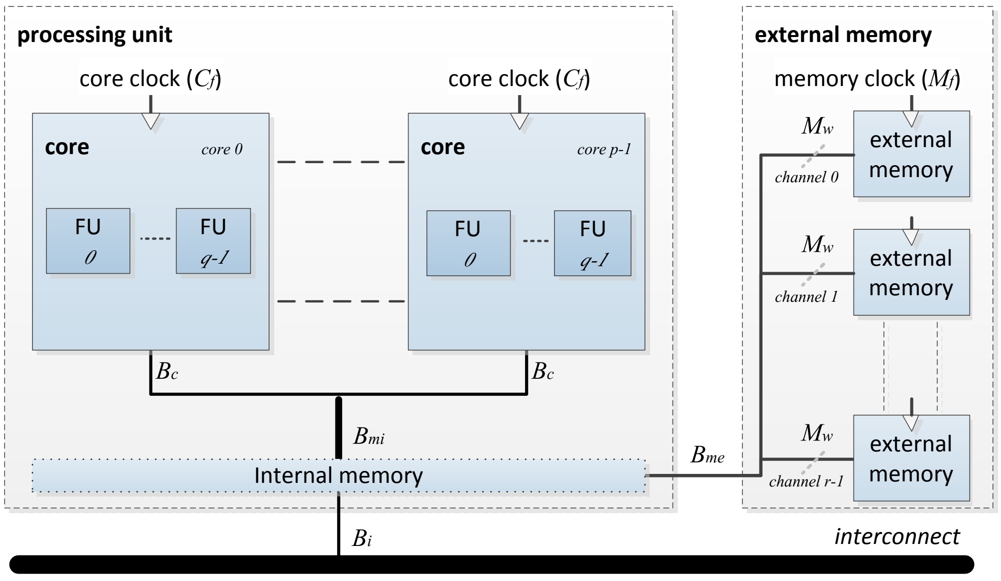


Figure 2.8: Simplified processing unit model which is used to calculate the theoretical Roofline of a processing unit. The model visualises the different parameters of the formulas used to calculate the roof and ceilings of a processing unit. A processing unit contains p cores, which have q function units. The processing unit accesses data from the external memory, internal memory or interconnect which all have a fixed bandwidth (B_{me}, B_{mi}, B_i).

that supports operation type i can perform a fixed amount of this type of operation per clock cycle ($FU(i)_{width}$). This implies it can execute $FU(i)_{width}$ number of operations of type i per clock cycle. The total number of C^i_{ops} that can be executed in one clock cycle equals the sum of all function units that support operation i in parallel (equation 2.2). Combined functional units are assumed to perform operations on data (so no memory accesses) and memory functional units are not counted in this equation (the $FU(i)_{width}$ of memory functional units equals 0 for all i).

The computational ceiling (equation 2.1) is then defined as the core operations multiplied by the number of cores (equation 2.3) multiplied by the frequency at which a core can execute the operations C_f (equation 2.4). From this calculation it is known that the computational roof is expressed as number of operations per second. If the frequency of a core C_f is expressed in Hz then the computational ceiling is expressed in operations per second. Additionally more then one computational ceiling may exist depending on the data processing behaviour of the processing unit, e.g. for a given PU floating point values are calculated slower then integer numbers which results in different ceilings for integers and floating point numbers.

$$C^{i}_{ceiling} = C_f \cdot C_{\#cores} \cdot C^{i}_{ops} \qquad [ops/sec] \tag{2.1}$$

$$C^{i}_{ops} = \sum_{k=0}^{q-1} FU(i)^{k}_{width} \tag{2.2}$$

$$C_{\#cores} \overset{def}{=} \text{number of cores} \tag{2.3}$$

$$C_f \overset{def}{=} \text{core clock frequency} \qquad [Hz] \tag{2.4}$$

2.4.2 Data source ceilings

In the extended Roofline model presented in this work, multiple memory bandwidths are introduced. The SPU model (figure 2.8) assumes three main data locations exist i.e. internal memory, external memory and interconnect. These memory locations all have a corresponding bandwidth B_{mi} for the internal memory bandwidth, B_{ic} for the interconnect bandwidth and B_{me} for the external memory bandwidth. It is assumed that memory functional units can load and store bytes to the memory in parallel with operational functional units performing operations on data. Combined functional units are assumed to only execute memory transfers instead of operations in order to calculate the memory bounds.

The internal memory contains local copies of data, it is located on chip to have a high bandwidth connection to the processing cores. The total maximum bandwidth for all cores is depicted by B_{mi}. However a core can have its own limit on bandwidth to the internal memory depicted by B_c. The relation between B_{mi} and B_c is then defined as depicted in equation 2.5. For example a single Intel Xeon core supports 16 byte loads per clock cycle (B_c), while the total internal memory bandwidth to the cache is much higher namely that of four cores in parallel (B_{mi}). Another example is an MPSoC that connects multiple cores via an on-chip interconnect with a bandwidth of B_{mi} when multiple cores use the interconnect the bandwidth per core drops $\frac{B_{mi}}{\text{cores using the interconnect}} = B_c$. Which one to use depends on the internal organization of the cores and what kind of application a designer uses (e.g. multi core or single core).

$$B_c \leq B_{mi} \qquad [byte/sec] \tag{2.5}$$

External memory is larger and is located off-chip, it is often slower then internal memory and faster then the interconnect. The speed at which data can be loaded and stored to the memory is defined by B_{me}

The third location where data can reside is the interconnect, which has a fixed bandwidth B_{ic}. Via the interconnect, data can be received from other processing units e.g. a GPU receives data from the CPU via the PCI-e interconnect located on a motherboard of a PC.

Every memory bandwidth B_j of type j such as B_{ic}, B_{me}, B_{mi} can be calculated by the following pa-

rameters. • M_f the memory clock frequency at which a memory unit issues transfers. M_{dr} is the data rate, which defines the number of data transfers that are executed per clock cycle, e.g. $M_{dr} = 1$ means 1 cycle for every transfer, $M_{dr} = 0.5$ depicts 2 cycles are required for every transfer and $M_{dr} = 2$ defines half a cycle is required for a transfer.
• M_w defines the number of bytes contained in one memory transfer.
• $M_{\#channels}$ is the number of memory channels available, every channel has the same M_f, M_{dr} and M_w. Doubling the channels doubles the theoretical bandwidth (B).

$$B_j = M_f^j \cdot M_{dr}^j \cdot M_w^j \cdot M_{\#channels}^j \qquad [byte/sec] \tag{2.6}$$

$$M_f \overset{def}{=} \text{memory clock} \qquad [Hz] \tag{2.7}$$

$$M_{dr} \overset{def}{=} \text{transfers per clock cycle} \qquad [transfers/cycle] \tag{2.8}$$

$$M_w \overset{def}{=} \text{size of single transfer} \qquad [bytes] \tag{2.9}$$

$$M_{\#channels} \overset{def}{=} \text{number of memory channels} \tag{2.10}$$

2.4.3 Creating the model

The Roofline model is defined as a graph that depicts computational intensity versus computational performance. The Roofline model defines the computational intensity of an algorithm in operations per byte ($[ops/byte]$). The higher the computational intensity the more operations are performed on bytes from the memory. The computational performance of the model is defined as the number of operations that can be processed every second $[ops/sec]$.

The performance of an processing unit is limited by either the memory bandwidth or the maximum computational performance. The computational intensity ($[ops/byte]$) is used to plot the memory bandwidth and the computational performance in a single plot using equation 2.12. This equation defines memory ceilings depending on the available bandwidth and computational intensity. From the memory ceilings that are present in the model a memory Roofline is calculated which is the maximum of all ceilings, depicted in equation 2.13. The same holds for the computational Roofline depicted in equation 2.14, where the maximum of computational ceilings defines the computational roof.

By combining the computational roof and memory roof in equation 2.15 the Roofline of a processing unit is constructed. When replacing the roofs in this equation by the available ceilings different memory and computational ceilings can be added.

$$\text{computational intensity} \overset{def}{=} x \qquad [ops/byte] \tag{2.11}$$

$$M^{j}_{ceiling}(x) = B_j \cdot x \qquad [ops/sec] \tag{2.12}$$

$$M_{roof}(x) = MAX\left(M^{1}_{ceiling}(x), M^{2}_{ceiling}(x), ...\right) \qquad [ops/sec] \tag{2.13}$$

$$C_{roof} = MAX\left(C^{1}_{ceiling}, C^{2}_{ceiling}, ...\right) \qquad [ops/sec] \tag{2.14}$$

$$\text{performance}(x) = MIN\left(M_{roof}(x), C_{roof}\right) \qquad [ops/sec] \tag{2.15}$$

2.5 Application utilisation Roofline

The third contribution to the original Roofline model is the addition of a utilisation Roofline to the model (depicted in section 2.2). The utilisation Roofline is a tighter upper bound on the Roofline model of the processing units by taking the operations and memory accesses of an application, algorithm, or GCB into account. This utilisation Roofline is defined in this section and gives a bound on the maximum performance of an application, algorithm or GCB on a given processing unit.

A GCB within an algorithm or application has a known number of computations and data accesses. Data required by the GCB is often accessed from a number of data sources, e.g. internal memory, external memory, or interconnect. Furthermore operations within the GCB are part of different computational ceilings, e.g. float, vector, or integer. This makes it possible to define a GCB, algorithm or application as a combination of memory and computational ceilings. These ceilings correspond to an operation or data type e.g the floating point ceiling defines the performance of the floating point operation type.

Operations in a GCB, algorithm or application are of a type (i) and the total number of executed operations of this type is given by CQ_i. Additionally MQ_j is the total amount of data reads and writes to a data source (j). The GCB, algorithm or application consists of n different operation types and can access m data locations. Every operation type i corresponds to a computational ceiling ($C^{i}_{ceiling}$) in the Roofline model and the data location j corresponds to a memory ceiling ($M^{j}_{ceiling}$). The computational and memory ceilings are explained in section 2.4. Note that the following equations assume the ceilings are present in the Roofline model of the processing unit on which the application is executed. Using the quantities per type and per location, the total number of operations (equation 2.16) and the total number of memory accesses (equation 2.17) is defined.

$$CQ_{total} = \sum_{i=0}^{n-1} CQ_i \qquad [ops] \tag{2.16}$$

$$MQ_{total} = \sum_{j=0}^{m-1} MQ_j \qquad [bytes] \tag{2.17}$$

Executing operations requires time; the total time required for all computations in an application is defined as tc. The time a operation of type i takes on the processing unit can be calculated by dividing the number of operations (CQ_i in the application by the speed at which a operation can be executed $C_c^i eiling$(equation 2.18). The same applies to the total time spend for memory accesses tm in which the number of bytes accessed MQ_j from data source j is divided by the bandwidth B_j of the data source to get the time per access type (equation 2.19).

$$tc = \sum_{i=0}^{n-1} \frac{CQ_i}{C_{ceiling}^i} \qquad [sec] \tag{2.18}$$

$$tm = \sum_{j=0}^{m-1} \frac{MQ_j}{B_j} \qquad [sec] \tag{2.19}$$

The utilisation Roofline (2.21) consist of two roofs, the Computational Utilisation Roof (cur) (2.22) and Memory Utilisation Roof (mur) (2.23). These roofs use the maximal performance in operations per second of the GCB on the processing unit and the bandwidth in bytes per second of the GCB on the processing unit. The maximal performance is calculated by using the total time tc required to execute all operations and the total number of operations CQ_{total} in the application. The executions of operation types in parallel is not taken into account. The bandwidth is calculated using the time required for all memory accesses tm and the total number of accesses to the memory MQ_{total}.

$$\textit{computational intensity} \overset{def}{=} x \qquad [ops/byte] \tag{2.20}$$

$$\textit{utilization-roof}(x) = MIN\,(mur(x), cur) \qquad [ops/sec] \tag{2.21}$$

$$cur = \frac{CQ_{total}}{tc} \qquad [ops/sec] \tag{2.22}$$

$$mur(x) = \frac{MQ_{total}}{tm} \cdot x \qquad [ops/sec] \tag{2.23}$$

Example utilisation Roofline calculation

To illustrate the utilisation Roofline, an example is discussed, this example depicts how a utilisation Roofline can be calculated for an application and processing unit that have the following properties.

Assume the application with the properties from table 2.2a uses two types of operations $i = \{0, 1\}$ and accesses data from two memory locations $j = \{2, 3\}$. 75 operations of type 0 are executed and

25 of type 1. The application uses (total read + write) 50 bytes from memory type 0 and 50 bytes from type 1. In total this application processes 100 bytes and executes 100 operations this yields 1 operation per byte for computational intensity. The ceilings and bandwidths of the processing unit are also of type $i = \{0, 1\}$ and $j = \{2, 3\}$ and have corresponding performance listed in table 2.2b

Combining these parameters using the previously defined equations a utilisation Roofline is created, this Roofline is illustrated in 2.9a.

CQ_0	CQ_1	MQ_2	MQ_3
$75[ops]$	$25[ops]$	$50[bytes]$	$50[bytes]$

(a) Application parameters

$C^0_{ceiling}$	$C^1_{ceiling}$	B_2	B_3
$12[Gops/sec]$	$8[Gops/sec]$	$8[Gb/sec]$	$2[Gb/sec]$

(b) Processing unit parameters

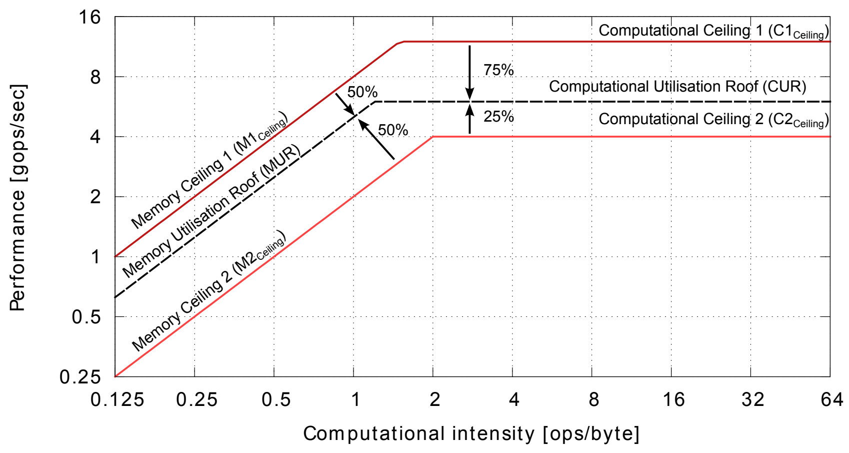
Table 2.2: Application and processing unit parameters for calculating the utilisation Roofline. An example of the calculation is given below

$$\begin{aligned}
CQ_{total} &= CQ_0 + CQ_1 = 75 + 25 = 100 && [ops] \\
MQ_{total} &= MQ_2 + MQ_3 = 50 + 50 = 100 && [bytes] \\
t_0 &= \frac{CQ_0}{C^0_{ceiling}} = \frac{75}{12} = 6.25 \quad , \quad t_1 = \frac{CQ_1}{C^1_{ceiling}} = \frac{25}{8} = 3.125 && [sec] \\
t_2 &= \frac{MQ_2}{B_2} = \frac{50}{8} = 6.25 \quad , \quad t_3 = \frac{MQ_3}{B_3} = \frac{50}{2} = 25 && [sec] \\
cur &= \frac{CQ_{total}}{t_0 + t_1} = \frac{100}{6.25 + 3.125} = 10.67 && [ops/sec] \\
mur(x) &= \frac{MQ_{total}}{t_2 + t_3} \cdot x = \frac{100}{6.25 + 25} \cdot x = 3.2 \cdot x && [ops/sec] \\
\textit{utilization-roof}(x) &= MIN\,(mur(x), cur) = MIN\,(3.2x, 10.67) && [ops/sec]
\end{aligned}$$

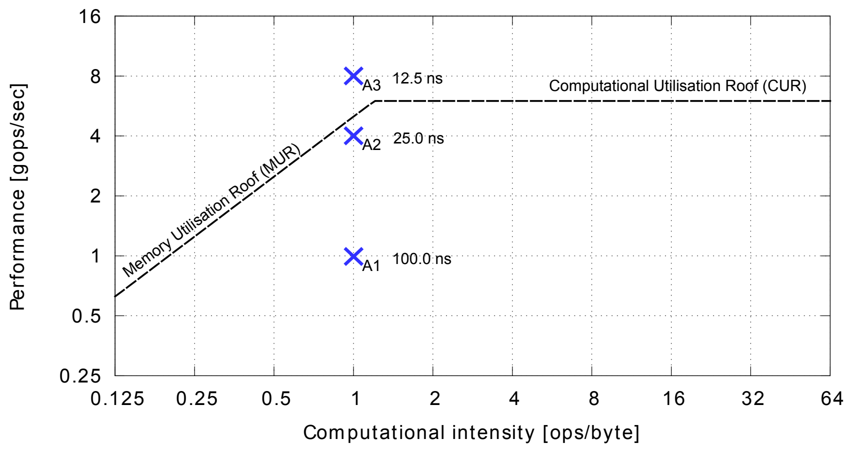
The utilisation Roofline can result in a tighter upper bound on performance compared to the original model (illustrated in figure 2.9a). When the utilisation Roofline is available the requirement of an application can be calculated depending on the time constraints of an application. From the application data 2.2a and a time constraint T in seconds, the requirement of the application can be plotted in the Roofline model. The time constraint T defines a period in $seconds$ in which the 100 ops and 100 $bytes$ of the application must be executed e.g. one frame of a video needs to be processed in T seconds. 100 ops in 100ns requires one giga operations per second of performance from the processing unit (equation 2.24 and figure 2.9b).

$$100[ops] \cdot \frac{1}{T} = 100 \cdot \frac{1}{100ns} = 1 \qquad [Gops/sec] \tag{2.24}$$

Figure 2.9b depicts multiple example requirements in the Roofline model. The 12.5ns requirement is situated above the utilisation Roofline, this implies that the application running on this particular processing unit cannot meet the requirement of 12.5ns.



(a) *Creating the utilisation roofline:* The application executes 75% of the operations from ceiling C0 and 25% of operations from C1. The resulting CUR is a weighted mean of these two ceilings. The MUR is the weighted mean based on the amount of bytes transferred from and to data source M0 and M1



(b) *Time constraints:* Applying different time constraints to the application has no effect on the utilization Roofline. By changing the time constraints the requirement of the application shifts in a vertical direction. Three different period requirements of an application (100 ops & 100 bytes) are illustrated in this figure: 100 ops * 1/100ns = 1 Gops/sec, 100 ops * 1/25ns = 4 Gops/sec, 100 ops * 1/12.5ns = 8 Gops/sec

Figure 2.9: Utilisation Roofline for an application and processing unit

2.6 Verifying a Roofline Model

To evaluate if the Roofline model is suited for hardware selection different processing units and the resulting theoretical model are compared to the real-life performance. Micro benchmarks can be created on every architecture to verify the theoretical Roofline to a measured Roofline, and to show that this method can represent the processing performance of a processing unit. These results can then be used to make a better estimation of the theoretical roof in the future.

A micro-benchmark measures the time it takes to execute a artificial algorithm. This algorithm has a complexity defined by the number of bytes it loads and stores from and to the memory and the number of operations that are executed on this data. This artificial algorithm mimics the use of a given operation type from a selected data source e.g. floating point operations on data from local memory.

The micro-benchmarks used in this work perform reads from a data source, execute operations on the data and write the data back to the data source. Figure 2.10 depicts this. Depending on the kind of processing unit the memory bandwidth is the performance limit for $<< 1$ computational intensity and computational performance is the limit for $>> 1$ computational intensity.

The reads are performed on a read buffer which can be stored on different data sources (e.g. internal, external memory or interconnect). Every loop iteration of the benchmark, x bytes are read from the buffer and y operations are executed on these bytes. The next iteration the next x bytes are read from the read buffer so that the loop reads the buffer in a sequential way. Results that are calculated in a loop operation are stored as z bytes into a write buffer which is also written in a sequential way. Sequential access of the memory is select in this case because most memories are optimized for this kind of access behaviour.

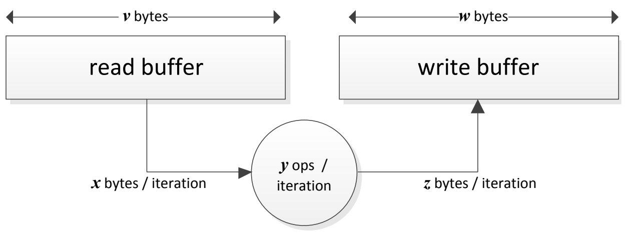


Figure 2.10: Micro-benchmarking is performed in the following way: two buffers are created a read buffer of v bytes, and a write buffer of w bytes. The measurement is performed with a loop in which y operations are executed on x bytes of the read buffer. The resulting z bytes are stored into the write buffer. The loop repeats, and the total time of all loops is measured. When all loops are finished the average time per loop is calculate. This yields the operations per second for a fixed operational intensity of $\frac{y}{x+z}$ operations / byte. The read and write buffer may have varying data locations to measure memory bandwidth of external, internal or interconnect. The number of bytes and the number of operations can be varied to measure different computational intensities.

Chapter 3

CPU Roofline model

The Central Processing Unit (CPU) is a general purpose processor that can be found in many systems. These systems range from a regular desktop computer to mobile phones and other embedded platforms. This chapter shows how a Roofline model can be constructed for a CPU and verifies this model using micro-benchmarks on three of them.

This chapter is organized as follows, First background information regarding the CPU is discussed in section 3.1. Then section 3.2 explains how the SPU model can be linked to a CPU and from this construct the theoretical Roofline model. Section 3.3 discusses which mirco-benchmarks are performed to verify the theoretical Roofline of a processor. The theoretical Roofline is then created and verified for three CPU types in sections 3.4, 3.5 and 3.6. The last section lists what kind of issues can form a bottleneck on the performance of an application on the CPU.

3.1 Background

In the '40s, mathematicians John von Neumann, J. Preper Eckert and John Mauchly came up with the concept of the stored instruction digital computer. Before this time computers were machines which had to be physically rewired to perform a different task. The stored instruction digital computer was born by using a memory to store instructions and data. These instructions control logic to perform different tasks on the data. Figure 3.1 illustrates the components of an store instruction digital computer namely the memory and the CPU. The CPU carries out operations on data stored in the memory and consists of two units, a function unit that performs operations on data from the memory, and a control unit that controls the process of fetching, decoding and executing these operations. Besides these two units, the CPU also contains a data path and a control path, the control path is used to control operation of memory and function unit while the data path transfers the required data and instructions.

Figure 3.2 illustrates a more in depth view of the CPU. This view adds instruction and data registers

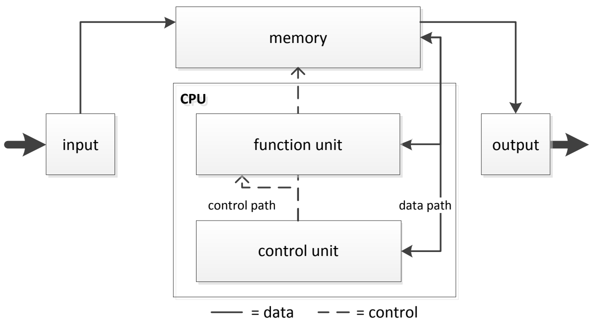


Figure 3.1: The function unit performs operations on data from the memory and a control unit controls the process of fetching, decoding and execution these operations. The control path is used to control memory and function units while the data path transfers the required data and instructions.

(A,B,C) and shows a function unit contains a Arithmetic Logic Unit (ALU). All actions performed by the CPU are synchronised on the CPU clock. Typically, every clock cycle an instruction is fetched from the memory by the control unit (CU) of the CPU and stored into the instruction register . The control unit then decodes the instruction and fetches the corresponding operands from the memory. These operands are then stored in registers of the ALU. Afterwards the CU instructs the ALU to execute an operation on the operands in the input registers (A,B), The result from this operation is then stored in the output register (C) of the ALU [1]. A cycle in which instructions are fetched from the memory decoded and executed is called a Von Neumann Cycle, hardware or logic implementing this cycle is said to have a Von Neumann Architecture. An extension to the Von Neumann Architecture is the so called Harvard architecture in which data (operands) and instructions are stored in separate memories. With this architecture the CPU is able to fetch the next instruction in parallel with operands for the current instruction. Almost all modern CPUs use this architecture type.

Figure 3.2 depicts an CPU with a single ALU. However modern CPUs often contain multiple function units. Figure 3.3 shows an example of a CPU with multiple function units. In these modern CPUs every ALU is dedicated to a given set of tasks, e.g. one function unit performs loads and stores to the memory while another performs integer and vector operations. Both function units can perform their tasks in parallel, but only one instruction at the time. This implies that the example function unit that contains two ALUs can only use one of them per instruction.

To control multiple function units, multiple instruction registers are available these are often called instruction ports. An instruction port corresponds to a function unit for example in figure 3.3 instruction port 1 contains an instruction for function unit 1. These instruction ports are supplied with instructions by an instruction scheduler. This scheduler fetches an instruction from the instruction stream and issues it into a free instruction port that supports that type of instruction.

[1]Note that data registers can also be represented by locations in a register file

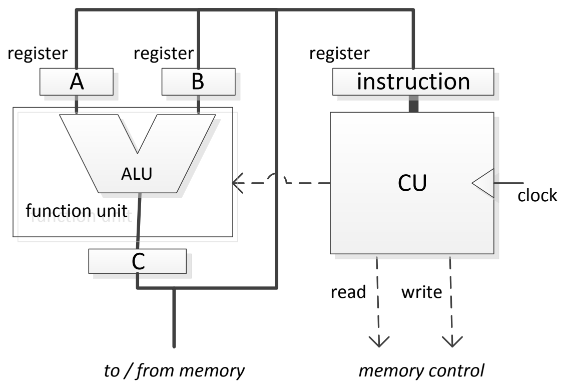


Figure 3.2: Basic CPU concept containing ALU, memory and a control unit (CU)

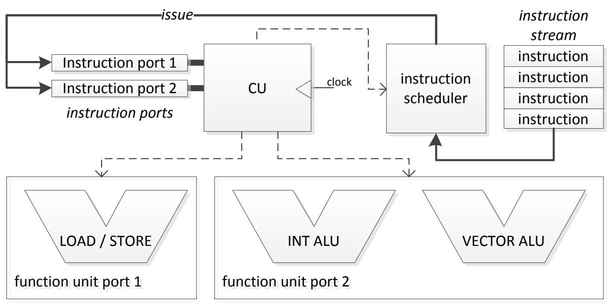


Figure 3.3: Modern CPUs contain multiple ALUs to process operands in parallel

3.1.1 Instruction and data reordering

When a instruction is issued into a free instruction port the corresponding function unit is configured for this to execution the operation contained in the instruction. However instructions from the sequential stream are not always ordered in such a way that all the function units can be used every clock cycle. Worst case two consecutive instructions need to be executed by the same function unit. In that case the CPU stalls because it cannot issue the next instruction. Two principles can be applied to overcome this problem.

• **Off-line scheduling:** The first option to prevent instruction stalls is creating an instruction stream with a correct order (schedule) before the program is executed. This can be done by hand (writing assembly) or by a compiler. In the latter case the compiler creates a schedule that is based on the internal architecture of the CPU. However the compiler needs knowledge about the internal architecture and the available function units ,ALUs and registers of a given CPU. The compiler then checks for instruction and data dependencies by static analysis of the code of an application.
• **On-line scheduling:** In the case of on-line scheduling the internal architecture creates a schedule during the execution of a program. For this the architecture contains a reorder unit which dynamically reschedules the instructions in the instruction stream. It also keeps track of the corresponding data to know which data to commit to the memory and in which order. Runtime dependencies in instructions and data can be overcome by the reorder unit. The CPU is called *in-order* when the instructions are executed in the order in which they appear in the instruction stream. It this case it is the responsibility of the compiler to schedule the instructions in the right order, to prevent the CPU from stalling. An execution stage executes the instructions *out-of-order* when the instructions in the instruction stream are rescheduled on-line inside the processor.

3.1.2 Memory interface and caches

The CPU interfaces with the data memory to load and store operands. The speed at which operands can be loaded from and stored to the external memory impacts the computational performance of a CPU. When operands can not be loaded or stored in time, function units of a CPU can stall. To overcome this problem modern CPUs contain on-chip data cache that temporary stores the required data.

This cache can reduce the average time required to get data from the memory. If this access time can be reduced depends on the temporal and spatial locality of an application. A cache controller manages the cache and keeps track of the available data in the cache. It decides which data in the cache to replace when accesses to the memory are done. A variety of cache controller exist simple ones that only apply a replacement policy (e.g. First In First Out (FIFO), Least Recently Used (LRU)), up to cache controllers that anticipate on memory accesses patterns and predict which data to load from the memory in advance.

The cache closest to the processing unit is called Level 1 cache. When a CPU contains multiple levels of cache each cache level often gets larger to increase the chance data is present, this also leads to a larger access time. The advantage in performance of cache is visible when data is reused (temporal locality of data), it speeds-up the application because long and slow memory accesses to the external memory are circumvented as the reused data is already present in the cache.

When a memory request for data is send to a cache it can happen that this data is not present on the current cache level (A cache MISS), in that case a lower (slower) level of cache is accessed and it takes longer to get the data. When the data is present at this level of cache (A cache HIT) data is immediately loaded, a fast memory access is the result.

3.2 Determining the theoretical Roofline

To determine the computational roof and memory bandwidth roof for the Roofline model of the CPU the model described in chapter 2 is used. Figure 2.8 depicts the different parameters used in the model. This section matches these parameters with parameters relevant for the CPU. First Instruction Set Architecture (ISA) and instruction throughput are introduced and then the theoretical roofs (computational roof and memory roof) are discussed to show how these are calculated.

3.2.1 Instruction Set Architecture (ISA)

The ISA defines which instructions, operands, registers are available for the programmer / compiler and how instructions are encoded into memory. For this research the encoding of instructions is not important, while the available instructions and their throughput is important for creating a Roofline model of the CPU. The computational ceilings and roof for the Roofline model of a CPU can be calculated by analysing the instructions of a processor. Three main categories of instructions exist in CPUs nowadays.

• Instructions which operate on one element from $\mathbb{Z}$, called integer instructions
• Instructions which operate on one element from $\mathbb{Q}$, called floating point instructions
• Instructions which perform one type of operation on multiple elements from $\mathbb{Z}$ or $\mathbb{Q}$ called vector or SIMD instructions.

Not all numbers from the $\mathbb{Z}$ and $\mathbb{Q}$ range are available within a CPU because this range is limited by the width (in bits) of the available registers and instructions e.g. 32 bit integers allow numbers in the range $-(2^{31})$ up to $2^{31}-1$.

Many ISAs are available e.g. MIPS, SPARC, Power, x86, ARM, Thumb, Thumb-II. For this research the x86 and ARM instruction set are selected, because these are the two most commonly used ISAs within Prodrive B.V. The ARM processor is a Reduced Instruction Set Computing (RISC) based processor having a limited amount of simple instructions. The argument for this is that 20% of available instructions is executed 80% of the time, and so complex instruction which are in the 20% part of the code are not required. The approach of having a fixed set of simple instructions, reduces the number of instructions required and makes the CPU architecture easier and reduces the required logic and power. x86 is a Complex Instruction Set Computing (CISC) based ISA, however the complex instructions are translated by a microcode translation unit to smaller simple instructions. These instructions can be processed by the RISC architecture inside the CPU. This approach makes it easier for assembler programmers because complex operations can be implemented with a single assembly instruction and this also reduces code size and instruction memory costs.

Besides these two ISAs x86 and ARM feature a number of extensions to support vector operations and floating point operations. For example ARM vector operations are grouped into the NEON extensions of an ARM CPU. The same holds for Intel CPUs that added SSE and AVX extensions to support vector operations.

3.2.2 Instruction throughput

A number of instructions exist within an processor to perform operations on data. The Roofline model is based on the throughput of instructions, therefore it is important to know the number of instructions that can be executed in parallel per clock cycle. Different types of instructions have varying throughput numbers. The ISA of a CPU defines which instructions are available for an x86 or ARM CPU, but the throughput of an instruction can be different for different CPU generations, brands and types. This difference is caused by the difference in micro-architecture inside an CPU. A micro-architecture is the implementation of the instruction set architecture in a CPU, depending on the kind of CPU the internal configuration changes. e.g. an Intel Atom only has 2 function units to process instructions while a large server CPU like the Xeon has 5 function units.

3.2.3 Computational roof

The computational roof of a CPU is determined by the functional units and the speed at which these can execute operations (figure 3.3). The operations which an functional unit performs can vary depending on the available ALUs inside the functional unit, therefore it matches the concept of a function unit from the SPU model (chapter 2). The computational roof is determined by the number of functions units and the throughput and type instructions that can be handled by these function units.

A CPU can have multiple cores, this matches with the cores in the SPU model of figure 2.8. In most cases all cores share the same core clock (C_f) but technologies exist which increase C_f when one or more cores is not used e.g. Intel turbo boost technology and AMD turbo core technology. For the calculations in this chapter it is assumed that all cores operate at the same fixed frequency. When turbo boost technology applies this is stated clearly.

A function unit within a CPU core can executed multiple types of instructions which correspond to multiple operations, this forms an analogue to function units from the SPU model. The width of a function unit is determined by the type of instruction (e.g. vector, float, double, integer etc.). By example, a function unit in a CPU supports the following operations; vector (4 x 32 bit) $FU_{width} = 4$, single precision floating point (32 bit) $FU_{width} = 1$, load and store $FU_{width} = 0$. To determine the computational roof the largest width for a function unit is selected, which in this case is $FU_{width} = 4$. When the maximum width of all function units is determined equation 2.2 and 2.1 are applied to calculate the computational roof.

Besides knowing which functional unit of a CPU supports which operation, C_{ops} can also be extracted by knowing the throughput of different instructions of the micro-architecture. Exact throughput numbers for an micro-architecture can be found in the programmers manual or datasheets of a CPU. Programmers manuals often state the reciprocal throughput of an instruction. Example: the x86 SSE2 *PADDD* instruction that operates on four 32-bit integers has a reciprocal throughput of $0,33$ on the Nehalem micro-architecture. So, three of these instructions can be executed every cycle, which results in 12 operations per cycle (four integer operations per instruction and three instructions per cycle). Implicitly this means the Nehalem micro-architecture has three function units in parallel which each can perform one *PADDD* instruction every cycle. For the SPU model this translates to three functional units which each support an FU_{width} of four.

3.2.4 Memory bandwidth roof

The SPU model defines three types of memory bandwidths to consider when estimating the memory Roofline of a processing unit B_{mi}. B_{me} and B_i. In the next paragraphs the three bandwidths and their relation within the CPU are made clear.

The internal memory bandwidth of the CPU is defined by the speed at which a processor can load and store operands from and to the cache memory, this speed represents the core bandwidth B_c of the SPU model. This speed is the same for every processing core and depends on the ISA and micro-architecture of the CPU. For instance a *MOVDQA* instruction in the Intel Nehalem micro-architecture moves 16 bytes from the cache memory to a vector register in one cycle (assuming a HIT in the cache). Every physical core has this bandwidth, so the total bandwidth of the internal memory becomes $B_{mi} = 4 \cdot B_c$ (matching equation 2.5 of the SPU model), for a CPU with four cores. The following equation represents this in this formula q represents the total number of cores.

$$B_{mi} \leq q \cdot B_c$$

Because the real implementation of the cache is often not known, B_{mi} can be smaller then calculated, but it gives an upper bound on performance. For the internal core bandwidth B_c the memory frequency is equal to the core frequency. $C_f = M_f$.

The external memory bandwidth of the CPU is defined by the speed at which data can be fetched from the memory when the cache has a MISS (when multiple levels of cache exist all levels have a MISS). The processor then reads data from external memory chips. These have a maximal bandwidth e.g. one channel of DDR3-1066 has a peak transfer rate of about 8533 MB/s. Some CPUs implement multiple channels of external memory, which has an analogue with the channels of the

SPU model.

A CPU often has an external connection to other devices e.g. GPU, FPGA etc. This is often a standardized bus or interconnect like PCI-e or ethernet. Example calculations of the above can be found in the next sections about ATOM, XEON and OMAP CPUs.

3.3 Determining the practical Roofline

This section discusses how the theoretical Roofline of a CPU is verified, and which things to keep in mind when measuring the performance of a CPU. This section also verifies the usability of the equations of the Roofline model for the CPU. Verification of the attainable performance is done using micro-benchmarks which measure the maximum computational performance at different computational complexities. The coming sections about the different CPUs then construct a theoretical Roofline and verify this Roofline using the information in this section.

3.3.1 Micro-benchmarks measurements

As depicted in section 2.6 x and z are the input and output data size of the micro-benchmark iteration, and y is the number of operations executed in one iteration. In total v are read linearly from the memory and w bytes are written linearly. A number of measurements are performed the approach of each measurement is discussed in the following lists.

The first mirco-benchmark verifies the computational roof and illustrates that combining different types of instructions is possible in the extended Roofline model.
• [int32] performing an addition on a 32 bit integer number in assembly
• [float32] performing an addition on a 32 bit floating point number in assembly
• [simd128] performing a int4[vec] (4 x 32 bit = 128bit) integer vector addition in assembly

In conjunction with the previous micro-benchmark the usability of the extra memory ceilings is verified by executing a cache and non cache benchmark. This can be accomplish by varying the buffer sizes of the read and write buffers of the benchmark.
• [cache] total size of the memory buffers $v + w$ that is allocated for the micro-benchmark is smaller then the cache size to force cache HITs. A dummy read is applied on the data to make sure it is located in cache when the micro-benchmark is started.
• [no-cache] total size of the memory buffers $v + w$ that is allocated is much larger then the largest cache size to force a cache MISS and so measure the external memory bandwidth.

Additional measurements are done by varying the manner in which the micro-benches are programmed, this to verify that the programming style can have influence on the performance of an application, algorithm or GCB.

• [assembly]programming the micro-benchmark in assembly and instruction the compiler and linker to not optimize the assembly code results in the most control over the number of operations and data transfers of the micro- benchmark, so a more accurate measurement is done which has minimal code overhead.
• [intrinsics] intrinsics are c / c++ functions which are implemented by the compiler. These functions use special data types to handle vector instructions and can be used to explicitly make c / c++ code use vector instructions in a piece of code.
• [plain-c] is the most naive implementation of the same micro- benchmark in c code. The compiler is instructed to use all optimizations (using vector instructions, applying loop unrolling etc.)

3.3.2 Micro-benchmarks and an operating system

In this research the CPU is assumed to run an Operating System (OS) to make easy control of the embedded platform possible. The OS contains drivers to control other processing units in the platform e.g. GPU, FPGA,or DSP. Furthermore the OS controls the applications running on the CPU. When trying to reach the maximum attainable performance of a CPU the OS can not be neglected. Properties of an OS like process scheduling and process pre-emption can influence the performance of an algorithm or micro-benchmark.

The scheduler of an OS controls process scheduling and decides which process gets time to execute operations on the CPU. Performance can be influenced when the scheduler decides to move one process to another processing core or even interrupt the process and let it sleep for a while. The embedded platform designer should be aware of this when running a micro-benchmarks or application. In this thesis Linux 2.6.32 is used as the OS. Linux is open source and because of this scheduler policies are transparent and controllable. Three settings of this scheduler may have influence on the measurements of a micro-benchmark.
• Processor affinity, this parameter can bind a process or application to a given processing core. This parameter can be different for every process running on the OS
• Process priority,
• Scheduling policy, the policy of a scheduler decides which process to run next. Examples are Completely Fair Scheduling (CFS) and FIFO policies
. Micro-benchmarks executed for the CPU in this thesis are configured to use the CFS policy. CFS is an implementation of the fair queuing algorithm, CFS uses a concept called "sleeper fairness", which considers sleeping or waiting tasks equivalent to those on the runqueue. The micro-benchmark in this section have a high priority and are bounded to a single CPU.

A micro-benchmark executes operation on data in a buffer. The size of the buffer is fixed an the number of operations executed is also fixed, figure 2.10 illustrates this concept. By measuring time, the performance [operations / second] of this micro-benchmark can be calculated, and combining multiple different micro-benchmarks and their performance then results in a practical Roofline for the CPU.

Measuring time is not trivial on a CPU within an OS. Special time functions such as *gettimeofday* in the operating system are often not accurate enough for performance estimation. Even when it has the accuracy, the calling overhead can not be neglected. A measurement can be done in which the calling overhead of the time function is calculated this overhead can then be subtracted from the measurement. However this is often not accurate enough, in this thesis the performance counters inside a CPU are used by programming and reading these counters via assembler instructions. A designer should think about the implication of calling operating system functions and know that they can introduce unwanted measurement overhead.

3.3.3 Micro-benchmarks using assembly

The calculated theoretical Roofline needs verification to be able to estimate the practical use of the applied method. Eventually the Roofline needs to be used for selecting a CPU when a sample is not yet available, and so this work tries to give an more accurate bound on the estimated performance by measuring the real-life performance of a given CPU.

Roofs and ceilings of the processing unit can only be reached by using all available parallelism in the hardware. Because c/c++ code is translated by the compiler, optimizations may influence the measurement of the micro-benchmark. Therefore more control is needed over the executed instructions to give an accurate measurement of the number of computational and memory operations. To accomplish this assembler programming is used to know which instructions are executed by the processing unit. Listing 3.2 and 3.1 show example assembly for the micro benchmark of $\frac{1}{4}$ operations per byte (32 operations executed on 128 bytes).

Listing 3.1: x86 SSE2 assembly benchmark

```
inline void operation_x86_64(const unsigned char* input, const unsigned char *output)
{
asm volatile
        (
    //4 x 128 bits load into xmm 128 bits registers
    //so the following code loads 4x16=64 bytes in total
        "movdqa (%%rdi), %%xmm8;\n"
        "movdqa 16(%%rdi), %%xmm9;\n"
        "movdqa 32(%%rdi), %%xmm10;\n"
        "movdqa 48(%%rdi), %%xmm11;\n"

        //8 x 4 operations (add) of 32 bits
        "paddd %%xmm8,%%xmm9;\n"
        "paddd %%xmm10,%%xmm11;\n"
        "paddd %%xmm8, %%xmm10;\n"
        "paddd %%xmm8, %%xmm9;\n"
        "paddd %%xmm10,%%xmm11;\n"
        "paddd %%xmm8, %%xmm10;\n"
        "paddd %%xmm9, %%xmm8;\n"
        "paddd %%xmm9, %%xmm11;\n"

    //4 x 16 = 64 byte store
        "movdqa %%xmm8, (%%rsi);\n"
        "movdqa %%xmm9, 16(%%rsi);\n"
        "movdqa %%xmm10, 32(%%rsi);\n"
        "movdqa %%xmm11, 48(%%rsi);\n"
        :
        :"D" (input), "S" (output)
        :"xmm8","xmm9","xmm10","xmm11"
        );
}
```

Listing 3.2: ARM Neon assembly benchmark

```
inline void operation_arm_neon(unint32_t* input, uint32_t* output)
{
asm volatile
    (
        //4 x 128 bits load and increase pointer
        //4 x 16 = 64 bytes in total in order
        "vld1.32 {q0},[%0]!;\n\t"
        "vld1.32 {q1},[%0]!;\n\t"
        "vld1.32 {q2},[%0]!;\n\t"
        "vld1.32 {q3},[%0]!;\n\t"

    //8 x 4 operations(add) of 32 bit
    "vqadd.u32 q4,q0,q1;\n\t"
        "vqadd.u32 q5,q1,q2;\n\t"
        "vqadd.u32 q6,q3,q0;\n\t"
        "vqadd.u32 q7,q2,q3;\n\t"
        "vqadd.u32 q0,q4,q5;\n\t"
        "vqadd.u32 q1,q6,q7;\n\t"
        "vqadd.u32 q2,q4,q6;\n\t"
        "vqadd.u32 q3,q5,q7;\n\t"

        //4 x 16 bytes store and increase pointer
        "vst1.32 {q0},[%1]!;\n\t"
        "vst1.32 {q1},[%1]!;\n\t"
        "vst1.32 {q2},[%1]!;\n\t"
        "vst1.32 {q3},[%1]!;\n\t"
        ://no output
        :"r" (input), "r" (output)
        :"memory", "q0","q1","q2","q3","q4","q5","q6","q7"
        );
}
```

3.4 Intel Atom E6xx

The Intel Atom is a low power CPU which powers a variety of devices. The E6xx embedded variant is based on the tunnel-creek micro architecture. It supports the x86-32 instruction set, and also supports SSE2, SSE3 and SSSE3 vector instruction sets. One important thing to notice is that this embedded variant of the Atom micro-architecture has no support for the x86-64 instruction set. This implies that fewer registers are available for assembler programming. The E6xx is an in-order processor that means it is important to use a compiler that is able to schedule instructions in the right order, because this prevents stalls caused by data dependencies. The E6xx and in general the Atom architecture feature two instruction ports which in turn contain multiple ALUs per port. Port 0 has a mixed functionality supporting computational operations as well as memory access operations. Port 1 only supports computational operations. Figure 3.4 depicts the functionalities within each instruction port. Note that port 0 contains the memory functionality to load and store data to and from the internal memory. When accessing memory with port 0, port 1 is the only function unit that can be used for execution computational operations. Both function units support SSE2 vector

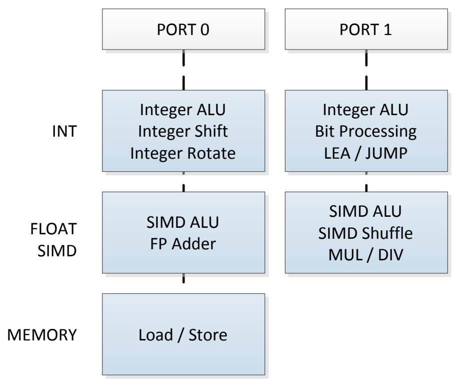


Figure 3.4: Intel Atom tunnel-creek micro-architecture instruction ports. Intruction port 0 and 1 both support integer and vector operations, additionally port 0 can handle floating point additions and port 1 can handle jump conditions. The load store unit is connected to port 0 and while loading and storing no operations can be executed by this port because every instruction port can only handle one type of instruction at the time

An Intel Atom micro-architecture features two instruction ports with multiple ALUs and functionalities. These ALUs support different operations e.g. integer, floating point and vector (SIMD) operations. Note that only one instruction can be execute per port. One port contains a load and store operation, while loading or storing data this port can not be used for executing operations.

instructions, these instructions also give the maximum possible performance for this processor of 8 operations per cycle (e.g. *PADDD* instructions, add four integer numbers per instruction).

The cache memory of the E6xx can be accessed via a *MOV* instruction. $B_c = B_{mi}$ assuming only one core is used by the micro-benchmark . The instruction that moves the most data is a *MOVDQA*, which moves 16 bytes from the memory to an SSE register or vice versa. Load and store functions can not be used in parallel, if data is loaded from no other data can be stored. This *MOVDQA* instruction is executed in one clock cycle. The memory clock an accesses to the cache is then defined by the core clock of the E6xx, that yields $M_f = C_f = 1.3[Ghz]$ for the Intel Atom E630. Applying equation 2.6 results in the following expected internal memory bandwidth when executing the micro-benchmarks on one core.

$$\begin{aligned} M_f &= C_f = 1.3 && [GHz] \\ M_{dr} &= 1 && [Transfers/Cycle] \\ M_w &= 16 && [Bytes] \\ B_c &= B_{mi} = 20.8 && [GByte/sec] \end{aligned}$$

External memory is connected to the Intel Atom via a Double Data Rate (DDR)2 interface The E6xx series supports DDR2 memory with a memory clock of 200 MHz. Every DDR2 transfer contains 8 bytes and so the external memory bandwidth can be calculated using equation 2.6 and the following parameters.

$$
\begin{aligned}
M_f &= 0.2 && [GHz] \\
M_{dr} &= 2 && [Transfers/Cycle] \\
M_w &= 8 && [Bytes] \\
B_{me} &= 3.2 && [GByte/sec]
\end{aligned}
$$

Table 3.1 illustrates the Roofline and ceiling parameters of the Intel Atom E630 processor for memory and computational operations. To verify computational ceilings the performance of the addition instructions is taken into account.

type	C_f $[GHz]$	$C_{\#cores}$	C_{ops} $[Gops]$	$C_{ceiling}$ $[Gop/s]$
SIMD	1.30	1	8	10.4
INT	1.30	1	2	2.6
FLOAT	1.30	1	1	1.3

type	M_f $[GHz]$	M_{dr}	M_w	B $[Gbyte/s]$
$B_c = B_{mi}$	1.3	1	16	20.8
B_{me}	0.2	2	8	3.2

Table 3.1: Roof and Ceilings of the Intel Atom E630

3.4.1 Micro-benchmark results of the Intel Atom E6xx

The following graphs depict the theoretical calculated ceilings and roofs by a red line. Measurements points of the verification are illustrated as marks and the different measurements points are connected by a dashed line to show the measured ceiling and Roofline. Figure 3.5 depicts the floating point, integer and SIMD (vector) performance of the Intel Atom E6xx. The integer and vector performance reach 95% of the theoretical computational performance, while the floating operations performance only 50% of the expected performance. The cause of this can be that x87 instructions, used for the floating point measurement of the benchmark, are stack based this can introduce overhead for pushing and popping variables on and off the stack. Memory performance is worse then expected at most 51% of the theoretical performance is reached by the micro-benchmarks. Most likely the memory controller of the Atom is not performing as expected with the attached memory, or the controller is not configured in the right manner.

The other micro-benchmark tested for different implementation styles, using cache memory, intrinsic function, assembly instructions or plain-c. An assembly implementation is most efficient in reaching the theoretical computational Rooflines, as can be seen in figure 3.6 at 64 ops/byte. The assembly performance reached 96% of the theoretical performance while programming with intrinsics reaches 76% of the attainable performance.

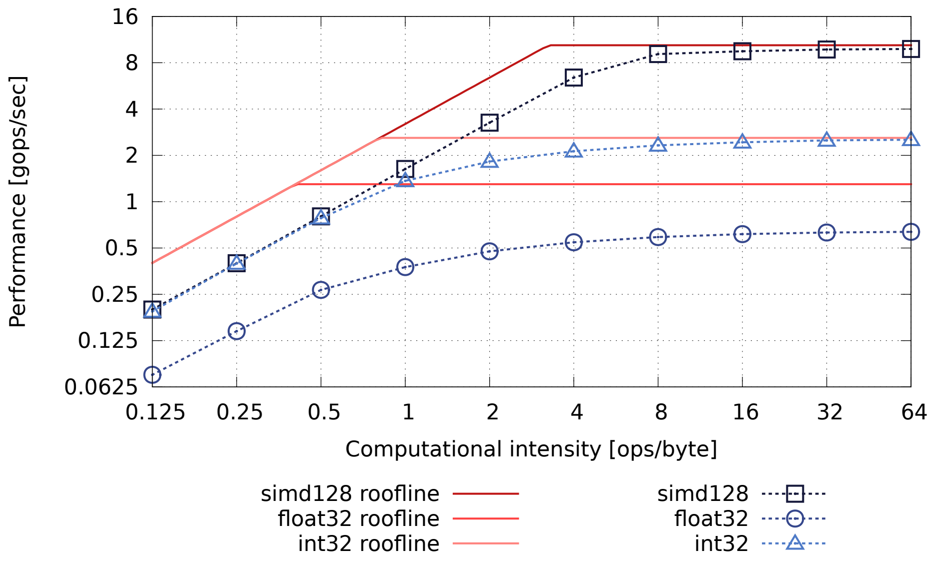


Figure 3.5: Intel E630 benchmark results for integer, single precision float and SIMD

The assembly benchmark reaches at most 71% of the cache memory Roofline and a maximum of 51% of the external memory Roofline. Intrinsics reach the same memory bandwidth but the compiler implementation of the parallel operations result in a slightly worse computational performance. The plain-c implementation can not be optimized by the compiler. Therefore the assembly code generated by the compiler uses additions and calculations which are implemented using non parallel integer computations. It was expected that the compiler could automatically vectorise the c-code because it only contains some constant array indexing.

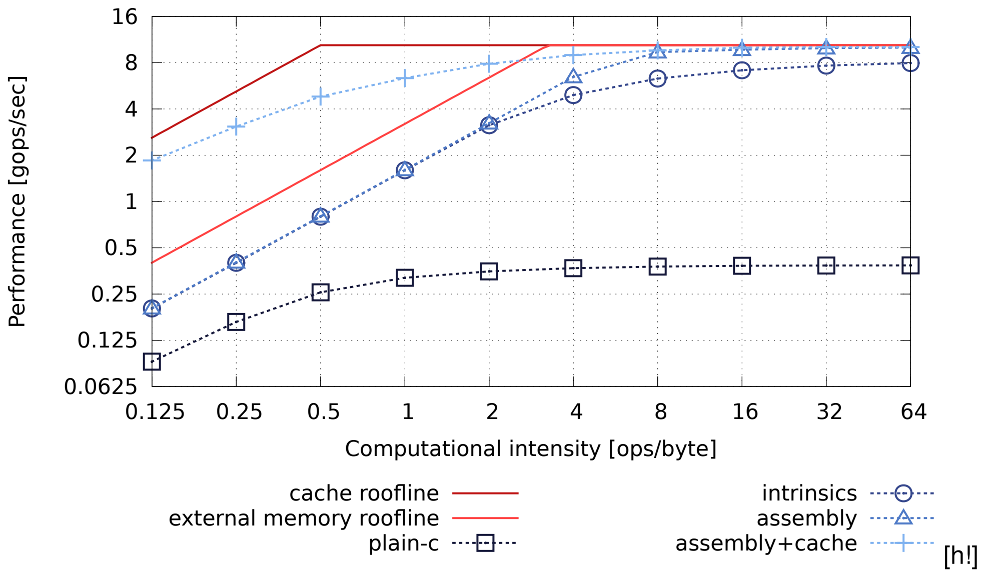


Figure 3.6: Intel E630 benchmark results for intrinsics, assembly and cache

3.5 Intel Xeon E5540

The Intel Xeon E5540 implements the Intel Nehalem micro-architecture. It consists of five instruction ports three are used for computational purposes one functional unit for loading data into registers and two units store data and address information to the memory. Like in the Intel Atom integer, vector (SIMD) and floating point ALUs are shared within an instruction port. Figure 3.7 shows the instruction ports available and the functionalities that correspond to each port. Note that every port can execute at most one instruction every cycle. The most performance (the computational Roofline) is obtained when executing two SIMD additions and an SIMD multiply. A separate instruction port for multiplication is a benefit because these operations often take multiple cycles while an addition needs one cycle at most. The micro-benchmarks in this section implement addition operations to give a equal comparison between the architectures in terms of performance. Multiplications are neglected in this case. This results in 8 operations for vector or SIMD operations per clock cycle and 2 operations per clock cycle for interger operations.

The Xeon E5540 features the Intel turbo-boost technology, this technology increases the clock frequency of a single core when other cores are not used. Only one core was used during the micro-benchmarks, therefore the frequency increased from the default $2.53GHz$ to $2.80GHz$. The memory interface of the Xeon E5540 differs from that of the Intel Atom in the way that it has three memory channels available instead of only one. According to specification in the datasheet of the Xeon E5540 three channels can process upto $25.6[GB/s]$. The micro-benchmark test configuration contains 3 DDR3-1066 Dual In-line Memory Module (DIMM) of 1 GB, one DIMM in every channel, which gives this maximum performance. The performance of the internal memory is comparable to that of the Atom the Xeon 5540 can also load/store 16 bytes in one cycle. However the difference is

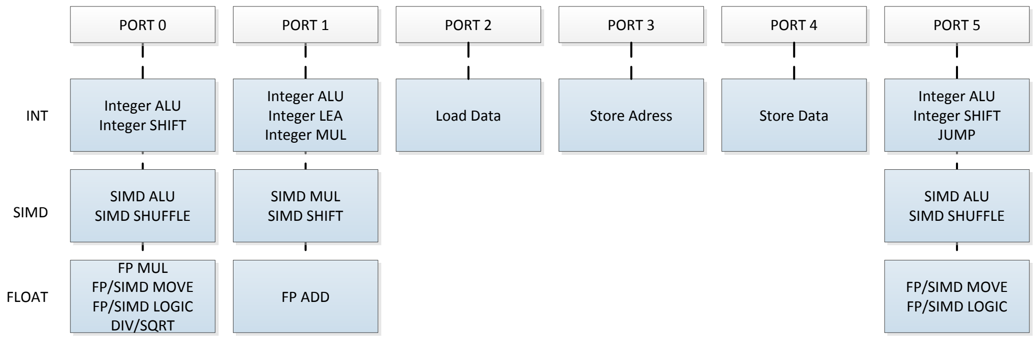


Figure 3.7: The Intel Nehalem micro-architecture features 3 instructions port to handle operations and three separate instruction ports to handle load and store actions to the memory. This allows the Nehalem micro-architecture to execute operations while interfacing with the memory

$type$	C_f $[GHz]$	$C_{\#cores}$	C_{ops}	$C_{ceiling}[Gop/s]$
SIMD	2.8 GHz	1	8	22.40
INT	2.8 GHz	1	3	8.4
FLOAT	2.8 GHz	1	1	2.8

$type$	M_f $[GHz]$	M_{dr}	M_w $[Byte]$	B $[GB/s]$
$B_c = B_{mi}$	2.53	2	16	40.48
B_{me}	0.533	2	8×3	25.6

Table 3.2: Roofs and ceilings of the Intel Xeon E5540

that the Xeon 5540 can perform a load and a store operation in parallel, loading 16 bytes and in the same cycle storing 16 bytes. This is a benefit for application loading and storing a lot of data so micro-benchmarks or applications that are bounded by a memory roof, or do simple operations on a lot of data. This difference is modelled by assuming the data rate of this cache interface to be 2 (two transfers per clock cycle $M_{dr} = 2$).

3.5.1 Micro-benchmark results for the Xeon E5540

From figure 3.8 it becomes clear the the expected triple channel bandwidth of the Intel Xeon E5540 processor can not be reached. Only 55% of the triple channel performance can be harnessed by the micro-benchmarks. Furthermore when executing floating point x87 instructions this drops down to 25% of the theoretical bandwidth. Computational performance can be reached by the integer and floating point micro-benchmarks. They reach about 98% of the available performance. However the SIMD micro-benchmark can only reach 76% of the theoretical performance.

The results in Figure 3.9 are gathered by applying different programming approaches on the CPU. 71% of the theoretical performance can be reached, for the external memory performance nothing

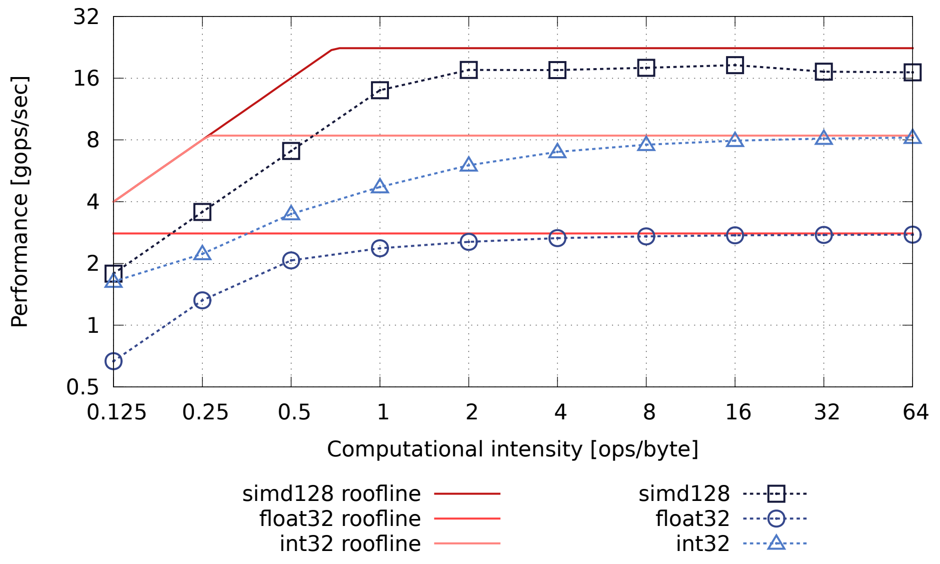


Figure 3.8: Intel Xeon 5540 benchmark results integer, SIMD and single precision floating point measurement

changes (55% of the theoretical performance is reached. Assembler reaches the highest performance 79% of the theoretical performance, using intrinsic functions yields a performance of 71% while plain-c implemented by the compiler only reaches 6% of the performance because it does not vectorize the micro-benchmark code properly.

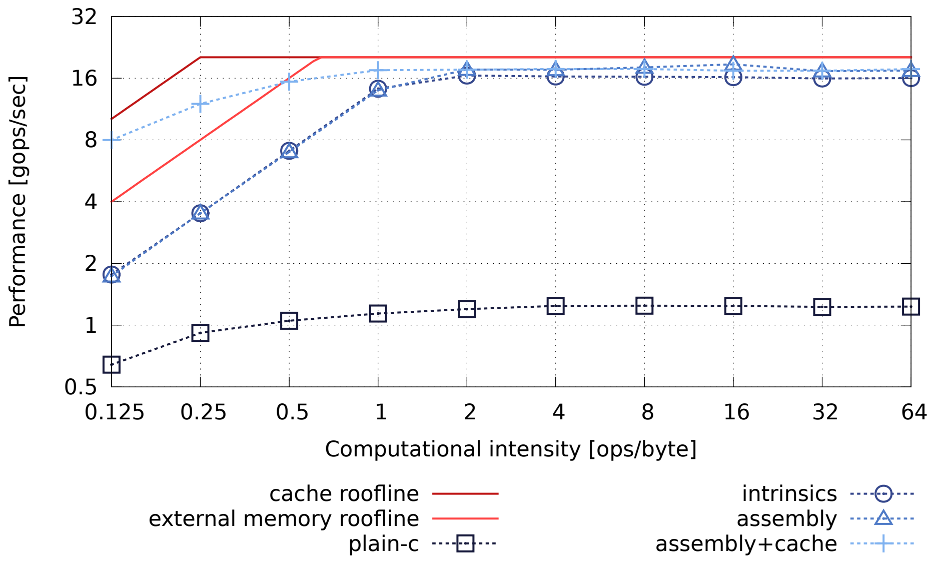


Figure 3.9: Intel Xeon 5540 benchmark results for intrinsics, assembly and cache measurements

3.6 Texas Instruments OMAP4430

The OMAP4xxx is an system on chip device that features a dual core Arm Cortex-A9 processor. Apart from the Cortex-A9 cores it also features a number of dedicated hardware functions e.g audio decoding, video decoding, and an on-chip GPU. The Cortex-A9 core supports VFPv3 floating point and NEON vector instructions. The Cortex-A9 core is an out-of-order processor, which means it can reschedule instructions on the fly. Compiler flags and optimizations are needed to enable VFPv3 and NEON instruction support.

The Cortex A9 features four instruction ports shown in figure 3.10. Two ports for processing integer instructions, one port for handling load and store memory instructions, and one port for handling NEON vector or VFPv3 floating point instructions. Integer instructions can be executed in parallel with NEON vector instructions but NEON and floating point instructions can not be executed in parallel. Using vector (SIMD) instruction in conjunction with integer instructions gives the highest performance on this processor, this is depicted in 3.3.

The OMAP4430 has two memory channels available with support for the LPDDR2 standard. The Pandaboard which is used to verify the prediction on the OMAP4430, has 1GB of LPDDR2-400 on board. The memory clock of this external memory is 200 MHz. Furthermore the highest internal memory bandwidth can be reached when using SIMD instructions. A $vld1.32$ SIMD instruction loads 16 bytes of data in 2 cycles ($M_{dr} = 0.5$). Table 3.3 depicts the resulting internal and external memory bandwidths for the OMAP4430.

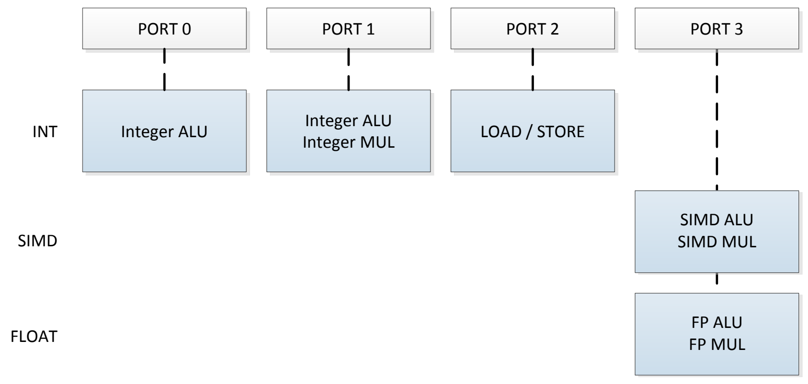


Figure 3.10: The Cortex-A9 micro-architecture inside the OMAP4430 features 4 instruction ports of which two support integer operations and one supports vector or floating point operations. Apart from these ports that operate on data a separate port is available to load or store bytes from or to the memory.

$type$	C_f $[GHz]$	$C_{\#cores}$	C_{ops}	C_{roof} Gop/s
SIMD + INT	1.00	1	6	6
SIMD	1.00	1	4	4
INT	1.00	1	2	2
FLOAT	1.00	1	1	1

$type$	M_f $[GHz]$	M_{dr}	M_w $[Byte]$	M_{bw} $[Gbyte/sec]$
MC_{bw}	1.0	0.5	16	8
ME_{bw}	0.2	2	8	3.2

Table 3.3: Roofs and ceilings of the Cortex-A9 part of the OMAP4430

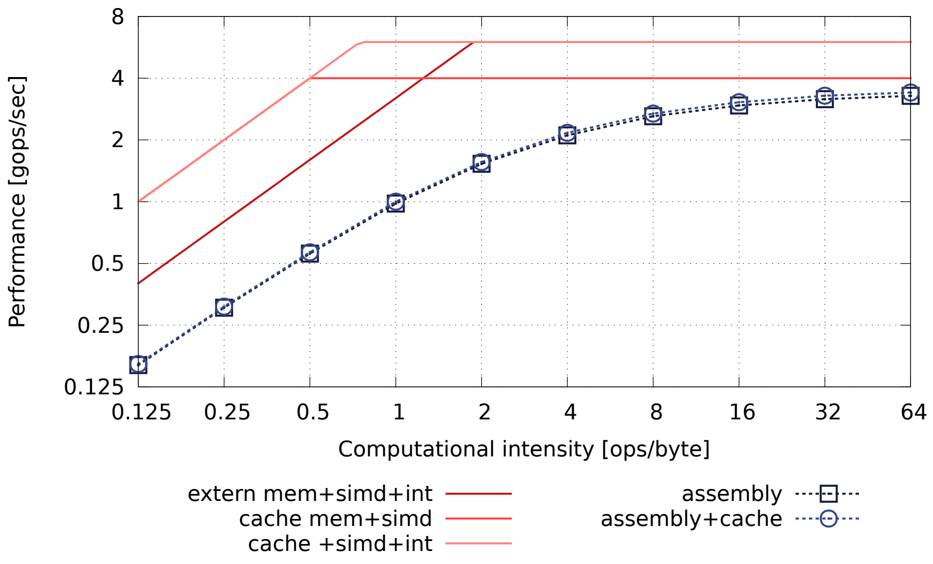


Figure 3.11: Texas Instruments OMAP4430 micro-benchmark results

3.6.1 Micro-benchmark results of the OMAP4430

This section elaborates on the micro-benchmark measurements results of the OMAP4430. Figure 3.11 depicts the measurements done for the Cortex-A9 core inside the OMAP4430 System-On-a-Chip (SoC). The two measurements represent micro-benchmarks that execute vector additions on data from the external memory and from the internal cache memory. External memory bandwidth only reaches 40% of the expected performance while the internal memory performance equals the external memory bandwidth. This implies that the cache memory is not used or is not configured in the correct way.

Because the OMAP4430 features multiple processing units on a single chip (SoC)e.g. the Arm Cortex-A9, GPU, audio decoder, and video decoder. These processing units can all get a memory bandwidth guarantee from the memory controller. In this guarantee lies the memory performance problem of the OMAP4430. The OMAP4430 is optimized for usage in tablet computers and mobile phones in which visual performance is important for the end user. Texas Instruments (TI) states that a fixed amount of the available bandwidth is reserved for the audio and video decoders. This bandwidth limit is also in place when audio and video decoders are not used. Because this limitation skews further measurements the micro-benchmarks for integer and floating point operations where not executed.

3.7 Performance restrictions

This section lists a number of properties that a designer must keep in mind when estimating the performance of a CPU and an application running on it. When not applied properly performance can be much lower then expected.

On the Intel CPUs as well as the OMAP4430 mixing instruction from integer, vector and floating point domain results in a penalty in terms of extra cycles. These penalties are different for every micro-architecture but these extra cycles are required to convert and move values from one domain into another, even going from integer to an integer SIMD domain can introduce penalties

When floating point arithmetic is required on an Intel platform a programmer should try to avoid x87 instructions and use vector (SSE) instructions instead. This is noted by the software optimization guide, *Use Streaming SIMD Extensions unless you need an x87 feature. Most SSE2 arithmetic operations have shorter latency then their X87 counterpart and they eliminate the overhead associated with the management of the X87 register stack.* [18] Using floating point arithmetic together with vector instructions on the OMAP4430 results in a penalty when these instructions are mixed. The penalty in cycles is caused by the use of the same registers for vector and floating point arithmetic. First executing vector instruction and afterwards executing floating point arithmetic resolves this problem. A designer should keep in mind the order in which operations are executed and try to avoid these circumstances by applying operations in a different order. When mixing operations and instructions can not be avoided a designer should know the penalties to more accurate performance estimations.

When accessing data in a program, a programmer must try to use memory bursts, load multiple bytes into the registers of a CPU. This speeds up memory loads on Intel CPUs Atom, Xeon and OMAP4430. A cache controller loads a complete cache line (consisting of multiple bytes). Furthermore the larger the burst length for a DDR controller the better it can reach it's theoretical memory bandwidth. Internal load and store units can often load multiple bytes in one access when one does not make use of this principle bandwidth of the CPU is left unused. A designer should know the underlying or internal organization of the memory for instance the OMAP4430 contains an internal bus which limits accesses to the external memory. When the a designer does not configure this bus in the correct way it can become a bottle neck for the performance of the Cortex-A9 cores contained within the OMAP4430

In CISC based processors and even in the Cortex-A9 translation units are present that can group or translate assembler instructions in more complex instructions or merge given order of instructions into a single internal instructions. This process is difficult to control but a designer must be aware of this when a accurate estimation is desired. Different ISAs have different instructions and register structures, when more registers are available a compiler and designer can make code execute more parallel because dependencies in code or data can be circumvented by having more choice in registers. For example the x86-64 instruction set extends the x86-32 instruction set with 8 extra general purpose registers and 8 extra SSE registers, in case of the ARM ISA the Neon extensions

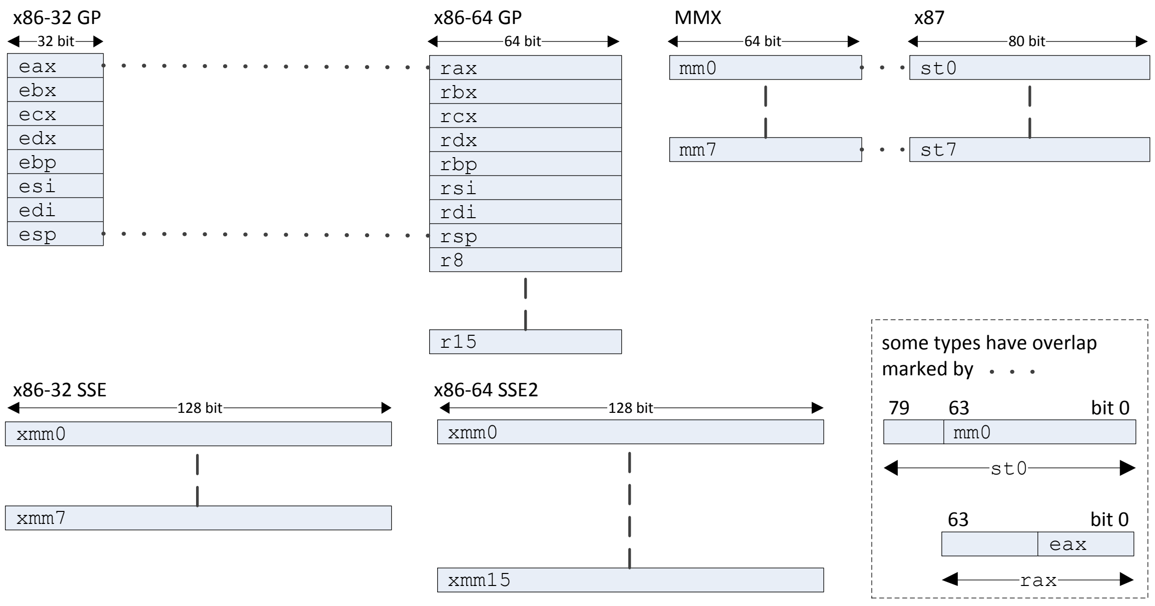


Figure 3.12: Registers of the x86 ISA and of extensions to it such as x87, MMX, SSE

add extra wide q registers to the instruction set. An overview of the differences in a instructions set and extensions is illustrated in figures 3.12 and 3.13.

Automatically applying vectorization code can only be successful when a designer has vectorization in mind when implementing or writing an algorithm. Vectorization can give great benefits and even applying floating point vector instructions can boost floating point performance on Intel CPUs.

Nowadays a CPU is complex and it is often part of a larger SoC. For this many configuration options are available via the BIOS, bootloader or via the OS and drivers. A designer should know about these settings and know how these influence performance, e.g Intel turbo-boost technology, Hardware pre-fetcher, enabling cache or bandwidth prioritization etc.

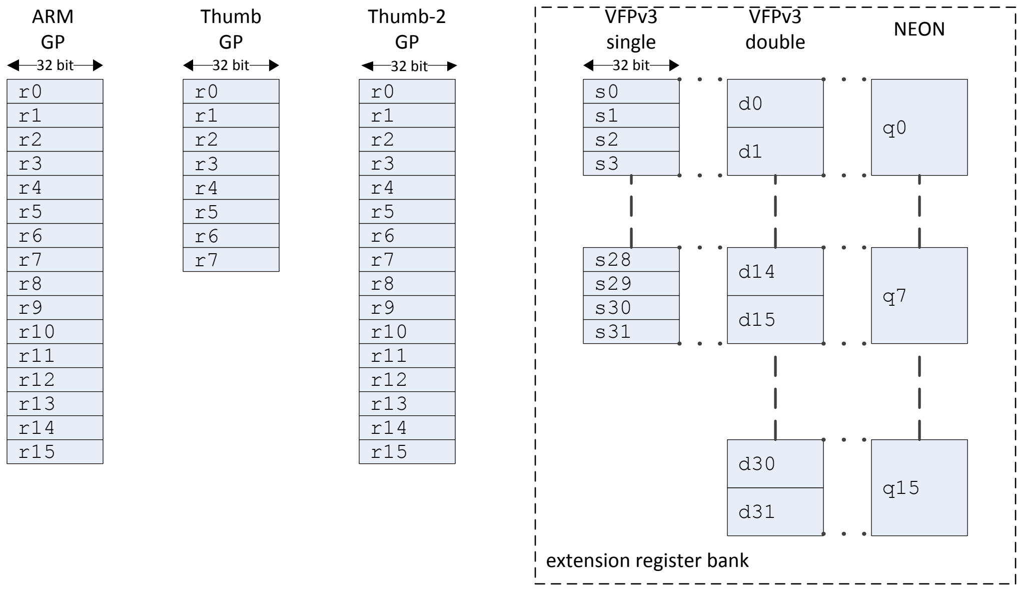


Figure 3.13: Register of the ARM ISA and extensions to it such as NEON and VFPv3

3.8 Conclusions

The extended Roofline model can be applied to an CPU like processing unit. Verifying this using micro-benchmarks shows that the performance of an CPU micro-benchmark can almost reach the performance of the theoretical estimated Roofline. External memory performance is the most difficult to estimate, for the three CPUs under test around 50% of the theoretical performance can be reached for all micro-benchmarks. Internal memory performance can be reached for about 70% of the estimated performance. Assembler programming still beats normal compiler based programming (intrinsics, plain-c) this gives a 7% up to 20% performance boost for the CPUs under test. Plain-c where the compiler should have detected vector operations where not implemented into the final program. When a compiler needs to vectorize code a programmer should keep in mind the design criteria that the compiler requires, this means an implementation must be made in which a compiler can recognize vectorization such as a for-loop that executes operations on an array.

Chapter 4

DSP Roofline model

A Digital Signal Processor (DSP) is a specialized microprocessor with an architecture designed for the fast operational needs of digital signal processing. In general DSP functions are mathematical operations on sampled signals. These operations are repetitive and numerically intensive. Traditional purposes of DSPs are modems, music synthesis and noise cancellation. While important, are nowadays the most DSPs used in the area of wireless communications and internet audio/video purposes [33].

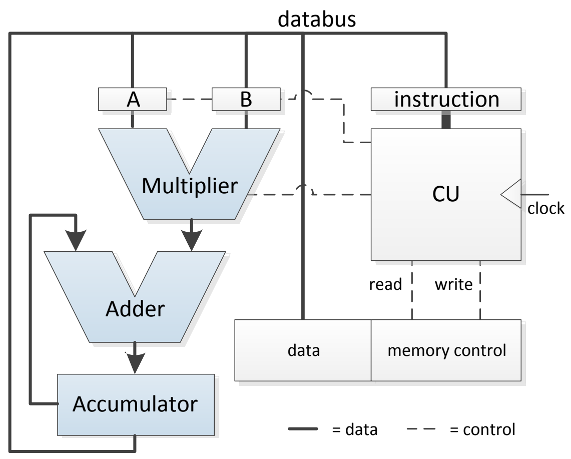
This chapter aims at verifying the Roofline model for DSP processing units. First, an introduction on DSPs is given by discussing the evaluation of DSP architectures in section 4.1. Next the C674x DSP architecture from Texas Instruments (TI) is discussed and used to verify the Roofline model. Section 4.4 concludes this chapter.

4.1 Background

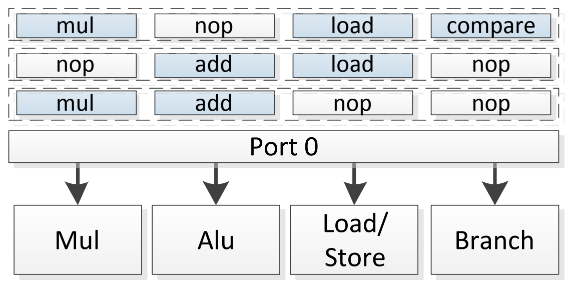
The DSPs which are available today have evolved gradually. However, in the work of Edwin J. Tan [34] four generations of DSPs are distinguished. This background section will give an overview of these four generations and will discus a fifth generation which has emerged in the last few years.

The first DSP processors are released at the end of the 70's. These DSPs were designed around a Harvard architecture with separate data and program buses for the individual data and program memories respectively. The key functional blocks are the multiply, add and accumulator units, but these processors could only perform fixed-point computations. The software that accompanies the chips had specialized instruction sets and addressing modes for DSP with hardware support for software looping. A graphical representation of the general architecture is depicted in 4.1a the light blue parts are complimentary to the general purpose processor shown in Figure 3.2.

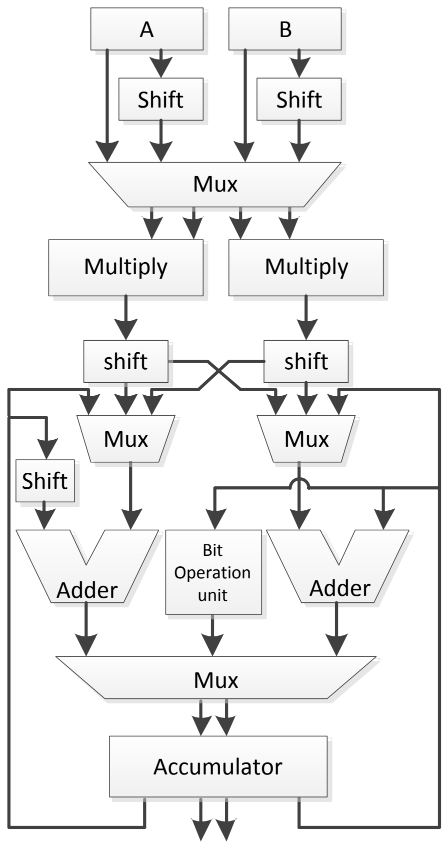
The second generation DSPs are enhanced versions of the first generation, in the sense that they have the same architecture with added features such as pipelining, multiple arithmetic logic units and accumulators. For this generation is a long data-path typical, an example is shown in Figure 4.4. This makes it possible to do multiply or more complex operations in one cycle. A reduced feature size also made it possible to add more peripherals as timers and counters.



(a) First generation conventional DSP architecture, which is a multiply-accumulate extension on the simple cpu from Figure 3.2



(c) Example VLIW architecture, with stuffed NOPs when there a no operations for a certain function unit



(b) simplified datapath of a Lucent technologies DSP16xxx, a second generation DSP

Figure 4.1: Early DSP architectures

In the third generation DSPs designers have incorporated more elements from general purpose processors in order to speed up the computations while at the same time retaining the critical DSP functionality. Single Instruction Multiple Data (SIMD) instructions are an example of general purpose elements in a DSP. SIMD is explained in more detail in section 3.1. Some of the third generation DSPs are based on a Very Long Instruction Word (VLIW) architecture, VLIW processes a fixed number of instructions either as one large instruction or in a fix instruction packet. The scheduling of these instructions is performed by the compiler [15]. This approach can yield an advantage in power consumption by decoding one large instruction instead of several instructions. However, this can cause a disadvantage, when a function unit, e.g., alu, mul. is not used a No Operation (NOP) instruction has to be inserted which has a negative impact on the code density. This is shown in 4.1c. The TMS320C64x family combines both SIMD and VLIW in its architecture. TI tries to increase code density by using execution packets which are fetched per 256 bits. The 256-bit fetch can contain multiple execution packets, which can be 1 to 8 instructions long. All instructions in an execution packet are dispatched together. A bit in each instruction indicates whether that instruction is the last one in its execution packet, this prevents the use of use of nops. It is the programmer's (or compiler's) responsibility to guarantee that all instructions in the execution packet can, indeed, be dispatched simultaneously. The hardware does no dependency checking among instructions.

The fourth generation DSPs are hybrids processors, hybrids are the combination of a DSP and cpu core on one chip. This is caused by the trend that Central Processing Unit (CPU)s have to process more audio and video content. The combination of CPU and DSP saves PCB space which allows for a smaller Printed Circuit Board (PCB) design. The biggest gain is in the power consumption which is an important factor for DSPs since these are often used for mobile devices.

The fifth generation emerged after the work of Tan et al.[34] and are multi- and many-core DSPs. which is enabled by a further decrease in feature size. The work of Karam et al. [20] discusses this generation and shows two trends, the first trend is placing several third generation cores together on one chip. Examples are the TNETV3020 from TI which combines 8 TMS320C64x+ cores and the Freescale 8156 which places 8 sc3850 cores together with a MAPLE-B. The other trend is to place many small DSP cores in a on chip network as is seen in the Tilera Tile64 which has 64 DSP cores and the picochip Pc205 which has 334 DSP cores combined with one ARM9. The survey of [33] states that most of the DSPs used today are in a System-On-a-Chip (SoC) (51%), even for traditional DSP chip vendors like TI and Analog Devices.

4.2 Determining the Roofline for the Omap 137

To verify whether the Roofline model is usable for DSPs, a verification of the model will be performed on real hardware. This verification will be performed on the Omap 137, which is selected because of availability. The Omap 137 is a hybrid DSP and contains an ARM 926EJ-S CPU core and a TMS320C674x DSP core. The C674x DSP core is from the third generation, and can be used as a slave to the CPU. For the Roofline verification is the DSP used in stand-alone mode. Figure 4.2 shows the simplified version of the C674x achitecture. The C674x is a VLIW which uses execution packets to dispatch instructions. The C674x is the superset of the C67x+ floating point and C64x+ fixed point Instruction Set Architecture (ISA). Next to the VLIW architecture the C674x contains an SIMD extension which allows the C674x to do four $8\ bit$ or two $16\ bit$ integer operations. For the verification of the Roofline model the operations are performed on $32\ bit$ data, this limits the number of multiplies per cycle to two single precision or integer. From the ISA it becomes clear that the Arithmetic, Logic and Load/Store units in Figure 4.2 can perform an add operation. Which makes the maximum add operations per cycle 6.

The theoretical Roofline model for the DSP is calculated using Equation 2.15 and the parameters from Table 4.1, where as the practical Roofline is determined by running micro benchmarks. The VLIW architecture makes it difficult to program the C674x in assembly as already stated by [30].

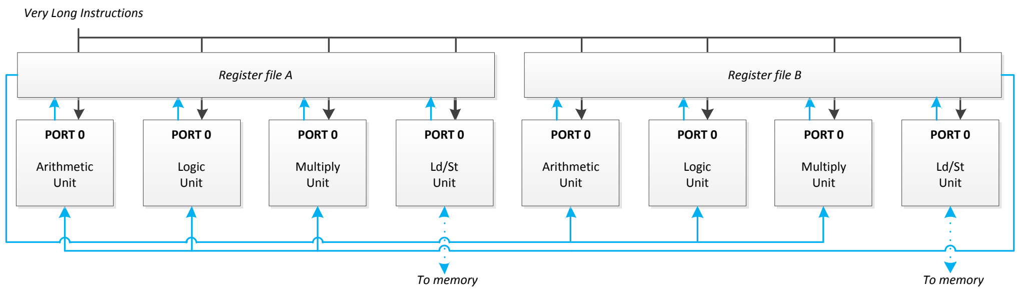


Figure 4.2: TI DSP C647x with an 8-way VLIW architecture, each function unit is labelled with its main function

C_f	$C_{\#cores}$	C_{ops}	M_f	M_{dr}	M_w	M_{roof}	C_{roof}
0.3 Ghz	1	8	0.133 Ghz	1	4 $Byte$	0.532 Gb/s	2.4 $Gops/s$
0.3 Ghz	1	6	0.133 Ghz	1	4 $Byte$	0.532 Gb/s	1.8 $Gops/s$

Table 4.1: Properties of the C674x

Therefore the micro benchmarks are performed using the schedule created by the C6000 compiler from TI at the maximum optimization level (O3). The results of the micro benchmarks are shown in Figure 4.3 using squares. The computational roof is composed of performing 6 additions and 2 multiplies in one cycle, this is also the maximum performance that TI states in the Omap L137 datasheet. However the C6000 failed at creating a schedule which performs these operations in one cycle. The first ceiling is formed by 6 additions per cycle, this ceiling is used to verify the Roofline model.

Figure 4.3 shows the calculated model in solid red, and in dashed blue the performed measurements. The figure shows the maximum performance can not be reached exactly, however the 94% of the computational roof can be reached. The memory roof is reach for 50 %. This is most likely caused by too small read bursts. However, in benchmark results published by TI, it is shown to be possible to reach 94% of the external memory throughput when using the DMA controller [37].

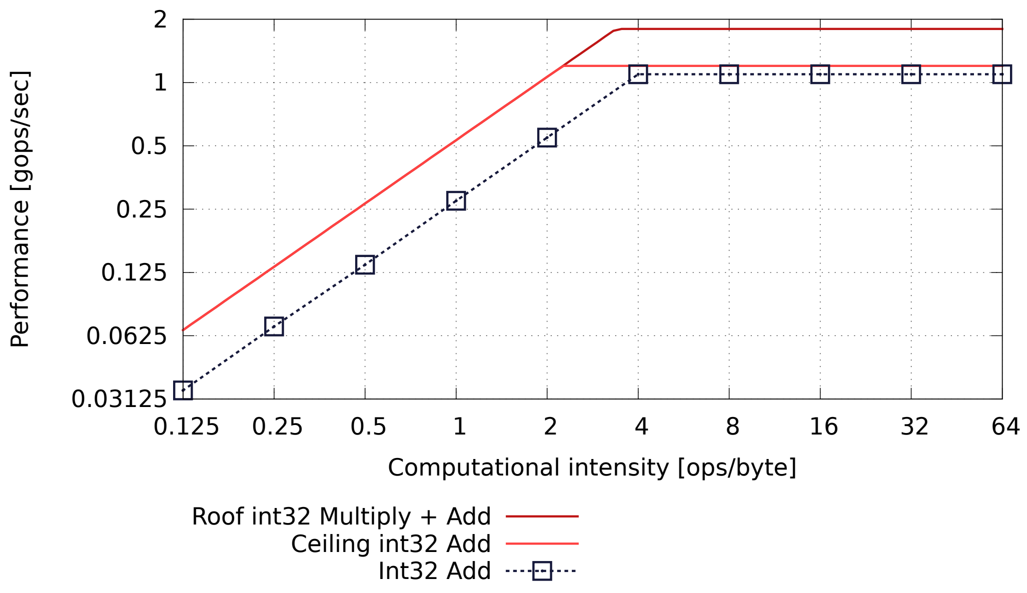


Figure 4.3: The Roofline model of the C674x+ architecture, in the Omap l137

4.3 Additional properties

This section discusses some extra features of the Omap L137 which are not directly visible in the Roofline model but can be important when selection processing units.

The TI C6000 compiler which is used to create the micro benchmarks has the option to form software pipelines. This is a technique used to decrease loop execution times, in a manner that mimics hardware pipelining. Software pipelining is a form of out-of-order execution, except that the reordering is done by a compiler or in the case of hand written assembly code, by the programmer. Without software pipelining, loops are scheduled such that loop iteration i completes before iteration i+1 begins. Software pipelining allows these iterations to be overlapped. Thus, as long as correctness can be preserved, iteration i+1 can start before iteration i finishes. This can permit a higher utilization of the available function units than might be achieved from non-software-pipelined scheduling. In an efficient software-pipelined loop, ii is smaller than s, ii is called the initiation interval this is the number of cycles between starting iteration i and iteration i+1. ii is equivalent to the cycle count for the software-pipelined loop body. s is the number of cycles for the first iteration to complete, or equivalently, the length of a single scheduled iteration of the software-pipelined loop. Figure 4.4 shows the advantage of applying software pipelining. The 674x+ architecture is also equipped whit an software pipeline buffer(SPLOOP), this buffer is used to (re)load the instructions from. This reduces the code size and reduces the overhead when the software pipeline is interrupted.

The C674x core uses a two-level internal memory architecture. Level 1 Data memory is a 32 KB 2-way set associated cache. The Level 2 memory consists of a 256 KB memory space that is shared

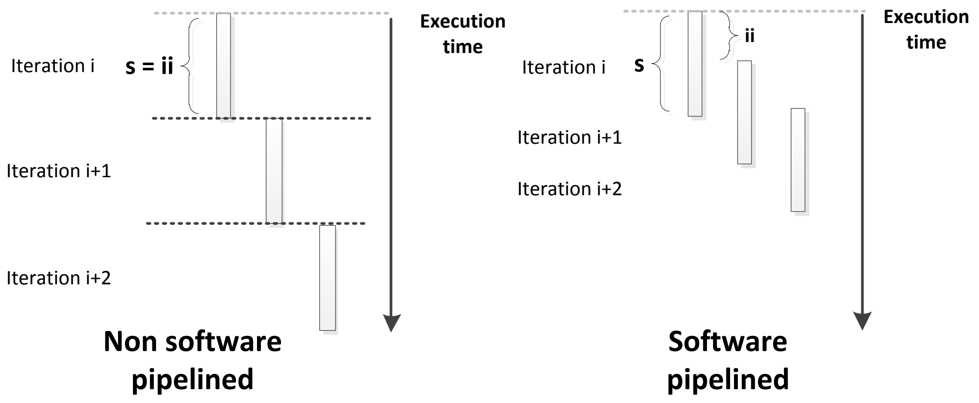


Figure 4.4: In a software-pipelined loop, even though a single loop iteration might take s cycles to complete, a new iteration is initiated every ii cycles. Where ii is equivalent to the cycle count for the software-pipelined loop body. s is the number of cycles for the first iteration to complete, or equivalently, the length of a single scheduled iteration of the software-pipelined loop.

between program and data space. L2 memory can be configured as direct mapped, cache, or a combination of both. The advantage of configuring memory as direct mapped is that one can be sure that the required data is in the in the internal memory. This has the disadvantage that it must be known at design time which data is needed.

A feature that expresses the hybrid nature of the Omap 137 is the Programmable Real-time Unit (PRU), which consist of 32bit RISC processors running at half the DSP frequency. The PRU has a local RAM which is shared between instruction and data and has access to the Direct-Memory Access (DMA) controller, the PRU also has the availability over 30 General Purpose Input-Output (GPIO) pins. The PRU can be used to relieve the DSP from the work caused by communication via an interconnect. E.g. The GPIO pins can be connected to a Ethernet physical transceiver , and the PRU can run the TCP/IP stack the handle the Ethernet communication and use DMA to transfer the data from and to the memory. When using this approach the DSP is decoupled from handling the communication. This means that it has to service fewer interrupts and has more capacity available for signal processing.

4.4 Conclusion

In section 4.2 the Roofline model for the 674x+ is determined and verified, here it becomes clear that the maximum performance can be reached within a small bound (6%). Therefore, it is concluded that the Roofline model is a usable concept to determine the maximum performance. Section 4.3 discusses key aspects of the TI DSP processors which should be taken in to account when this architecture is selected.

Chapter 5

GPU Roofline model

A Graphical Processing Unit (GPU) is a specialized circuit designed to rapidly manipulate and alter graphics in a framebuffer. This framebuffer is intended to output the graphics on a display. GPUs are used in numerous devices e.g. mobile phones, workstations, tablets, and game consoles. Modern GPUs are next to manipulating graphics also able to perform general purpose computations, this is possible since the introduction of programmable shaders.

This chapter presents and verifies the Roofline model for the GPU. Where section 5.1 gives a background on GPUs by discussing several GPU architectures and programming models for GPUs, this section may be redundant for readers who are known with GPUs. The architectural properties which are discussed in this section will be used in section 5.2 to create the Roofline model. The next section discusses the Roofline model for the GPU and shows the measurements used to verify the Roofline model. section 5.4 discusses the circumstance which may restrict the performance of GPUs. Finally, section 5.5 formulates a conclusion of the verification.

5.1 Background

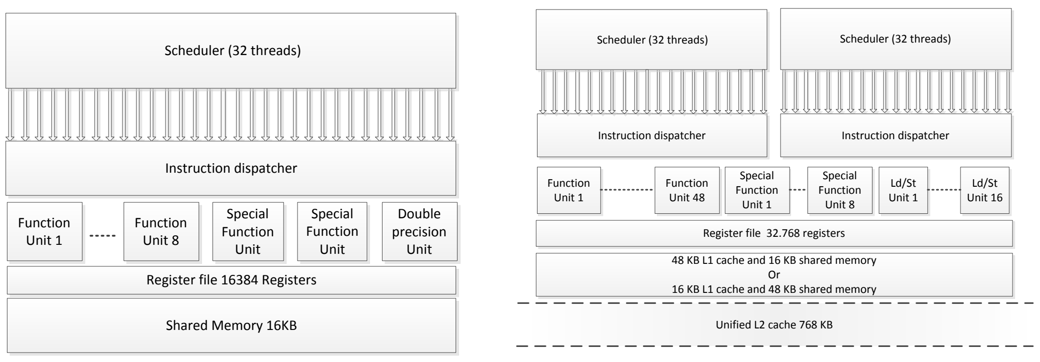
The first GPUs emerged in the 1970s and provided hardware control for mixed graphics and text modes on Atari 8-bit computers. In the 1980s more functions are offloaded to the GPU and the in 1985 released Amiga is the first computer that features what would now be recognize as a full graphics accelerator, offloading practically all video generation functions to hardware.

In the 1990s 2D/3D graphics became increasingly common in computer and console games, which led to an increasing public demand for hardware-accelerated graphics. To interface with the growing number of graphics hardware OpenGL appeared in the early 90s as a professional graphics Application Programming Interface (API). With the advent of the OpenGL API, GPUs added programmable shading to their capabilities. Each pixel can now be processed by a short program that could include additional image textures as inputs, and each geometric vertex could likewise be processed by a short program before it is projected onto the screen. NVIDIA is the first to produce a chip capable of programmable shading, the GeForce 3 in 2002. Since the introduction of programmable shaders researchers are using the GPU to perform general purpose calculations. In 2006 NVIDIA introduced Compute Unified Device Architecture (CUDA) API which is designed for general purpose calculations on NVIDIA GPUs, CUDA is further discussed in subsection 5.1.2.

This section presents three GPU architectures, which are able to do general purpose calculations. The properties from these architectures are used in section 5.2 to detemine the Roofline model. Embedded processors such as TI Omap 4430 and Intel Atom have a graphics processing unit incorporated on the die. These GPUs are not used in this research because there are no tools and drivers available to use these GPUs for general purpose processing. It is expected that within a couple of years, drivers and tools will be available to access and program these GPUs. It is expected that these GPUs will have a similar structure as the examined hardware from NVIDIA and AMD.

5.1.1 GPU Architectures

The Tesla architecture is NVIDIA's first dedicated General Purpose GPU which is released in 2008. GPUs with the Tesla architecture consist of several stream processors, shown in figure 5.1a, each of these stream processors have eight execution units, two special function units and one double precision unit. The number of function units can vary from 32 to 48 and special function units from 4 to 8. The double precision unit handles the 64-bit floating point operations. The special function unit is responsible for executing transcendental functions and mathematical functions such as square root, sine and cosine. The execution units can execute one single precision floating point or 24-bit integer instruction per cycle, a 32-bit instruction is done in 2 cycles. Each execution unit is associated with one load/store unit. One Tesla GPU is composed of several streaming multiprocessors on a die and can be connected via a PCIe link to a Central Processing Unit (CPU).



(a) Stream Processor from the Tesla Architecture, 8 function units, 2 Special function units, 1 Double precision unit

(b) Stream Processor from the Fermi Architecture, 48 function units, 8 Special function units, 16 Load/Store units and L1 cache

Figure 5.1: NVIDIA Architectures

Fermi Architecture

The Fermi architecture is the successor of Tesla architecture, Fermi is released in 2010. The Fermi architecture is shown in figure 5.1b and is targeted to high performance computing as its predecessor. Fermi improves the Tesla architecture on several points, 32-bit integer institutions per cycle instead of 24-bit, fused multiply-accumulate instructions for a higher accuracy and double precision calculations on the execution units, in stead of a separate double precision unit. This enables one double precision calculation every two cycles per execution unit. Instead of one thread scheduler per stream processor there are 2 schedulers which each schedule 32 treads. The Fermi architecture is expanded with a L1 and L2 cache structure and the memory coalescing requirements(5.4.3) are relaxed to enable a better external memory usage. The number of execution units per stream processor are increased from 8 to 48, with a maximum of 16 stream processors per GPU.

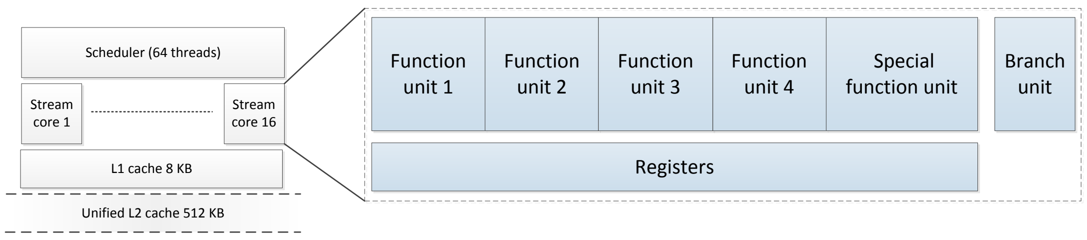


Figure 5.2: AMD VLIW5 architecture

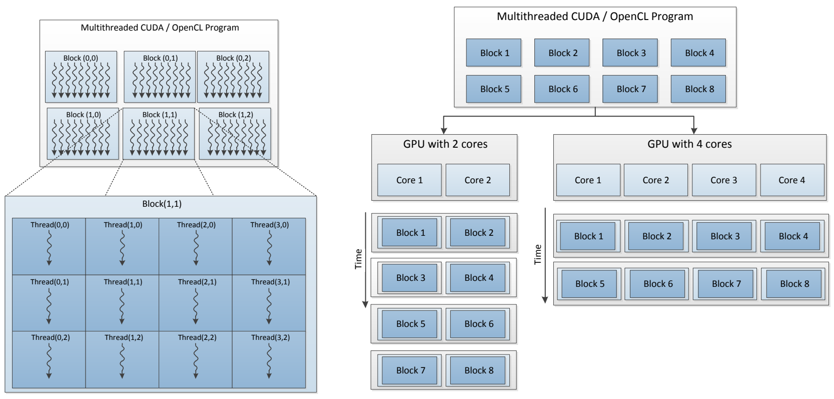
Barts Architecture

The Barts architecture is at a high level similar to the design hierarchy of Fermi. Barts consists of 14 compute units these will be refereed to as stream processors to keep it line with the NVIDIA terminology, each stream processor contains 16 stream cores. AMD GPU architectures are different from NVIDIA architectures in that they try to extract Instruction Level Parallelism (ILP) alongside Thread Level Parallelism (TLP) through a Very Long Instruction Word (VLIW) architecture. Each stream core in Barts contains 4 function units and one special function unit. Each function unit can perform one floating-point operation per clock, while the special function unit can additionally perform transcendental operations. The basic Single Instruction Multiple Threads (SIMT) scheduling block is handled at stream core level and is called a wavefront in AMD terminology, which is nearly analogous to the NVIDIA warp(5.4.2). A wavefront consists of 64 work-items. Since there are 16 stream cores in a compute unit, four work-items are executed on each stream core. One instruction is pipelined from each of the four work-items in the stream core, so that any memory access latency can be hidden by the three other execution cycles. Scheduling of wavefronts is done by the dispatch processor for all compute units. The instruction stream for the kernel is broken up into a series of clauses, depending on the operation. For instance, there are ALU clauses for computations and fetch clauses for memory accesses. The wavefront scheduled changes at every clause change. Each instruction in a clause is a 5-wide VLIW bundle.

5.1.2 Programming model

Modern day multi-core CPUs and many-core GPUs are highly parallel systems. These systems need a programming model which describes the parallelism instead of having to finding the parallelism in a sequential instruction stream. The CUDA and openCL parallel programming models are designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages such as C. CUDA is an NVIDIA specific language while OpenCL is processor and manufacturer independent.

The CUDA programming model is used to construct a kernel. A kernel is the software which runs on the GPU, an example is shown in listing 5.1. The kernel is executed for every thread that is defined at launch time. The threads can be grouped in blocks and each block is assigned to a stream processor as shown in 5.3b, all threads in a block can use the on-chip shared memory and there are mechanisms available to synchronise the threads with in a block. For image operations it is typical to create a thread for every pixel, in matrix operations its typical to create a thread for each element of the output matrix. Each block and each thread within a block has a unique coordinate as shown in 5.3a, these coordinates can be used to determine which data elements should be processed by this particular thread.



(a) Threads are grouped in blocks, which both have their unique coordinates

(b) A block is assigned to a stream processor, but blocks are not always executed in the same order

Figure 5.3: Cuda and OpenCL programming model

5.2 Determining the theoretical Roofline

The theoretical Roofline for the GPU is determined as described in section 2.4.1. The Roofline model is calculated with equation 2.15, which consist of two parts equation 2.6 for the memory bound and 2.1 for the computational bound. The memory part is calculated using 2.6 with a small change because the values of M_w and $M_{\#channels}$ are not known but the product of the two is known.

The computational roof is calculated using equation 2.1, where $C_{\#cores}$ is the number of stream processors times the function units which can be used in parallel in the case of the Tesla architecture 8 per stream processor. C_{ops} is de determined by number of operations which can be done in parallel, in the Tesla case 2 operations one multiply and one addition, this resembles the case where you only perform multiply adds on the and

The first ceiling is calculated by changing the C_{ops} to 1 this resembles the case where a single logic or arithmetic function is performed. The second ceiling is base on performing special functions e.g geometrical or transcendental. The special function units can not operate in parallel with the alu units therefore $C_{\#cores}$ is change to the number of stream processors times the special function units.

5.3 Determining the practical Roofline

To determine whether the theoretical Roofline also holds in practice a number of micro benchmarks are performed. Micro benchmarks are small synthetic programs which in this case consist of 3 parts; load, process, store as shown in figure 2.10. First a data element is loaded from external memory next a number of calculations is perform on this element, e.g. multiplies or adds, afterwards the element is stored in external memory. By varying the number of calculations the operational intensity i.e. operations per byte is varied. In 5.1 an example is given of a multiply micro benchmark which performs "OPS" times a multiplication per data element.

Listing 5.1: Benchmark kernel code

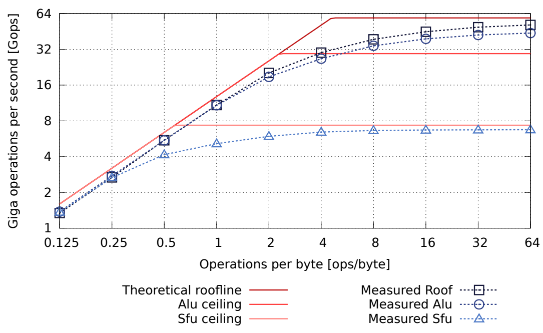
```
__global__ template<unsigned int OPS> Kernel_Multiply(float* g_idata, float* g_odata)
{
  // Calculate thread index
  unsigned int i = blockIdx.y*blockDim.y + threadIdx.y;
  unsigned int j = blockIdx.x*blockDim.x + threadIdx.x;
  unsigned int id = i+(j*blockDim.x);
  float result = g_idata[id];           // Fetch the data

  #pragma unroll
  for (unsigned int i=0; i<OPS ;i++)
  {
    result *= result;                   // Perform the computations
  }
  g_odata[id] = result;                 // Store the result
}
```

	C_f	$C_{\#cores}$	C_{ops}	M_f	M_{dr}	$M_w \times M_{\#channels}$
Roofline	0.92 Ghz	32	2	0.4 Ghz	2	16 $Bytes$
Ceiling 1	0.92 Ghz	32	1	0.4 Ghz	2	16 $Bytes$
Ceiling 2	0.92 Ghz	32	$\frac{1}{4}$	0.4 Ghz	2	16 $Bytes$

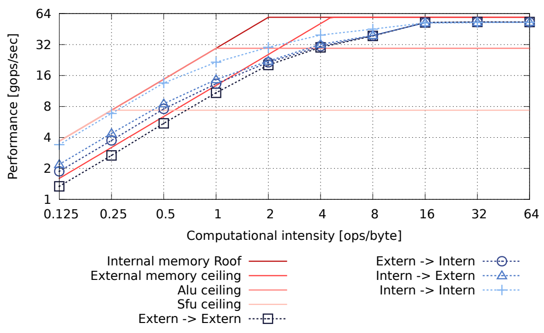
Table 5.1: Properties of the NVIDIA Quadro FX1700

5.3.1 NVIDIA FX1700

The NVIDIA Quadro FX1700 is a midrange GPU targeted at workstations, with a power consumption of max 42W. The FX 1700 has a Tesla architecture with the following properties:



(a) Roofline model with external memory and two computational ceilings, The Alu performance is higher than expected because these operations can also be performed on the Sfu

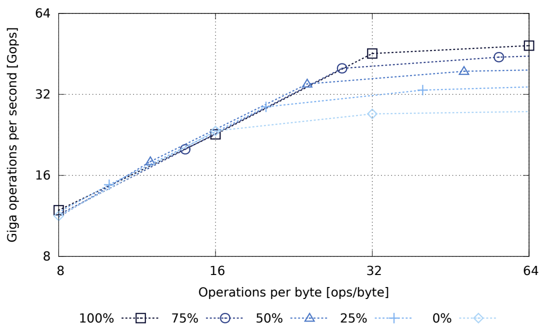


(b) Roofline model with the internal memory as roof and a external memory as a ceiling. The dashed lines show the result of the micro-benchmarks and their source and destination

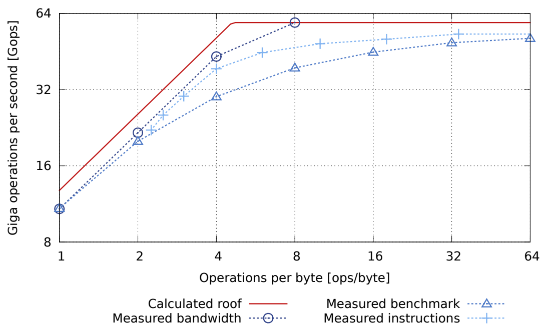
Figure 5.4: The theoretical performance, in solid red, and practical, in dashed blue, of the FX1700

Figure 5.4 displays two Roofline models, both from the FX 1700. Each figure displays a theoretical model using grey lines and a measured model which is display using blue colors. The roof is calculated by performing only Multiply-Accumulate (MAC) the first ceiling is calculated by performing only logic and arithmetic operations. The second ceiling is calculated by performing only transcendental functions. The first figure(5.9a) displays a Roofline model which uses uncoalesced memory accesses, this results in a dramatic drop in memory bandwidth which is show in the roof line model as lower and longer slope. The second Roofline model(5.4a) shows the maximum obtainable memory bandwidth of the FX1700.

The roof of a GPU can only be reached by performing MAC as those count as two operations, the next ceiling is the ceiling where only logic and arithmetic functions are performed. There are not many applications which only use MACs, it is more likely that an applications consist of a mix of MAC and arithmetic calculations. Such an application can reach a performance between the roof and the first ceiling, the actual performance depends on the precise mix and scales linear between the ceilings. An Example of different instructions mixes is shown in figure 5.5a



(a) Non optimal instruction mix



(b) Roofline in 3 steps; theoretical Roofline, theoretical Roofline with measured bandwidth and practical Roofline

Figure 5.5: NVIDIA FX1700 Ceilings

type	C_f	$C_{\#cores}$	C_{ops}	M_f	M_{dr}	$M_{\#channels} \times M_w$
ION roof	1.23 Ghz	16	2	0.667 Ghz	2	8 $Bytes$
ION ceiling 1	1.23 Ghz	16	1	0.667 Ghz	2	8 $Bytes$
ION ceiling 2	1.23 Ghz	16	$\frac{1}{4}$	0.667 Ghz	2	8 $Bytes$
GTX460 roof	1.35 Ghz	336	2	1.8 Ghz	2	24 $Bytes$
GTX460 ceiling 1	1.35 Ghz	336	1	1.8 Ghz	2	24 $Bytes$
GTX460 ceiling 2	1.35 Ghz	336	$\frac{1}{12}$	1.8 Ghz	2	24 $Bytes$

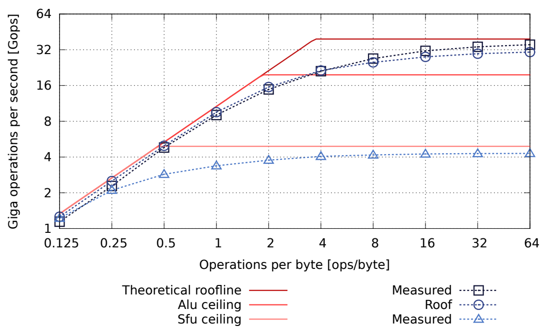
Table 5.2: Properties of the NVIDIA ION and GTX460

From the micro-benchmark results it becomes clear that 94% the theoretical maximum Gops can be reached, as is shown in 5.4a. However, this is only the case when the kernel is computational bounded. When the kernel is memory bounded the performance is somewhere between 70% and 80%. This is due to the extra operations that are created with loops ans address calculations. This is displayed in Figure 5.5b which shows the theoretical Roofline in solid red and the benchmarks in the dashed line with triangles. If instead of the intended operations, all the instructions except loads are taken into account then the performance is higher as shown by the dashed line with crosses. The dashed line marked with circles shows a bandwidth measurement which lines up with the memory bounded results from the micro-benchmarks.

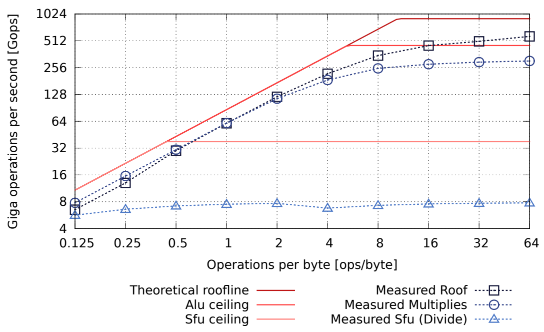
5.3.2 NVIDIA ION and GT460

The NVIDIA ION is a low-end GPU targeted at nettops and netbooks and has a "low" Power usage 14 Watt, the ION is based on a Tesla architecture. The NVIDIA GT460 is a high-end GPU which is targeted at the gamers market and has a maximum power consumption of 160 Watt. The GT 460 is based on a Fermi architecture. The ION and GT460 properties are listed in table 5.2.

The ION reaches 82% of the theoretical memory performance and 96% theoretical computational performance. The GT460 reaches only 65% of the theoretical memory performance and 63% theoretical computational performance. The deviations in memory performance are probably caused by the memory controller, because different memory bandwidth test are performed which all have a performance equal to the performance shown in 5.6b. The computational performance of the GT460 is very low compared the other processing units. This is probably due to the design of the micro-benchmarks, which are designed for the Tesla architecture instead of the Fermi that an more elaborate architecture.



(a) NVIDIA ION



(b) NVIDIA GTX460

Figure 5.6: NVIDIA ION and GTX 460

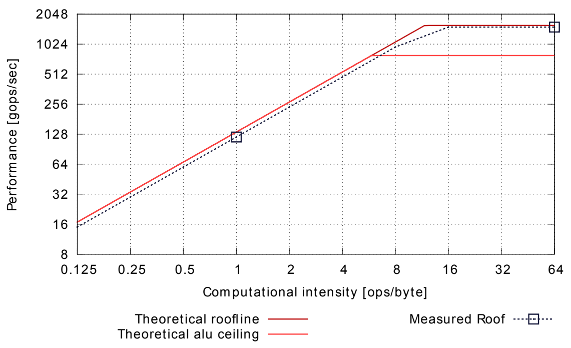
5.3.3 AMD HD6870

The AMD HD6870 is a high-end GPU targeted at workstations, with a power consumption of max 151W. The HD6870 architecture differs from the NVIDIA as discussed in section 5.1.1. The HD6850 consist of 12 SIMD-cores which are comparable to a NVIDIA stream processors. These SIMD-Engines consist of 16 5-way VLIWs which brings the total amount of cores to $14 \cdot 16 \cdot 5 = 960$ execution units.

The Roofline model shown in 5.7b is calculated using table 5.7a and consist of a MAC roof and a Arithmetic Logic Unit (ALU) ceiling. The roof is verified using two micro benchmarks one to verify the memory roof and one to verify the computational roof. These micro benchmarks are created with Compute Abstract Layer (CAL) and OpenCL instead of CUDA as proposed in section 5.3. The memory roof benchmark is created with OpenCL, which is not used for the computational roof because the OpenCL compiler removes some operations during one of the compilation steps. Therefore is CAL used to create a computational benchmark which is a AMD specific programming language. The practical performance shown in figure 5.7b can reach 85% of the memory roof and 94% of the computational roof.

type	C_f	$C_{\#cores}$	C_{ops}
Comp.	0.775 Ghz	14	16 · 5
	M_f	M_{dr}	M_w
Mem.	1.1 Ghz	2	192 Bit

(a) Properties of the AMD HD6870



(b) Roofline of the AMD HD6870

Figure 5.7: AMD HD6870 properties and Roofline

5.4 Performance restrictions

If a application for which a Roofline model is created contains one or more of the listed properties, then should the Roofline model adjusted accordingly. The performance which is shown in the Roofline models above is based on ideal circumstances, in these circumstances the GPU can reach near maximum performance. When the ideal circumstances are not met the performance of the GPU drops. The following list contains examples circumstances which decreace the performance.

- Uncoalesced memory access
- Intra warp diversion
- Minimal amount of threads
- Atomic operations
- Code expansion

5.4.1 Minimal amount of threads

Each kernel that is executed on the GPU consist of a number of blocks which contain a number of threads as explained in subsection 5.1.2. For Tesla processors, each stream processor can accommodate up to 8 blocks or 768 threads. Once a block is assigned to a steam processor, it is further partitioned into warps. Which is a set of 32 consecutive threads, these warps are used to cover the long latency of the function units. The latency of one instruction is 24 cycles and execution of one warp takes 4 cycles this means that there should be atleast $\frac{24}{4} = 6$ warps or $32 \cdot 6 = 192$ threads to cover the read after write dependency. In practise the scheduler needs atleast 8 warps to cover the function unit latency as shown by [41]. To be able to achieve the maximum performance it must be possible to spit GCB in atleast $32 \cdot 8 \cdot \#stream\ processors$ threads.

5.4.2 Intra warp diversion

A warp is an set 32 consecutive threads, which are all executed at once an a streaming processor. If threads of a warp diverge via a data dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjointed code paths. The code example given in Listing 5.2 results in a compare and two moves. First the compare is executed then a move is executed for the threads where the compare yields true thereafter the move is executed for the threads where the compare is false. The impact of this branch is limited, in contrast to the the code example in Listing 5.3 where the instructions f to g will be executed in a serial fashion. If a GCB is constructed as the example in Listing 5.3 the performance of the GPU will drop to the level of a serial execution.

Listing 5.2: Split execution

```
if( input[threadId] < threshold )
   input[threadId] = 1;
else
   input[threadId] = 0;
```

Listing 5.3: Serial execution

```
if( threadId == 0 ) f...g
  elseif( threadId == 1 ) f...g
  elseif( threadId == 2 ) f...g
      ...
  elseif( threadId == 31 ) f...g
```

5.4.3 Uncoalesced memory access

When a warp executes an instruction that accesses the external memory, it coalesces the memory accesses of the threads within the warp into one or more 32, 64 or 128-byte memory transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads. In general, the more transactions are necessary, the more unused words are transferred in addition to the words accessed by the threads, reducing the instruction throughput accordingly. For example, when a 32-byte memory transaction is generated in stead of a 128-byte transaction for each thread is the throughput is divided by 4. 5.9a shows an example of micro-benchmarks which perform uncoalesced memory transfers.

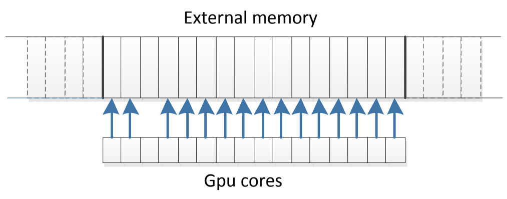
To coalesce, the memory request for a half-warp must meet the following conditions:

- The size of the words accessed by the threads must be 4, 8, or 16 bytes, If this size is:
- 4, all 16 words must lie in the same 64-byte segment,
- 8, all 16 words must lie in the same 128-byte segment,
- 16, the first 8 words must lie in the same 128-byte segment and the last 8 words in the following.
- Threads must access the words in sequence: The kth thread in the half-warp must access the kth word.

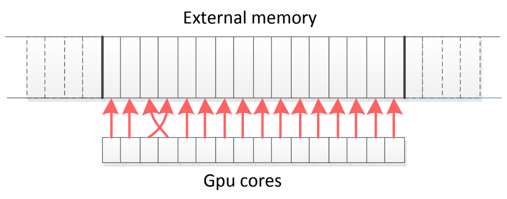
If the half-warp meets these requirements, a 64-byte memory transaction, a 128-byte memory transaction, or two 128-byte memory transactions are issued if the size of the words accessed by the threads is 4, 8, or 16, respectively.

Coalescing is achieved even if the warp is divergent, i.e. there are some inactive threads that do not actually access memory. This is shown in 5.8b

If the memory access pattern of a GCB can not implemented in such a way that it meets the previous mentioned the memory bandwidth will drop dramatic and therefore will not be able to reach the maximum performance in the memory bounded area of the Roofline.



(a) example of a coalesced memory access



(b) example of a uncoalesced memory access

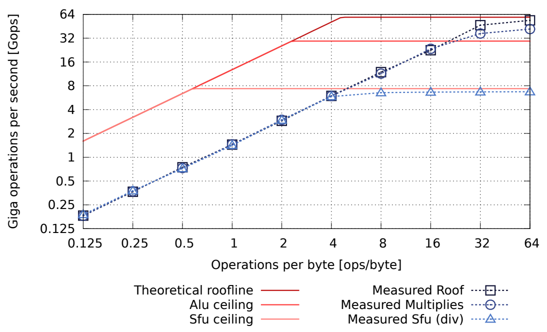
Figure 5.8: One of the important considerations in terms of external memory accesses is memory coalescing. Coalescing is the merged access of 16 sequential positions in external memory (1 position per thread). And also, all 16 words must lie in the same segment of size equal to the memory transaction size.

Atomic operations

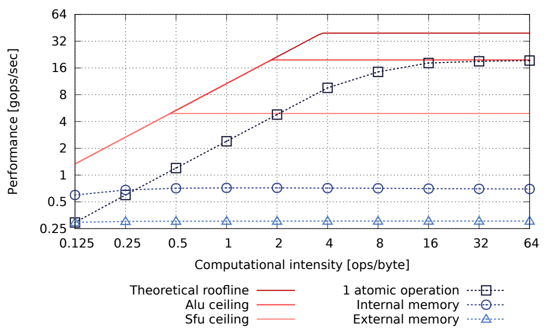
An atomic function performs a read-modify-write atomic operation on a 32-bit or 64-bit word in the external or internal memory. For example, atomicAdd reads a 32-bit word at a address in global or shared memory, adds a number to it, and writes the result back to the same address. The operation is atomic in the sense that it is guaranteed to be performed without interference from other threads. In other words, no other thread can access this address until the operation is complete. This is a very usefull operation to allow multiple threads to work on the same data, the down side is that is is very slow and therefore leads to a huge performance penalty as is shown in 5.9b.

5.4.4 Code expansion

The operations visible in the C are not always the operation executed on the GPU, Listing 5.4 and Listing 5.5 are examples of one C operation which is translated in to 3 GPU instructions. The code expansion is in this case caused by the lack of instructions for 8 bit logic operations. If a certain GCB contains many operations which are expanded to multiple instructions, the performance will be lower than expected using the Roofline.



(a) Calculated and measured Roofline with uncoalesced memory accesses



(b) The red lines show the Roofline model of the FX1700, the dark blue line is a measurement where one atomic operation is added to the micro benchmark

Figure 5.9: Performance restrictions on GPUs

Listing 5.4: C code bitwise and

```
    unsigned char output = input[0] & 0xFF;
```

Listing 5.5: PTX expansion of bitwise and

```
        ld.global.u8    %r28, [%r27+-1];
        mov.u32         %r29, 0;
        setp.eq.u32     %p6, %r28, %r29;
        @%p6 bra        Lt_0_10754;
```

5.5 Conclusion

This chapter aims to verify whether the performance indicated by the Roofline model can be achieved in practise. First number of GPUs are selected of which a theoretical Roofline model is created using the architecture properties of the GPU, the results are shown in section 5.3. From these results it is concluded that the practical computational performance can almost reach the theoretical performance. The computation performance of the micro benchmarks reach on average 95% of the theoretical performance, aside from one exception. The practical bandwidth achieves only 70% to 85% of the theoretical bandwidth. The micro benchmarks performed in this chapter may not be representative all Generic Computational Block (GCB)s, therefore section 5.4 discusses several issues which can retain a GCB from reaching the theoretical limit. These restrictions can be taken into account when creating the Roofline model, which increases the accuracy of model for the GCB.

Chapter 6

FPGA Roofline model

The Roofline model is suited for regular architectures in which the data path is fixed and instructions control how the data is processed. However the Field Programmable Gate Array (FPGA) is an flexible architecture consisting of logical gates that can represent any logic function. The next sections describe how to model this flexible FPGA and construct a Roofline model for it. The Roofline model of an FPGA tries to map the Roofline model in a general way while the example of constructing a Roofline model is done for an Xilinx type Virtex-6 FPGA.

6.1 FPGA basics

An FPGA is regular structure of digital blocks, illustrated in figure 6.1. The definition of an block is different for every FPGA family, however common parts can be identified. Commonly two types of blocks identified within an FPGA, logic blocks and dedicated hardware blocks. Programming an FPGA is done by making connections between the different digital blocks and the available switch matrix. This switch matrix transports data and signals from one block to the next making connections between different digital blocks.

A logic block contains one or more Look Up Table (LUT)s and Flip Flop (FF)s that can be combined to create a logic function. LUTs define a logic function on a number of logic inputs and an FF is used to store the logic result of a LUT. Results from the LUT can be passed to the next Logic Block (LB) via a carry signal or the switch matrix. By connecting a number of logic blocks complex logic functions can be created.

Dedicated hardware blocks are specialised functions that can be used in FPGA designs, these functions need to be fast or consume a vast amount of logic blocks when implemented in FFs and LUTs. Examples of dedicated hardware blocks are, local memory, digital signal processor and multiplier blocks.

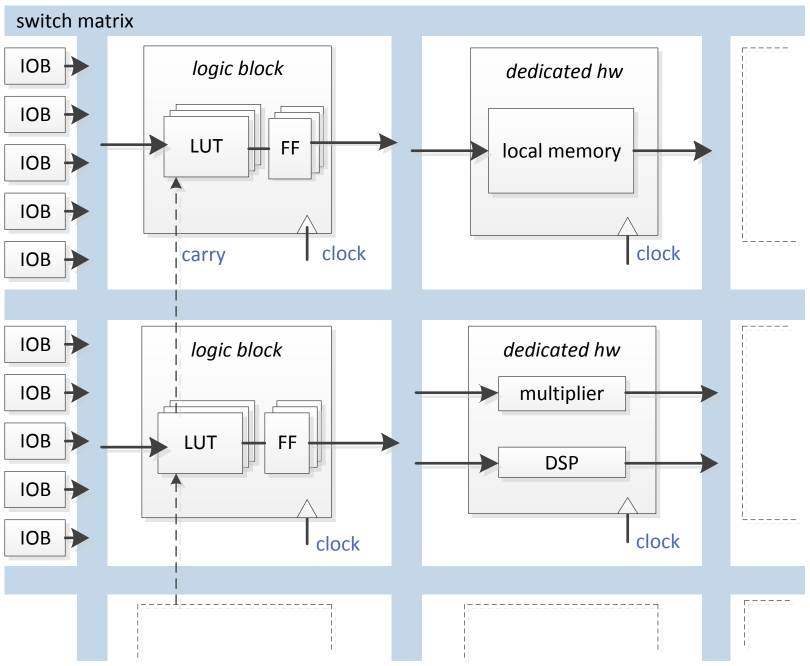


Figure 6.1: The main components inside an FPGA are logic blocks, dedicated hardware blocks, and input-output buffers (IOB). Logic blocks consist of one or more look up tables(LUTS) and flip flops (FF), that are connected together to form a logic function. Functions that are commonly used are implemented in dedicated hardware blocks. Examples of functions implemented in dedicated hardware blocks are, local memory, multipliers, and digital signal processors. All components can be connected together using the switch matrix. The organization of logic blocks, dedicated hardware blocks, and the switch matrix inside an FPGA depends on the FPGA family.

To interface with the rest of an embedded system the FPGA contains in and output pins. Internally these pins are connected to Input-Output Buffer (IOB)s that in turn can be connected to the switch matrix. Depending FPGA family different IOBs exist, some are used for General Purpose Input-Output (GPIO), while others are dedicated clock or high speed interconnect buffers.

6.2 A Roofline for the FPGA

The simplified processing unit (SPU) model illustrated in section 2.8 does not fit the regular digital block structure of the FPGA in which processing cores and function units can not be recognized. However the FPGA can be viewed as a basic processing unit with connections to interconnect, external memory and internal memory, as figure6.2 illustrates.

To define a Roofline model of the FPGA some assumptions are needed about its contents. All these assumptions are based on information from experienced FPGA designers. Figure 6.3 illustrates

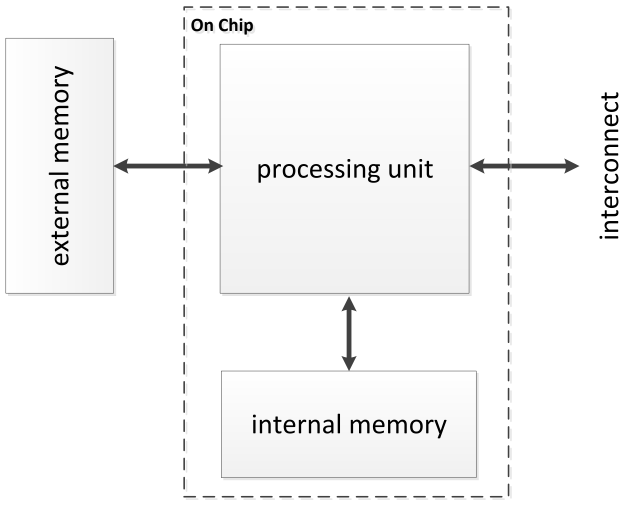


Figure 6.2: The FPGA also fits this basic model in which the processing unit communicates with external memory, internal memory and an interconnect.

which assumptions are done on how FPGA-logic is divided over the different functionalities inside the FPGA. This figure is a reference for the Roofline of the FPGA.

In the first estimation of an FPGA design, designers at Prodrive B.V. assume 20% upto 30% of the FPGA logic is needed by place and route tools. The 30% of unused logic is needed to give the place and route algorithm room to move logic functions in the FPGA, movement of logic can be convenient when timing needs to be improved later on. Timing is a requirement on the design and calculated by the designers e.g. how fast a byte or signal should travel from input to output. Figure 6.4 illustrates the principle, the distance of f1 and f2 is different for the two figures, the shorter the distance, the faster a signal travels from f1 to f2. By reserving space, the place and route algorithms are able to move these functions to improve timing of the complete system.

Communicating with external data sources is done by means of memory controllers and interconnect controllers. It is assumed that these controllers are created using logic inside the FPGA in the absence of dedicated controllers. Furthermore connecting memory or an interconnect to the FPGA is done using GPIO or high speed interconnect pins. However these in- and output pins are limited. Inside the FPGA a pin is connect to an IOB, which in turn can be connected to the switch matrix to transport the input or output signal to a logic function.

The computational power of the FPGA comes form the combination of computational functions instantiated in logic inside the FPGA. From this it is assumed that performance is limited by the available logic inside the FPGA. Moreover the FPGA has dedicated hardware processing blocks like Digital Signal Processor (DSP)s and multipliers that also perform computations on data. These are also taken into account when calculating the maximal attainable performance of the FPGA

In the following subsections Virtex XC6VLX240T FPGA from Xilinx is used to illustrate how a Roofline

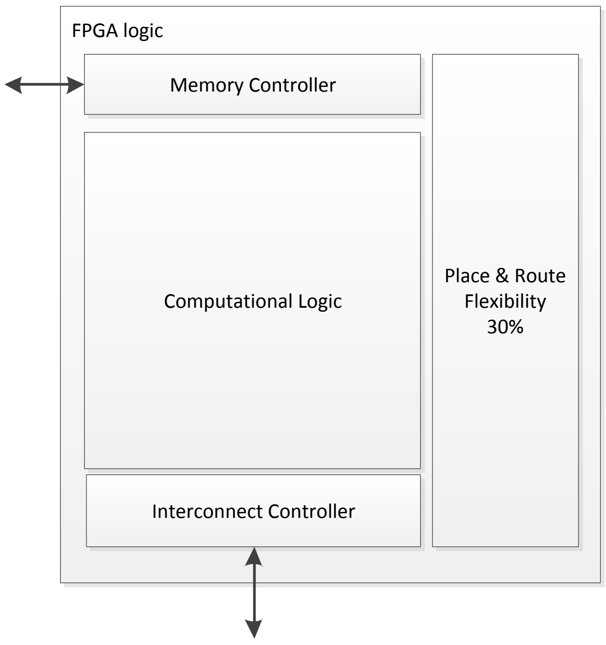
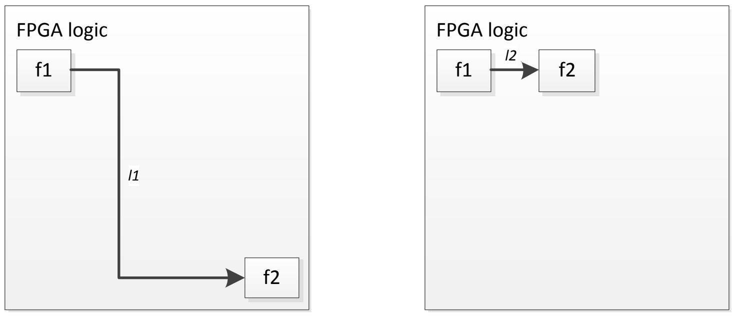


Figure 6.3: Figure of different areas defined in the FPGA in order to estimate the Roofline



(a) Output of logic function f1 needs to travel across the complete FPGA to reach logic function f2. This implies the signal needs more time to travel across the switch matrix

(b) Output of function f1 travels a short distance to reach logic function f2. It needs a short time to travel across the switch matrix.

Figure 6.4: Short and long distance between two logic functions. Place and route tooling tries to meet timing requirements of the designers. Moving logic function to a different location inside the FPGA can improve timing. In figure a,b the length of the path is depicted by $l1$ and $l2$ in which $l1 > l2$, this implies the signal that travels over $l1$ needs more time.

Table 6.1: Xilinx Virtex-6 XC6VLX240T available and required amounts of logic

logic type	available	PCI-e	DDR2
registers (FF)	301440	980	2771
LUTs	150720	1089	1638
BRAM	416	2	8
GPIO	400	N/A	114

model can be created for an FPGA. The XC6VLX240T exists in a number of packages, the kind of packaging defines the number of I/O pins that are available. For this research and to illustrate the calculation of an FPGA Roofline it is assumed that the FPGA is packed in a FF784 package (29mm x 29mm), this results in 12 GTX pins, and 400 I/O pins.

6.2.1 The FPGA memory Roofline

Two types of memory are available inside the FPGA, flip flops contained within the logic blocks and local memory / glsbram contained within the dedicated hardware blocks. In the example depicted below the assumption is made that logic blocks are not used as internal memory, but are solely used for executing computations.

Available memory inside the FPGA is small (14976 Kb of Block RAM (BRAM) is available inside the XC6VLX240T FPGA). However external memory can be attached to the FPGA to store larger amounts of data. To calculate an example theoretical Roofline it is assumed that the FPGA is connected via Peripheral Component Interconnect Express (PCI-e) bus to other processing units and that it has one or more Double Data Rate (DDR)2 channels with controllers to store data into external DDR memory chips. The costs of the peripheral interfaces are listed in table 6.1. This table lists the required block-ram and logic elements of these interfaces, the rest of the FPGA can then be used to created other logic to perform operations on the available data.

DDR2 memory controller

The DDR2 interface connects the FPGA to external memory chips or Dual In-line Memory Module (DIMM) modules. Multiple channels can be created to increase data bandwidth. The number of memory channels that can be created depends on the available amount of external pins that are available on the FPGA A DDR2 memory channel requires 114 Input-Output (IO) pins to communicate with the DDR2 memory modules. The number of pins available on an FPGA depends on the type of packaging and FPGA family. In the case of the XC6VLX240T it has 400 GPIO pins available. This illustrates only 3 DDR controllers can be instantiated because 3 controllers need 342 GPIO pins. Three DDR2 controllers require $2771 \times 3 = 8313$ registers, $1638 * 3 = 4914$ LUTs, and $3 * 8 = 24$ BRAM. A single DDR2 controller running at 200 MHz can reach about 75% of its theoretical bandwidth is caused by the need for synchronization and refreshing of the memory. This bandwidth is 200 MHz * 2 * 8 bytes which yields 3.2 GB/s. To reach 75% of the bandwidth a burst of 64 cycles is required, increasing the burst size increases this percentage. Burst of 64 cycles, times 2, times 64 bit (up going edge down going edge) gives 8192 bits of data per 64 cycles. This requires buffering to make sure enough data can be transferred, buffering in the memory controller is assumed to require 8 block rams per controller.

PCI-e interface

It is assumed that the FPGA communicates to other processing units like a Central Processing Unit (CPU) via the PCI-e interconnect. This interconnect uses dedicated PCI-e pins to connect the FPGA to the PCI-e lanes. The design of the PCI-e interface supports 1 up to 4 lanes. This implies a bandwidth of 500 MB/s upto 2 GB/s. Dedicated GTX pins are used for this purpose. Moreover only one PCI-e controller can be connected because of the limited amount of pins available in the XC6VLX240T FPGA.

Local memory or Blockram

Blockrams (BRAM) are used to store results processed by the computational logic, analogue with the cache inside a CPU, and the internal memory definition of the SPU model. In the Virtex-6 FPGA considered in this example blockrams are organized in blocks of 36Kb that if necessary can be split into two blocks of 18Kb. According to the IPcore tooling an dual port blockram (no ECC) with a width of 32 bit and a dept of 1K can be clocked at 386 MHz and uses one 36Kb blockram. The available number of block rams inside an XC6VLX240T FPGA are 416, and 24 of them are used for buffering the DDR2 controllers. $416 - 24 = 392$ blockrams can then be used for internal memory purposes. Assume the block rams are configured as Simple Dual Port Blockram (Read or Write but not both) 1K times 32 bit uses one blockram of 36Kb, and assume a clock frequency of 150 MHz is used for driving these blockrams. The internal bandwidth can then be calculated using this data shown in equation 6.1

$$B_{mi} = 392 \cdot 150MHz \cdot (32bit/8) = 26GB/s \tag{6.1}$$

Equation 6.2 depicts the internal memory bandwidth when all blockrams can be used at their maximum frequency that is stated in the datasheet.

$$B_{mi} = 392 \cdot 386MHz \cdot (32bit/8) = 70GB/s \tag{6.2}$$

6.3 The FPGA computational Roofline

The computational performance inside an FPGA is mainly harnessed by the available logic blocks and available dedicated hardware blocks. How these are organized is different for every FPGA family.

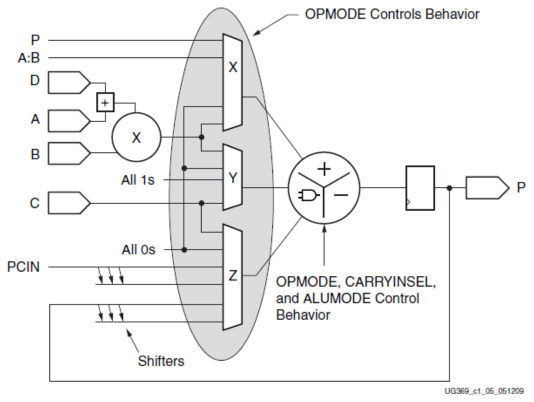


Figure 6.5: Basic functionality of DSP48E1 dedicated hardware block inside the Virtex-6 FPGA, as illustrated in the datasheet Xilinx UG369

This section illustrates an example calculation of an FPGA computational Roofline for the Xilinx XC6VLX240T FPGA. Before introducing the computational Roofline equations knowledge about the computational blocks is required.

6.3.1 Dedicated hardware block - DSP48E1 slice

The DSP48E1 slice contains an 25x18 bit multiplier with registers (A,B,C,D,P) and control logic to select the functionality of the DSP48E1 slice. Functionalities that are present are, multiplication, multiply accumulate, addition, substraction pattern recognition etc. In this example it is assumed that DSP48E1 blocks are used to implement multiplications inside an FPGA. Figure 6.5 illustrates the basic components of such a DSP slice.

6.3.2 Logic block - Configurable logic block

Configurable Logic Block (CLB) is the Xilinx name for a logic block. In the Virtex-6, a CLB contains two slices. A slice contains Four LUTs and eight flipflops. The look-up tables (LUTs) in Virtex-6 FPGAs can be configured as either 6-input LUTs (64-bit ROMs) with one output, or as two 5-input LUTs (32-bit ROMs) with separate outputs but common addresses or logic inputs. Each LUT output can optionally be stored in a flipflop. Four flipflops per slice (one per LUT) can optionally be configured as latches. In that case, the remaining four flip-flops in the slice must remain unused.

With the previous knowledge, ceilings can be identified depending on the type of operation. In this example it assumed that the clock frequency of the total system can be tuned to run at a 150 MHz. According to the FPGA designers at Prodrive B.V. this is a reasonable value to use for this kind of design. But the attainable clock speed depends on the timing of the total system and in worst case situations this can be much lower. However the Roofline model only states an upper bound on this performance and for this purpose this assumption is valid.

From the previous memory Roofline calculation that logic is reserved for the PCI-e and DDR controllers. The following calculations can be done to give a first estimation of the computational roof of the example FPGA Virtex-6 XC6VLX240T. 6.1 depicts the usage of the PCI-e and DDR2 cores in terms of logic and BRAMs.

Total LUTs Available	150720
Routing flexibility 30%	-45216
DDR2 core 3x	-4914
PCIe core 1x	-1089
Computational LUTs	99501

Table 6.2: Number of LUTs available for computational purposes

Total FFs Available	301440
Routing flexibility 30%	-90432
DDR2 core 3x	-8313
PCIe core 1x	-980
Computational FFs	201715

Table 6.3: Number of FFs available for computational purposes

Before calculating the Roofline, a designer needs to determine the number of adders / multipliers that can be implemented inside the FPGA for this the resource usage of these functions inside the FPGA is estimated. Estimation of resources is done using Xilinx ISE Core generator tooling. This tooling shows the resource usage of one 32 bit addition and multiplication for a given Xilinx FPGA. Two possible configurations are selected one speed optimized and one area (resource usage) optimized. Using these numbers the maximum number of operations for a given configuration can be calculated as illustrated in table 6.4.

	Available	Mult1	Mult2	Add1	Add2
LUT	99501	0	614	0	32
FF	201715	0	0	0	32
DSP	768	4	1	1	0
	Max amount $\rightarrow$	192	162	768	3109

Table 6.4: FPGA resource usage for two operations multiply and add with two implementation options for each of them. These numbers are based on 32 bit wide operations

Assume 3109 additions (ADD2) in logic + 768 additions in DSP(ADD1) gives 3877 addition operations. These 3877 additions are executed at $150MHz$ and this results in 581 Gops/sec. Additions are the computational Roofline of the FPGA.

A ceiling for multiplications is 162 Multiplies (MULT2) + (768-162)/4 which gives 151 Multiplies of type MULT1 and this yields 313 Multiplies in total. 313 multiplications at $150MHz$ results in 46 Gops/sec .

These calculations in combination with the calculations for the memory bandwidth Roofline result in figure 6.6, which illustrates a combination of different ADD,MULT,DDR2,BRAM, and PCI-e ceilings.

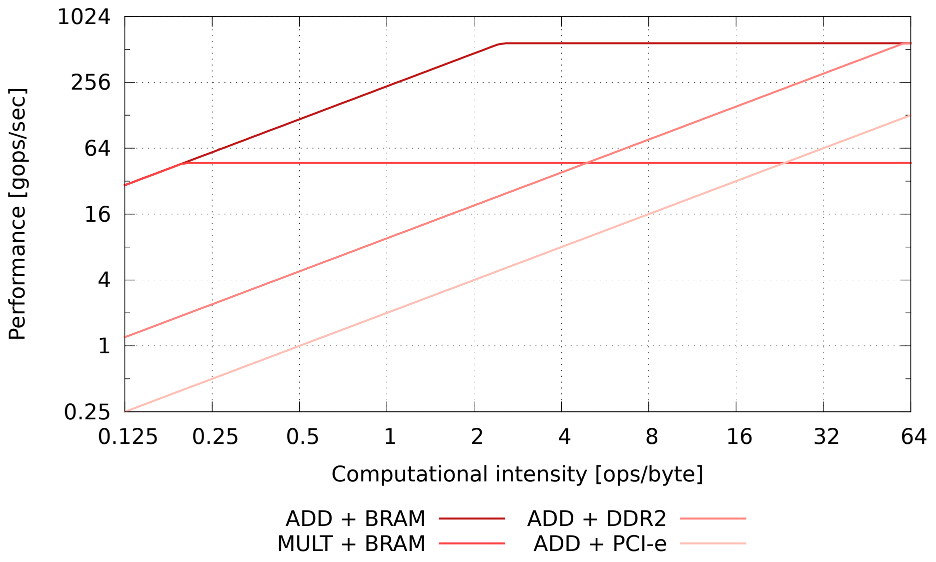


Figure 6.6: Resulting Roofline after estimation of computational performance and memory bandwidth

The previous example calculations and Roofline can be linked to terms of the SPU model while the layout of the model itself does not apply (the FPGA does not contain processing cores). Performance and size of memory and interconnect depends on the available resources, local memory blocks, logic blocks and input output pins.

$$RM_x = \text{Required amount of resource x for memory controller} \tag{6.3}$$

$$RA_y = \text{Available amount of resource y} \tag{6.4}$$

$$M_{\#channels} = MIN\left(\frac{RA_{logic}}{RM_{logic}}, \frac{RA_{pins}}{RM_{pins}}, \frac{RA_{localmem}}{RM_{localmem}}\right) \tag{6.5}$$

Equation 2.6 can then be used in conjunction with $M_{\#channels}$ to calculate the external memory bandwidth B_{me} of the FPGA. In this equation $M_{dr} = 2$ and $M_w = 8$ because a DDR2 memory is used Interconnect is defined in the same manner as normal memory, equation 6.5 is used to calculated the number of channels. The interconnect controller architecture defines the Memory width (M_w), datarate $(M_{dr})$and interconnect frequency (M_f).

Local memory can be calculated in the same manner. However local memory can not be used without considering the core generator tool that is used to implement local memory. The core generator tool creates a memory core which consists of a configuration of one or more local memory blocks. From this configuration which is defined by M_w and M_{depth} the number of local memory blocks per memory core can be calculated. When the amount of local memory blocks per memory core is known, the total amount of local memory blocks can be divided by the number of available local memory blocks to get the number of memory channels. Equation Equation 2.6 can then be used to calculate the internal memory bandwidth B_{mi} of the FPGA. In this equation M_{dr} and M_w are defined by the chosen implementation of local memory.

$$M_{depth} = \text{number of data elements stored in the memory core} \tag{6.6}$$
$$M_w = \text{size in bytes of data elements stored in the memory core} \tag{6.7}$$
$$RM_{localmem} = config\left(M_{depth}, M_{width}\right) \tag{6.8}$$
$$M_{\#channels} = \frac{RA_{localmem}}{RM_{localmem}} \tag{6.9}$$
$$\tag{6.10}$$

Computational performance is also limited by the resources inside the FPGA it is assumed that computations are implemented using dedicated dsp blocks, flip flops and logic. The number of operations of type i that can be implemented inside the FPGA can be calculated as depicted in equation 6.13. When multiple type of operations are implemented one should know that the available amount of resources decreases for every operations that is implemented. Resources are shared over multiple operations.

$$RO_x = \text{required amount of resources for a given type of operation} \tag{6.11}$$
$$RA_y = \text{available amount of resource y} \tag{6.12}$$
$$C^i_{ops} = MIN\left(\frac{RA_{luts}}{RO_{luts}}, \frac{RA_{ff}}{RO_{ff}}, \frac{RA_{dsp}}{RO_{dsp}}\right) \tag{6.13}$$

Available resources ($RA_{...}$ are the main limitation in the FPGA, note that a designer chooses which controller (memory, interconnect) or operation gets the most resources. For every function in the FPGA $RA_{...}$ can be a different amount. Favouring the fastest memory controller can result in a lot of used logic, while implementing multiple smaller (in terms of logic) controllers results in almost the same performance and less logic, the remaining logic can then be used to implement more operations.

6.4 Determining the practical Roofline

The theoretical model as illustrated in the previous sections illustrates on upper bound on the performance of an FPGA, the accuracy of this approach can be improved by doing micro-benchmarks. However implementing a complete design is a subject for future studies. Based on existing designs, performance and computational intensity can be estimated for a Generic Computational Block (GCB). This can then be used to plot an estimate in the theoretical Roofline.

6.5 Performance restrictions

FPGA designs implemented in an FPGA are often limited by the amount of BRAM that is available, especially for image processing. Image processing uses line buffers that store part of the image for processing. The larger the image the larger the line buffers the are required.

Resource usage in this chapter is based on the numbers generated by Xilinx IPcore generator tooling. This only gives an indication of the used resources but does not always correspond to the real implementation or the resource usage after place and route. To be more accurate a designer can use knowledge from previous FPGA designs to get a more accurate estimation on performance and resource usage.

In this chapter the theoretical Roofline assumes that all computations are executed in parallel. A designer should be causes when assuming this in a real life design,a better estimation can be made when using the architectural design or block diagram to estimate the performance of different blocks inside the architecture.

In an FPGA more smaller buses at higher clock frequency are preferred over wide (example 256 bit) buses at lower clock speed this to make routing possible. A designer should be careful when implementing control flow in a FPGA this can result in larger designs and often a streaming data approach is better way to implement a design. Control flow can then be handled by an soft core processor.

6.6 Conclusions

In this chapter a example is given of how a Roofline for the FPGA can be constructed. A theoretical model can be constructed and is usable to plot GCBs into it. However because of the lack of knowledge about real FPGA designs the Roofline model is not verified yet. This can be part of future research.

Chapter 7

The decomposition of VCA applications

This chapter discusses how applications can be placed in the Roofline model. The first two sections give a detailed explanation of step one and two from the approach presented in chapter 2. Step one is discussed in section 7.1 and section 7.2 discusses step two and provides a simple example. Section 7.4 presents a use case, in which an application decomposed in to multiple GCBs. These GCBs are described using the properties presented in section 7.2. Section 7.5 and section 7.6 discuss the effect of multiple GCBs on one processing unit. This section continues with an introduction to Video Content Analysis (VCA).

Video Content Analysis (VCA) is the automatic analysis of video in order to detect objects and determine temporal events in a stream of images. VCA is used in many domains to analyse and detect objects or events. Example domains are health care, automotive, traffic, and security. Within these domains there are different applications, e.g., in the security domain smart cameras are used to detected faces and compare them with a database of known suspects. An example from the health-care domain is the use of VCA to estimate the sleeping pose of a patient.

VCA is not only used in industrial products like smart cameras but also in the consumer electronic and automotive industry (mobile phones, cars). An example of VCA on a mobile phone is augmented reality which adds information or objects to a live video stream recorded by the camera of a mobile phone. Within the automotive industry VCA is used to detect traffic signs at the side of the road, the Audi A8, BWM 7-series and Opel Insignia already incorporate this technology. VCA is an active research area where many companies and scientist are continues improving and designing new algorithms, as described in the survey of [17].

A VCA application needs a computational platform to perform the video analysis, This computational platform processes the video stream acquired from one or more cameras and uses the algorithms contained in the VCA application to filter the required information from the video stream. The cameras that exist today, can produce images of over 15 million pixels, processing these images at an acceptable frame rate(Frames Per Second (FPS)) requires substantial computational capacity.

7.1 VCA applications

From the prestudy [31] it is known that many different VCA domains exist e.g health care, traffic, security, entertainment. Investigating all the different domains would consume to much time, therefore the traffic and security domain is chosen for further investigation. This domain is an active research area as is shown in the survey of [21]. In the pre-study [31] a number of applications and related papers from the traffic and security domain are investigated to find the de facto standard algorithms. However, only Mixture of Gaussians is found as the de facto standard for background modelling. Despite being unable to find many de facto standard algorithms presents the prestudy a common flow which is found in most security and traffic domains. This flow is listed below:

- Background modelling
- Morphological filtering
- Segmentation
- Object tracking
- Behaviour analysis

Many applications in security and traffic follow this flow, which consist of five parts. The algorithms which are used for a given part vary per application.

7.2 Describing Generic Computational Blocks

As mentioned in the previous section, applications consist of algorithms. An algorithm can be defined as a finite set of unambiguous instructions performed in a prescribed sequence to achieve a goal, however the size of this set can vary tremendously between algorithms. Algorithms can even be constructed from multiple sets. To avoid ambiguity the notion of GCB is introduced. GCBs are the building blocks of algorithms, an algorithm can contain one or multiple GCBs to produce its output. GCBs are basic computations which are often used in more then one algorithm. Examples of GCBs in VCA are, histogram creation, integral image calculation, convolution filter, threshold. In VCA a GCB can be considered as one process pass over the complete image.

Determining whether a processing unit is suited for a GCB requires properties of GCBs and processing units which can be compared. These properties are used to describe the GCBs and are adopted from [24]. The work of Nugteren presents a classification and description for image processing algorithms. First the used properties are explained followed by the proposed notation.

- Input

- Output
- Complexity
- Parallelism
- Estimated Number of Calculations(ENoC)

Input $\{Size, Number, Reuse\}$

Input describes the size of the input set for the GCB and how many elements of the input are used per operation. There can exist multiple inputs and these inputs may originate from different GCB. Inputs can be from different locations, however a input can only be from location. E.g. one input can come from the external memory, while another input is located in the internal memory. Reuse can occur if multiple input elements map to one output element given that the input and output size are equal. Reuse is an important factor for the internal memory, which can be used to locate the reused inputs close to the processing. Reuse can be determined in various ways, e.g. profiling, static analysis, simulation. An example of profiling is [1] which uses the address trace of one or more GCBs to determine the reuse distance. This is the number of distinct memory access between the reuse of a data element. The reuse distance can be used to calculate the total reuse. section 7.4 uses static analysis to determine the reuse.

Output $\{Size, Number, Reuse\}$

Output describes the size of the results produced by the GCB and has the same parameters as the input. The value of these parameters does not need to be the same as the input. Reuse of the output is the reuse within the GCB. This is the case when a previous produced output is used to produce another output, this is often referred to as recursion. A GCB may produce multiple outputs which can be stored on different locations, e.g., internal memory or external memory.

Complexity

The complexity describes what kind of operations the GCB contains. E.g. arithmetic operations, logical operations, branches, floating point operations or transcendental operations which can more difficult to compute e.g. $\sin \cos$.

Parallelism

In computer architecture literature [15] are several different levels of parallelism defined i.e. Instruction Level Parallelism (ILP), Data Level Parallelism (DLP) and Thread Level Parallelism (TLP). These levels describe how a the processing unit architecture exploits the parallelism in the workload. However, when a GCB is analysed for parallelism it is not clear on which processing unit it will run and therefore it is not known which how many instructions there are in a GCB. Therefore, is in this work the notion of Independent processable components(ipc) defined. ipc specifies the groups of operations which can be done in parallel. Exploiting the parallelism in a GCB might require extra operations which are introduced by distributing the work. These extra operations are not taken into account.

Estimated number of Operations (ENO)

ENO describes the number of operations is a GCB, e.g. in case of a convolution filter the calculation of one new pixel times the frame size. This is used to estimate how many calculations will be executed by the processing unit. In case of a 3 x 3 convolution filter the processing unit has to perform the multiplication of 9 input pixels with 9 coefficients followed by the sum of the calculated products and divide the sum by the filter size. which brings the number of calculations to $18 = 9 + 8 + 1$. This is an estimation in practice also the control flow and memory indexing has to be taken into account.

7.2.1 Example GCB

To illustrate the notation and calculations a simple example GCB from the VCA domain is discussed. Frame differing is such a simple operation, which consist of the subtraction and thresholding of two video frames. This can be used to detect motion in a video stream. An example of the input and output is given in Figure 7.1.

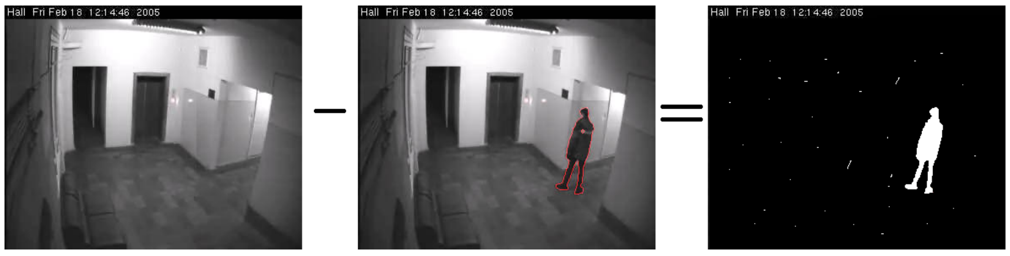

Hall Fri Feb 18 12:14:46 2005
Hall Fri Feb 18 12:14:46 2005
Hall Fri Feb 18 12:14:46 2005

Figure 7.1: GCB example of frame differing, where the left and middle frame are the input and the right frame is the output

$$
\begin{aligned}
&Input \mid input\ used\ per\ operation \rightarrow Output \mid output\ produced\ per\ operation \\
&A \times B \mid Element_x \wedge A \times B \mid Element_y \rightarrow A \times B \mid Element_z \\
&A \times B \mid Pixel_x \wedge A \times B \mid Pixel_y \rightarrow A \times B \mid Pixel_z
\end{aligned}
$$

The input and output frames are denoted as "$A \times B \mid Pixel_x$", where "$Pixel_x \mid Pixel_y$" and "$\rightarrow Pixel_x$" denote the two input elements which map to one output $Pixel_z$. In order to find a suited processing unit the parameters in the description must be quantified to determine the required bandwidth and operations per byte. This quantification can be based on many factors, e.g the provided input resolution, required output precision. A possible quantification of the parameters could be; 1 byte for $Pixel_x$ and $Pixel_y$, 1 byte for the output $Pixel_y$, A frame size of 640·480 (VGA) which represents $A \times B$. These numbers are then placed the description which becomes:

$$640 \times 480 \mid 1\ byte\ \wedge\ 640 \times 480 \mid 1\ byte\ \rightarrow\ 640 \times 480 \mid 1\ byte$$

This means to produce one result, 2 bytes have to be read from the memory and 1 byte is written to the memory which brings the total memory usage $640 \times 480 \times 24\ bytes$. The operations are estimated on 4, i.e subtract, absolute value, compare, branch. These operations have to be performed for al the elements in the output frame which brings the total amount of operations to $640 \cdot 480 \cdot 4$. This GCB contains large amount of parallelism which can be derived from the fact that the produced output is not reused. The operation from the two input elements to the output element can be calculated in parallel for every output element. The amount of parallelism is therefore denoted as $640 \cdot 480$. The complexity consist of logic, arithmetic and branches, this can be concluded from the frame differ operations mentioned in this paragraph.

- **Input**: Two $Pixels \times A \times B$
  Reuse: No reuse
- **Output**: $A \times B$ binary output frame
  Reuse: No reuse
- **Parallelism** : $A \cdot B$
- **Complexity**: Logic, arithmetic, branches
- **ENCPO**: 4 operations per element

The properties enumerated above cannot be placed in the processing unit model directly, the model uses operations per byte$[ops/byte]$ and giga operations per second$[Gops/sec]$. These can be derived from the GCB properties. $[ops/byte]$ is the amount of operations divided by the sum of the in- and output. In order to calculate the $[Gops/sec]$ it must be known at which rate the frames should be processed. If for instance a frame rate of 30 frames per second is required then the $[Gops/sec]$ becomes the total amount of operations times the frame rate.

$$\frac{640 \cdot 480 \cdot 4}{640 \cdot 480 \cdot 24} = 0.167\ [ops/byte]$$
$$640 \cdot 480 \cdot 4 \cdot 30 = 0.037\ [Gops/sec]$$

7.3 Operations per byte of a GCB

Getting an accurate indication of the required performance of an application is key to the success of the method presented in this document. The complexity of the GCB and the requirements (frames per second, or image size) determine if the GCB is computational bounded or memory bounded. There exist several options which can be used to estimate the complexity, the operations and memory accesses, of a GCB. These options range from high accuracy to rough estimations which can yields large errors. The most accurate option is to dynamically analyse the source code on a processing unit. However the processing unit has not been selected yet, and running this on all the available processing unit requires to much time. Because dynamic analysis or profiling is not a option are various forms of static profiling discussed. These range from level pseudo-code estimations up to low level assembler estimations.

- Paper and pencil & Pseudo code
- intermediate language
- Assembly

Pseudo code Pseudo code is a compact and informal high-level description of the operating principle of a algorithm. It uses the structural conventions of a programming language, but is intended for human reading rather than machine reading. Details are typically omitted in pseudo code because they are not essential for understanding of the algorithm, e.g. variable declarations, system-specific code and some subroutines. The pseudo code can give a idea which calculations are important for an algorithm. Listing 7.0 shows pseudo code for the frame differing GCB from subsection 7.2.1 the operations extracted from the pseudo code are shown in Table 7.1. Pseudo code is not an agreed upon standard.

Assembly In the case were there is assembly code available, or source code from a high level programming language which can be compiled. The assembly code also includes address calculations, register transfers The disadvantage of assembly code analysis is that assembly is usually generated for a certain architecture this means that the amount an type of operations are depended from the architecture things as register spilling are taken into account. Some things are hard to determine from assembly code e.g. loop trip. Listing 7.2 shows the assembly code of the frame differing GCB generated by GNU Compiller Collection (GCC) for the X86 architecture.

intermediate code Pseudo code is a to high-level description to make a accurate estimation, Assembly code is low-level but depends on the architecture. Therefore would it better to use a notation which is low-level but not architecture depended. This is found in intermediate representation, which is the description of an abstract machine designed to aid in the analysis of programs. The term comes from the use in compilers, where a compiler first translates the source code of a program into a form more suitable for code-improving transformations, as an intermediate step before generating object or machine code for a target architecture. Low Level Virtual Machine (LLVM) is such a compiler infrastructure, which is designed for compile-time, link-time, run-time, and "idle-time" optimization of programs written in arbitrary programming languages. LLVM can used to generate an intermediate code which can be used to extract the operations and bytes. In many cases is LLVM able to extract loop-trip-counts and perform architecture independent optimizations. Listing 7.1 shows the intermediate code of the frame differing GCB generated by LLVM.

Listing 7.0: Pseudo code example of frame differing

```
for all $pixels \in Frame$ do
    if $ABS(pixel_{in1} - pixel_{in2}) < 16$
        $pixel_{out} \leftarrow background$
    else
        $pixel_{out} \leftarrow foreground$
```

Listing 7.1: Intermediate code of frame differing generated by LLVM

```
bb:
  %indvar = phi i64 [0, %entry], [%indvar.next, %bb]
  %scevgep  = getelementptr [307200 x i8]*
        @output_img, i64 0, i64 %indvar
  %scevgep2 = getelementptr [307200 x i8]*
        @input2_img, i64 0, i64 %indvar
  %scevgep3 = getelementptr [307200 x i8]*
        @input1_img, i64 0, i64 %indvar
  %0 = load i8* %scevgep3, align 1
  %1 = zext i8 %0 to i32
  %2 = load i8* %scevgep2, align 1
  %3 = zext i8 %2 to i32
  %4 = add i32 %1, 15
  %.off = sub i32 %4, %3
  %or.cond = icmp ugt i32 %.off, 30
  %storemerge = select i1 %or.cond, i8 -1, i8 0
  store i8 %storemerge, i8* %scevgep, align 1
  %indvar.next = add i64 %indvar, 1
  %exitcond = icmp eq i64 %indvar.next, 307200
  br i1 %exitcond, label %return, label %bb
```

Listing 7.2: Assembly code of frame differing generated by GCC for a X86 processing unit

```
    .L10:
        leal    (%rax,%rbx), %r9d
        movslq  %r9d,%r9
        movzbl  (%rdi,%r9), %r10d
        movzbl  (%rsi,%r9), %r11d
        subl    %r11d, %r10d
        addl    $15, %r10d
        cmpl    $31, %r10d
        sbbl    %r10d, %r10d
        notl    %r10d
        movb    %r10b, (%rdx,%r9)
        addl    $1, %eax
        cmpl    %ecx, %eax
        jne     .L10
    .LVL4:
    .L8:
        addl    $1, %ebp
        addl    %r13d, %ebx
        cmpl    %r8d, %ebp
        je      .L9
    .LVL5:
    .L3:
        movl    %r12d, %eax
        testl   %ecx, %ecx
        jg      .L10
    .LVL6:
        jmp     .L8
    .L9:
```

Approach	Pseudo code	LLVM	Assembly
Operations	5	16	21
Bytes	$3 \cdot a$	3	5

Table 7.1: Results of different analysis approaches which are used to estimate the operations and bytes of frame differing

7.4 VCA use case

The GCBs found in the prestudy are used to construct an application, which is used as a test case for the proposed method. The task-graph of the application is shown in Figure 7.2. This section explains how a suited processing unit can be found for a certain GCB. Each subsection presents a GCB, describes the properties listed in section 7.2 and discusses which processing unit is suited for this GCB.

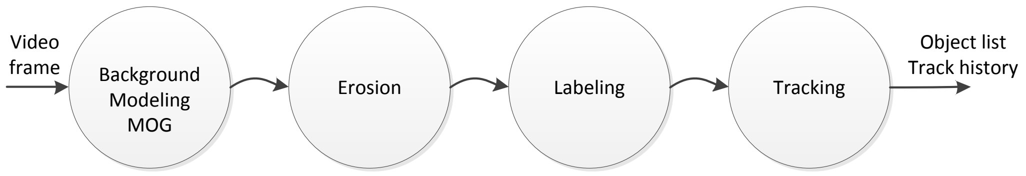


Figure 7.2: Task graph containing base on the de facto standard flow, found in the prestudy [31]

Frame size	FPS	$pixel_\alpha$	$pixel_\beta$	$Component$	$Label$	$Region_r$	$Region_c$
2352×1568	30 FPS	3 bytes	1 byte	24 bytes	4 bytes	12 bytes	20 bytes
5616×1568	30 FPS	3 bytes	1 byte	24 bytes	4 bytes	12 bytes	20 bytes

Table 7.2: Quantisation of the GCB variables and parameters

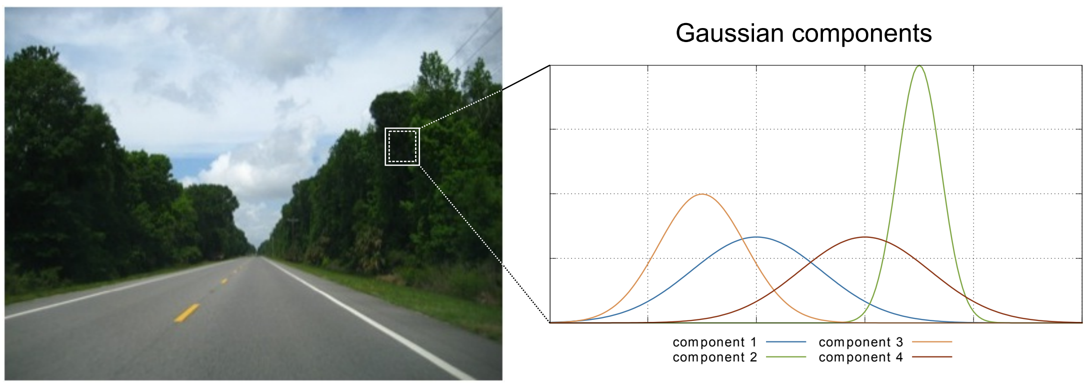


Figure 7.3: Example of mixture of gaussians which keeps a set of normal distributions to determine whether a pixel belongs to the background

7.4.1 Mixture of Gaussians

Mixture of Gaussians (MoG) background modelling method is introduced by Stauffer and Grimson in 1999 [32], and is now the most used method for background subtraction [31] due to its speed and simplicity. In this method, each pixel is modelled as a mixture of normal distributions and all the pixels are qualified as foreground or background.

The GMM stores K separated normal distributions(components) for each pixel parametrized by mean, variance and mixing weight, with K typically between 3 and 5 (depending on the complexity of the scene). Each incoming pixel is to compared to the K components belonging to the location of the pixel. The component which is "close' and has the largest weight is classified as the current background, the component is then updated with a learning factor and the pixel is assigned a binary label. A sample is "close" to a component if the Mahalanobis distance from the component falls within, for example, three standard deviations. If there are no "close" components, a new Gaussian component is created with a predefined mean, weight and a large initial variance. If the maximum number of components is exceeded, the component with lowest weight will be discarded.

GCB description

$$A \times B \mid pixel_\alpha \wedge A \times B \times K \mid component \rightarrow A \times B \mid pixel_\beta \wedge A \times B \mid component$$

The MoG operation uses one input pixel and K gaussian components to produce one background pixel and an updated gaussian component. The pixels and components are both in- and outputs. The components contain state information, However this is not reused within the same frame.

Input: $A \times B$ RGB input frame, $A \times B \times K$ Gaussian components
Reuse: No reuse
Output: $A \times B$ binary output frame, $A \times B$ Updated Gaussian component
Reuse: No reuse
Parallelism : $ipc = A \cdot B \cdot K$,
Complexity: Logic, arithmetic, divisions, reciprocal square roots, branches, floats
ENO: $A \cdot B \cdot 325$ operations

Hardware selection

This section will give an example how a suited processing unit is determined for MoG, first the parameter in the GCB description above need to be quantized. This is done using Table 7.2. Next the operations per second and the operations per byte are determined as shown in the example of frame differing (7.2.1).

$$\frac{2352 \cdot 1568 \cdot 325}{2352 \cdot 1568 \cdot (3 + 1 + (5 \cdot 24))} = 2.419\ [ops/byte]$$
$$2352 \cdot 1568 \cdot 300 \cdot 30 = 33.19\ [Gops/sec]$$

The GCB contains a parallelism, which can be concluded from the fact that the input and the output are not reused within a frame. The icp of MoG is $A \cdot B \cdot K$, because the operation from pixel to pixel and the operation on the components can be done in parallel. This parallelism makes the GCB suited for Graphical Processing Unit (GPU)s and FPGAs, However this GCB also contains floating point operations which are inefficient on a FPGA compared to a GPU. The branches can have a performance limiting effect this is discussed in more detail in section 5.4

7.4.2 Erosion

Erosion is one of the primary operations in morphology, the basic idea in binary morphology is to probe an image with a simple, pre-defined shape, drawing conclusions on how this shape fits or misses the shapes in the image. This simple "probe" is called structuring element, i.e. a subset of the space or grid. The structuring element used in Figure 7.4 is a 3 by 3 grid which is marked with grey crosses. This operation can be described as $z \mid (B)_z \subseteq A$ where the erosion of A by B is the set of all points such that B translated by z is contained in A. In other words the structuring element is placed on a pixel of the input image, if all the pixels in the structuring element belong to the shape (white) the centre pixel remains part of the shape in the output image. This is repeated for every pixel in the input image. The structuring element is often referred to as neighbourhood.

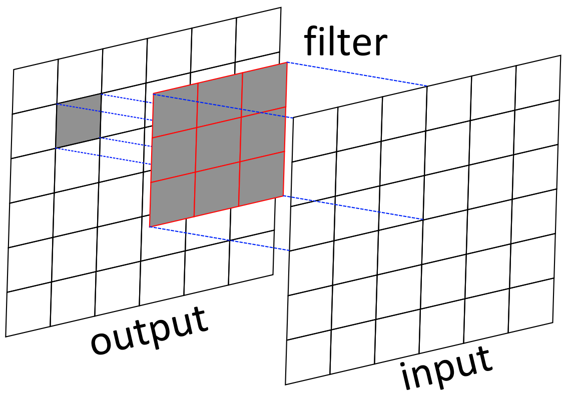


Figure 7.4: Example of a 3x3 erosion operation

GCB description

$$A \times B \mid (n \times n)Neighbourhood_\beta \rightarrow A \times B \mid pixel_\beta$$

Erosion uses per operation a $(n \times n)Neighbourhood$ of $pixel_\beta$ from the input image to produce one output pixel. The erosion operation uses multiple pixels to produce one output and input frame is the same size as the output frame, from this it can be concluded that there is possible reuse. Because there is no relation between pixels in the input frame or between pixels in the output frame, this means that erosion can be computed in parallel for every pixel.

Input: $A \times B\ pixel_\beta$
Reuse: $(A \cdot B - boundaries) \cdot (n \cdot n - 1)$
Output: $A \times B\ pixel_\beta$
Reuse: none
Parallelism :$ipc = A \cdot B$ **Complexity**: Logic, arithmetic
ENO: 115

Hardware selection

For selecting hardware the requirements in Table 7.2 are taken into account. For this calculation example a ·5 erosion filter is used. The GCB uses a $5 \cdot 5$ Neighbourhood which means that each pixel, except for the borders, is used 25 times. This creates a huge load for the memory $(A \cdot B \cdot 25)$ in which the frame is located. This can be overcome by reusing the pixels in the neighbourhood, reuse requires a internal memory close to the processing unit. The size of this memory is determined by the neighbourhood size. In case of a 5x5 neighbourhood and a row-wise order, is the last reuse of a pixel after 5 times the row length. To be exact four times the row length plus the neighbourhood width. This means that the minimum amount of internal memory is: $2352 \cdot 4 + 5 = 9413 Bytes$, often it is faster to use more local memory to at least store 5 lines because this makes the memory addressing simpler. In the case a circular buffer is used can it be profitable to use a power of two for the number of lines which are stored in the local memory. This reduces the otherwise expensive modulo operation to an add combined with a bitwise and.

$$\begin{aligned}\frac{ENO}{A \cdot B \cdot (n \cdot n + 1) \cdot pixel_\beta} = \frac{2352 \cdot 1568 \cdot 115}{2352 \cdot 1568 \cdot 25} &= 4.600\ [ops/byte] \\ ENO \cdot FPS = 2352 \cdot 1568 \cdot 115 \cdot 30 &= 12.72\ [Gops/sec]\end{aligned}$$

The GCB contains a large amount of input reuse which requires an internal memory of atleast 9413 Bytes. This GCB also contains a large amount of parallelism, which can be concluded from the fact that the produced output is not reused within a frame. Erosion has a ipc of $A \cdot B$, because the operation from pixel to pixel can be done in parallel. This parallelism makes the GCB suited for GPUs and FPGAs, This GCB can run on every processing unit given that enough internal memory is available.

7.4.3 Connected component labelling

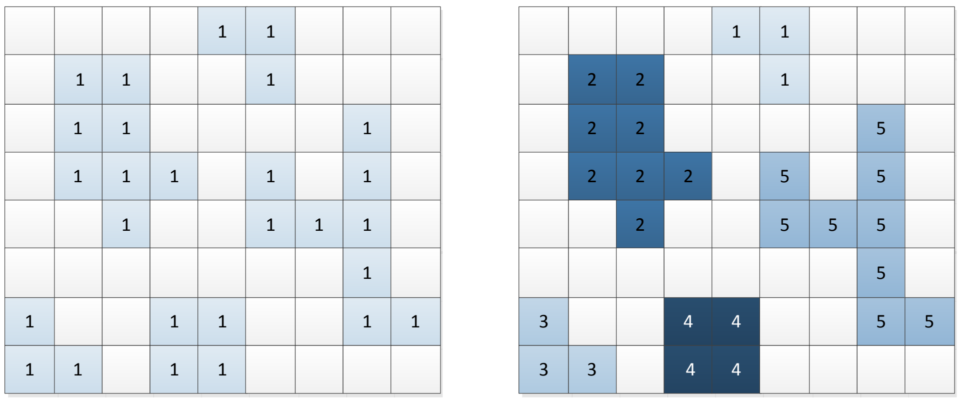
Connected Component Labeling (CCL), blob extraction, or region extraction is an algorithmic application of graph theory, where subsets of connected components are uniquely labelled based on a given heuristic. In the field of image processing CCL is used to assign a unique label to neighbouring pixels with an identical value. This results in a set of isolated pixels with a unique label. These set can be further inspected for properties as size, position or perimeter.

CCL typically uses a 3x3 neighbourhood to determine whether a pixel belongs to the same set, if the pixel belongs to the same set it is assigned the label associated with that set. This introduces a problem because the label of the current pixel depend on the label of the neighbouring pixels which in terms depends on the current pixel.

There exist various approaches to CCL which use one or more passes over the image. For this example a two-pass method is considered, the two-pass processes the frame in a row-wise manner. In the first pass the connectivity is checked and a temporal label is assigned, in the second pass the temporal labels are replaced by the label of its equivalence class. This occurs when pixel sets are merged e.g. if a V-like shape is labelled, it is not known until the bottom of the V is reached that the two lines belong to the same set.

GCB description

$$A \times B \mid \frac{1}{2}(n \times n) Neighbourhood \rightarrow A \times B \mid label$$



(a) input sample (b) output sample

Figure 7.5: Connected component labelling example, where all the neighbouring pixels are assigned an identical label

CCL uses per operation a $(3 \times 3)Neighbourhood$ of $pixel_\beta$, however if the frame is processed in a row-wise manner are not all labels known. Only the pixels on the top and left mid are labelled. This reduces the required internal memory from to 2 lines + 3 pixels for a $(3 \times 3)Neighbourhood$ to 1 line + 1 label. Which brings the required internal memory to $(2352 \cdot 1 + 1) \cdot 4bytes = 9412bytes$. This GCB maps a neighbourhood to a label which is on it self parallel for every label, however the produced labels are reused within a frame. This greatly reduces the parallelism in the GCB. CCL can be performed in parallel processing unit using a multi-pass approach as proposed in [19].

Input: $A \times B\ pixel_\beta$
Reuse: none
Output: $A \times B\ Label$
Reuse: $(A \cdot B - boundaries) \cdot (3 + 1)$
Parallelism:$1 \leq ipc \leq A \cdot B$,
depending on the number of frame passes
Complexity: Logic, arithmetic
ENO: 127

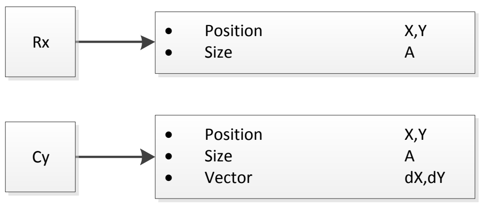
Hardware selection

For selecting hardware the requirements in Table 7.2 are taken into account, this table states that 4 bytes are needed to store a label. Which is calculated using the worst-case. The minimum size for a group of pixels which share the same label is one, however if the previous mentioned erosion filter is taken into account this size becomes 6x6. Which consist of a 5x5 neighbourhood and a border on 2 sides. The erosion filter removes pixels groups smaller than 5x5, this makes the worst-case pattern a chain of 5x5 pixels separated by one pixel. Which would result in a maximum of 102443 labels this requires 17 bits, in Table 7.2 the label size is rounded a unsigned integer size. In practice often assumption are used to reduce the number of possible labels in order to use less bits. The reuse of output within the frame makes this GCB less suited for parallel architectures and therefore more suited for a CPU or DSP

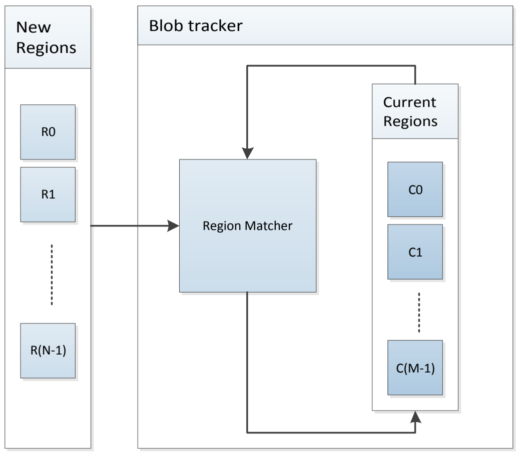
$$\frac{2352 \cdot 1568 \cdot 127}{2352 \cdot 1568 \cdot 97} = 1.309 \ [ops/byte]$$
$$2352 \cdot 1568 \cdot 127 \cdot 30 = 14.05 \ [Gops/sec]$$

7.4.4 Blob-tracker

After background estimation is applied a segmentation GCB (CCL) creates regions of interest from the available foreground mask. These regions are then processed by an object tracker which matches and tracks the regions over multiple frames. For every frame the blob tracker receives N new regions (R_0 .. R_{N-1}) from the segmentation GCB. Within the blob tracker the state of M current regions (C_0 .. C_{M-1}) are tracked. The task of the region matcher within the blob tracker is to match one of the new regions R with a current region C.



(a) Region properties



(b) Blob tracker with new regions R and current regions C

Figure 7.6: Basic blob tracker which matches the new regions with the regions known from the previous frame

There are many properties which can be used to match R_N with C_M, a simple matching principle based on position and speed of a region is assumed in this blob tracking GCB, only the position and size are used. These properties are used to define a candidate region C_m' for C_m. The new regions R_n are compared to the candidate regions and qualified as a match when R_n overlaps with the extrapolated (predicted) position C_m', of C_m in the next frame.

Figure 7.7 depicts the process of selecting a candidate region for C_0 in the current frame t_0, using the position (X,Y) and displacement vector (V) described by (dX,dY). The expected position C_0' in frame t_{+1} is calculated. A simple linear extrapolation is used and only the previous frame t_{-1} is taken into account to calculate the new displacement (V') using (dX,dY) of C_0. C_0' is compared to the new regions(R_0 .. R_{N-1}). If the area of the extrapolated region C_0' overlaps with a candidate region R_0, then is C_0 updated with the displacement of C_0 to R_0 and position of R_0. This is process is repeated for the next frame t_{+2}.

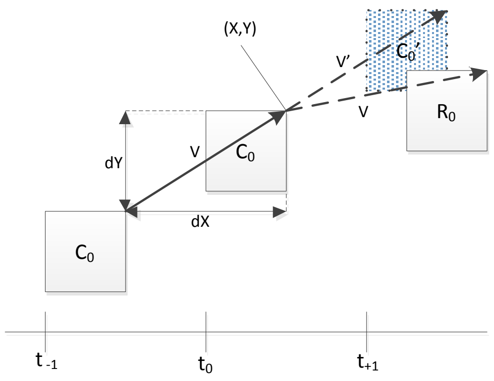


Figure 7.7: Example of extrapolation using the known properties of the current regions

GCB description

$$N \mid Region_r \wedge M \mid Region_c \ \rightarrow \ O \mid Region_c$$

This GCB contains parallelism, the GCB one can take one element from N and compare it to all the elements of M in parallel, or this can be done the from M to N. However, M and N are not know at design time and can vary from 0 to 102443 this makes it hard to create an parallel implementation. The same counts for the reuse of N or M, which need the be known to calculate the required internal memory.

Input: $N + M$ $Regions$
Reuse: $N + M$
Output:O $Regions$
Reuse: none
Parallelism ipc = N or M
Complexity: Logic, arithmetic, branching, floating
ENO

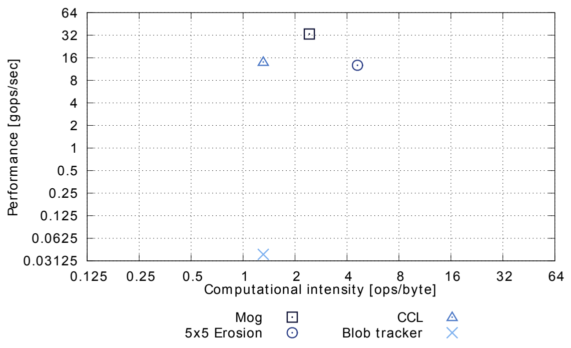
Hardware selection

For selecting hardware are the requirements in Table 7.2 taken into account. However, extra assumptions are needed because N and M are not known. It is assumed that the blob-tracker must be able to track a maximum of 100 object at the same time. The blob-tracker compares every region from N with every region of M this yields a maximum of 10.000 compares. The region descriptors are larger in byte size but less in numbers, therefore they only require $(N \cdot 12 + M \cdot 20)$ a maximum of 3200 bytes internal memory.

$$\frac{N \cdot M \cdot 127}{N \cdot 12 + M \cdot 20} = A\ [ops/byte], \qquad \frac{100 \cdot 100 \cdot 127}{100 \cdot 12 + 100 \cdot 20} = 1.309\ [ops/byte]\ \texttt{worst-case}$$
$$N \cdot M \cdot 127 \cdot 30 = B\ [Gops/sec], \qquad 100 \cdot 100 \cdot 127 \cdot 30 = 0.0381\ [Gops/sec]\ \texttt{worst-case}$$

	MoG	Erosion	CCL	Tracking
Gops	300	115	127	25
Bytes	100	42	97	15
CPU	O	O	X	X
DSP	O	O	X	X
GPU	X	X	O	-
FPGA	-	X	O	-

(a) The GCBs and suited-ness of processing units, where 'X' means suited, 'O' means feasible and '-' means ineffective



(b) GCBs from table 7.8.a in the Roofline model. Without processing units

Figure 7.8: Overview of the discussed GCBs and and suited processing unit types

7.5 Multiple GCBs on one processing unit

Running one GCB on a processing unit is not always efficient, it can be the case there is processing capacity left unused when only one GCB runs on a processing unit. This unused capacity can be consumed when running several GCBs on one processing unit. This is only possible when there are multiple GCBs capable of running on the processing unit with spare capacity. An additional advantage of combining GCBs on one processing unit, is it can save a transmission of data to an other processing unit. The data produced as output can be used by the next GCB as input. The first GCB can store the produced output in external or in internal memory, The second GCB can consume it from this memory.

The combination of GCBs on one processing unit is modelled as the sum of the individual GCBs. The operations of each GCBs are summed as well as the bytes. Equation 7.2 show the sum of operations on a processing unit, equation 7.3 shows the sum of bytes. The variables MQ_{total} and MQ_{total} are define in section 2.4.1. The timing constrained of each GCB is not relaxed. This is illustrated with an example of two GCBs in table 7.3.

$$gcb \overset{def}{=} \text{Number of GCBs} \tag{7.1}$$

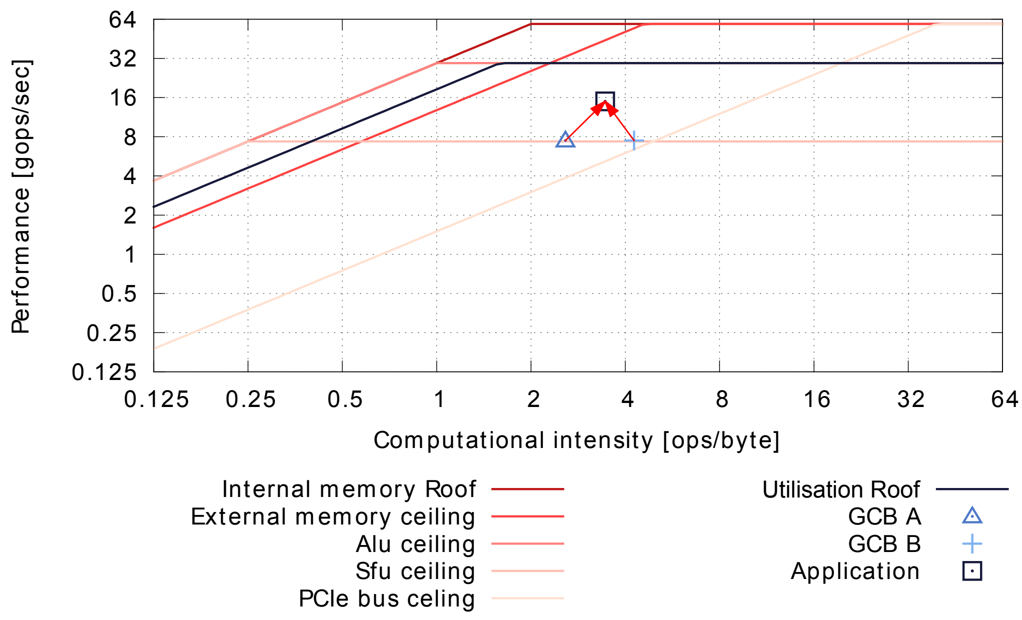
$$CQ^{pu}_{total} = \sum_{g=0}^{gcb-1} CQ_{total} \tag{7.2}$$

$$MQ^{pu}_{total} = \sum_{g=0}^{gcb-1} MQ_{total} \tag{7.3}$$

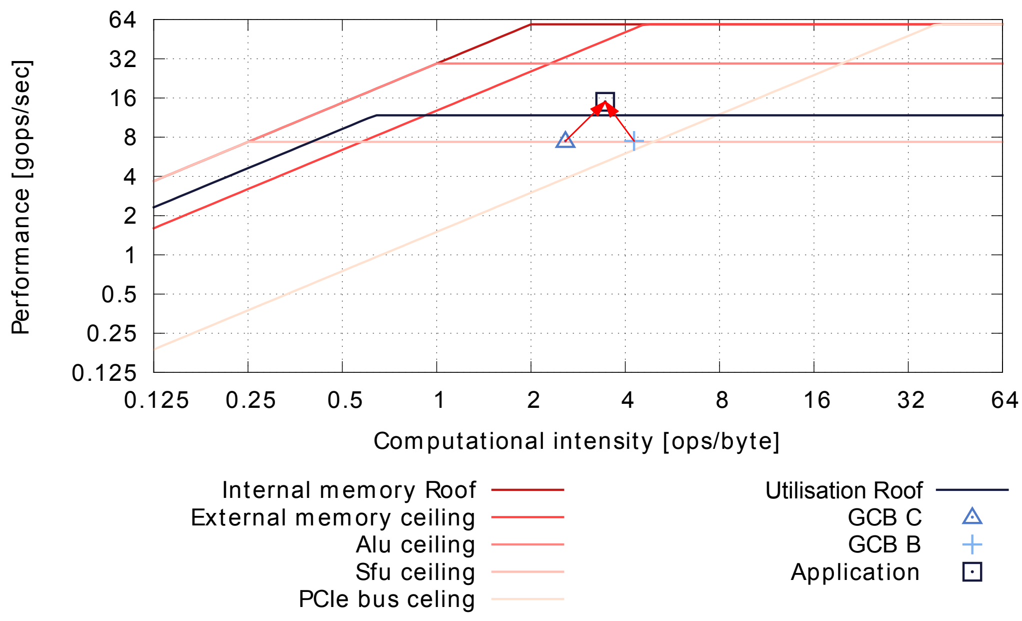
The utilisation Roofline is an important line for combined GCBs, this line shows whether it is possible to run the GCBs combined on one processing unit. An example of combined GCBs is shown in 7.9a. It can be the case two GCBs can each run on a processing unit, where the combination of the two cannot. This case is shown in figure 7.9b. The utilisation Roofline can change as a consequence of combining, this is because the combined GCB can have a different ratio of operations.

GCB	Bytes	Ops	Ops / Sec	Ops / Byte
A	5	20	20	4
B	20	15	15	0.75
Totaal	25	35	**35**	**1.4**

Table 7.3: GCBs A and B combined on one processing unit



(a) Combining GCB A and B on this processing unit is possible because the processor is not fully utilised, and the u roof is not crossed



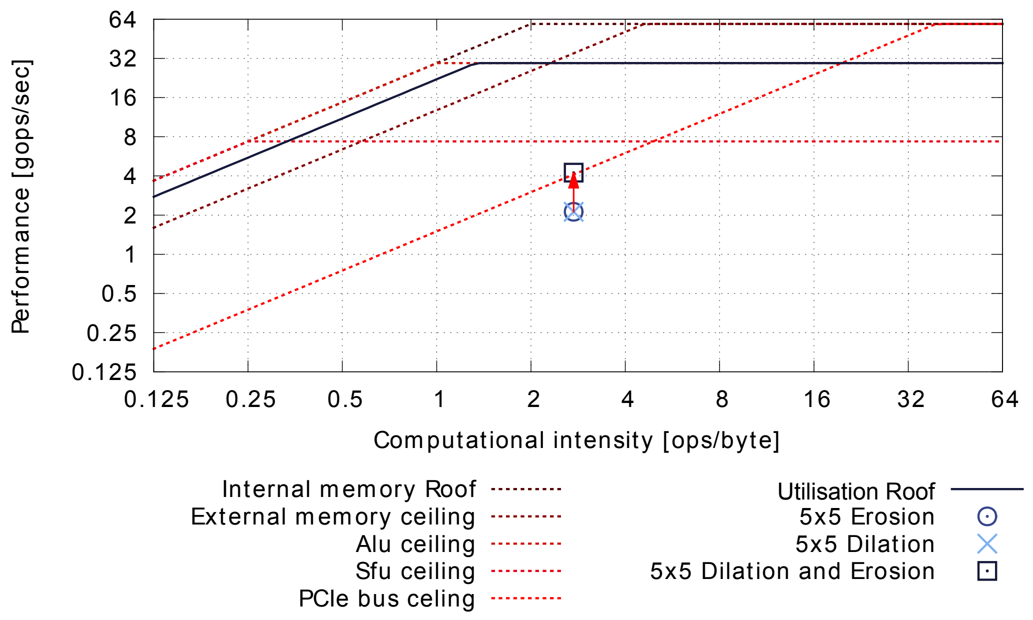
(b) Combining GCB B and C on this processing unit seems possible because the processor is not fully utilised. However, when the GCB are combined the utilisation roof is crossed

Figure 7.9: Combining GCBs A,B and C. The application is the result of the combined GCBs. If the application is depicted above the utilisation Roofline, then it is not possible to combine the GCBs on this processing unit

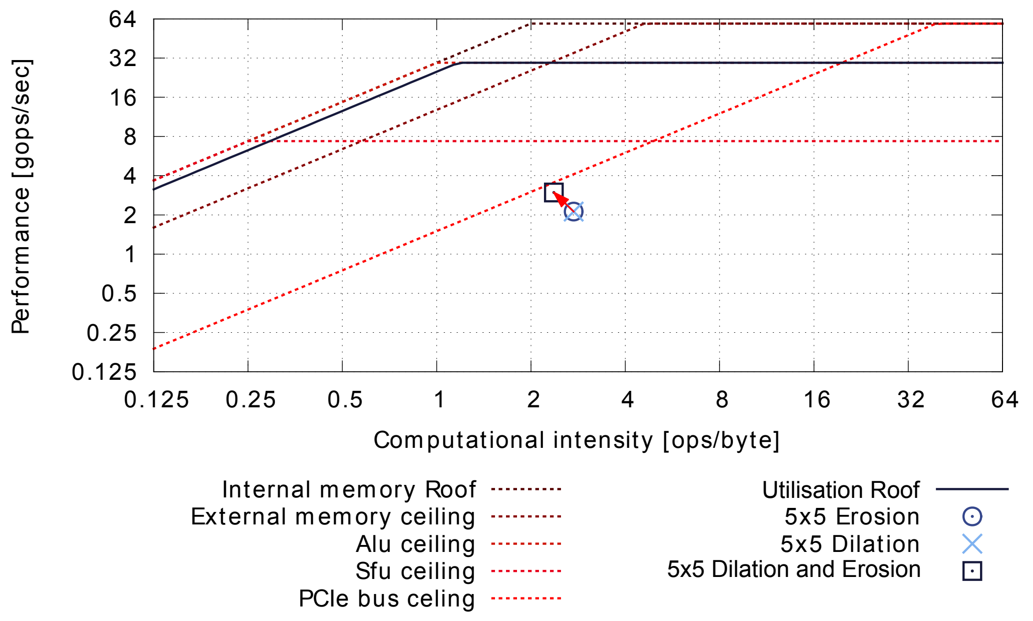
7.6 Pipelining GCBs on one processing unit

In the previous section the effect of running multiple GCBs on one processing unit is discussed. When a GCB running on a processing unit operates on data produced by a GCB on the same processing unit, it might be possible to pipeline these GCBs. Pipelining is the coupling of two or more GCBs in series. Where the result produced by the previous GCB is consumed by the next GCB which, in turns produces a result for the next GCB. In some cases the next GCB cannot start before a certain amount of results is available. In such case a buffer can be used to store results. This buffer can be located in internal or external memory, the internal memory is often faster and therefore the preferred location for the buffer. Reading from and writing to the buffer might require synchronisation to make sure the results are transferred correctly. The synchronisation between reading and writing can cost operations, which is discussed in the next paragraph. Pipelining is modelled similar to combining GCBs, the difference of pipelining is that it aims to use a data source with a higher bandwidth. Pipelining is not always beneficial, e.g. ,when the buffer does not fit into the internal memory or when delay of synchronisation is larger than the data source speed-up.

The effect of pipelining in shown in figure Figure 7.10, with the use of the morphological operations erosion and dilation. Erosion is described in subsection 7.4.2 and dilation is the inverse of erosion. 7.10a shows the combination of erosion and dilation on one processor, in this case the NVIDIA FX1700. Figure 7.10 shows the pipelined version of closing, where the results of erosion are stored in the internal memory of the FX1700. This results in a reduction from 4.24 Gops to 2.94 Gops. The improvement of 1.45 Gops this is expected because the reading from internal memory is 1.6 times faster and writing is 1.4 times as can be seen in 5.4b. The position of the pipelined GCBs on the x-axis has changed in relation to the combined GCBs. This is due to the extra operations required to synchronise the accesses to the intermediate results. From this test it is concluded that the speed-up can be determined from the Roofline model. However, in order to model the pipelined GCB in advance the increase in operations should be known.



(a) Combining Erosion and Dilation



(b) Pipelining Erosion and Dilation

Figure 7.10: The difference between the combined(a) and the pipelined(b) shows pipelining can be beneficial. However, in order to model the pipelined GCB in the Roofline model, it must be known if extra operations are required to handle the synchronisation

7.7 Platform

Applications often consist of multiple GCBs, these GCBs can be combined or pipelined on one processing unit as discussed in the previous sections. However, it can be the case that the computational capacity of one processing unit is not sufficient for multiple GCBs, or the GCBs require different types of processing unit. To overcome this problem multiple processing units must be used. These processing units can be of a different type as shown in figure 7.11, where in this case the platform consist of a CPU and GPU.

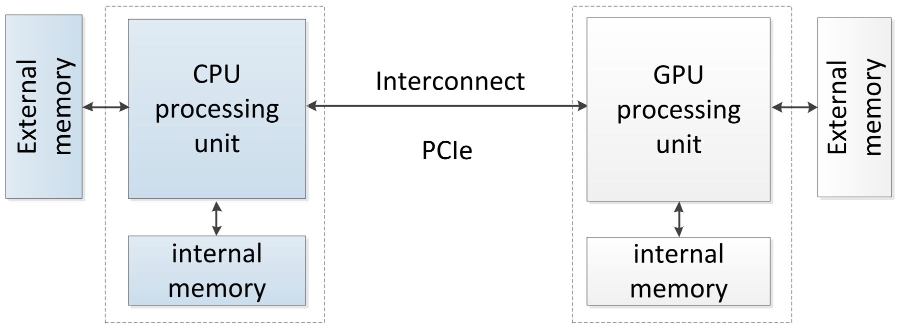
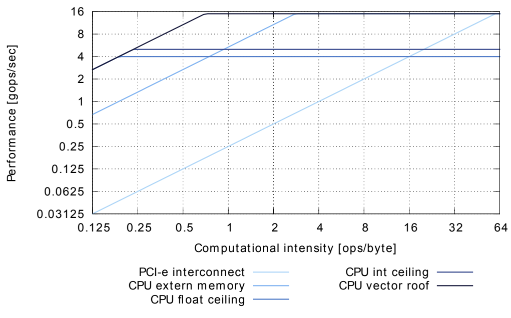


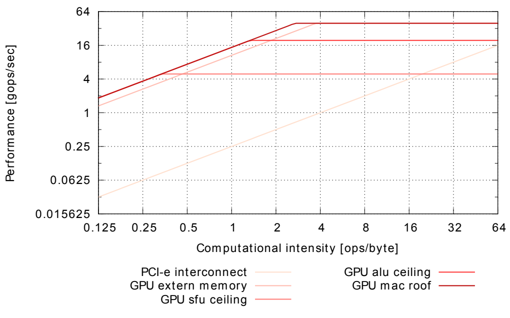
Figure 7.11: Simple example platform

The platform model consist of two or more processing units and is created with the information used to model individual processing units. It is assumed that the processing units operate in-parallel with each other. This makes the total available processing capacity the sum of the individual capacities. This sum is modelled (in the Roofline model) by stacking the individual computational roofs and ceilings on top of each other. The data sources ceilings and roofs are summed and displayed per data source. Figure 7.12 shows the previous explained process where two processing units 7.12a and 7.12a are modelled as one platform 7.12c by summing the individuals capacities. The application is modelled as the sum of the GCBs and is calculated with equation 7.2 and 7.3. In order to determine the operations and accessed bytes of the GCBs, the mapping of GCBs to processing units must be known or assumed. The mapping is used to determine the communication between the processing units.

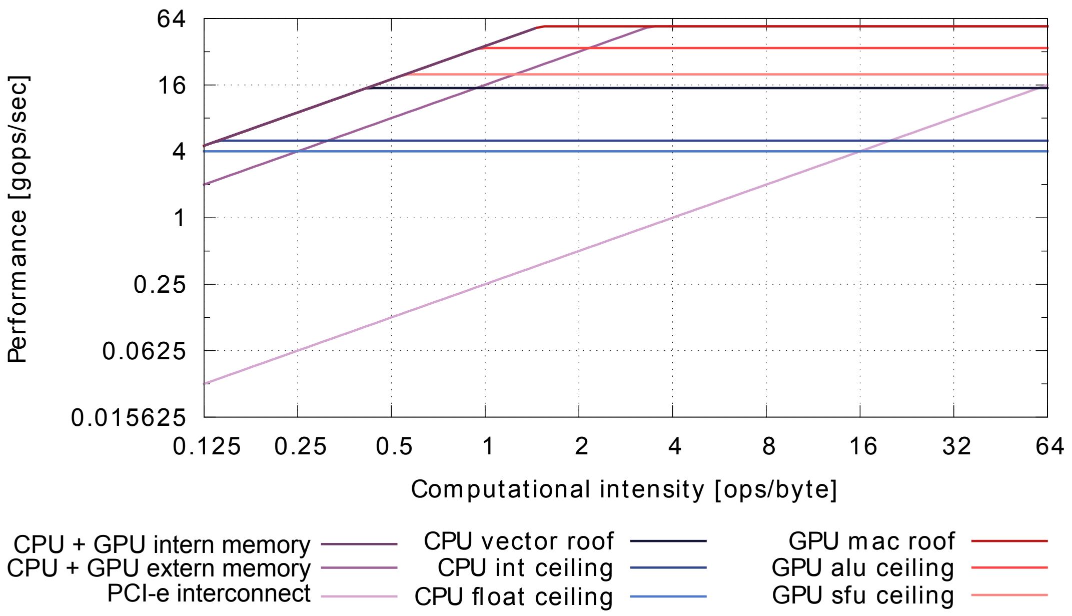
The roof of the platform model gives the maximum available processing capacity. However, this is not always a relevant upper bound for an application. To create a tighter bound for an application the Utilisation Roofline is used. The Utilisation Roofline is explained in detail in section 2.5. The Utilisation Roofline is based on the ratio of different operations in the application and can be calculated using equation 2.21. However, this equation does not take into account that the load of individual processing units might differ. For example, when Processing Unit (PU)1 is utilised for 90% and PU2 for 60%, the utilisation Roofline will show that there is still an 50% performance available. However, there is only an additional 10% increase possible on both platforms before PU1 is above its roof. To overcome this problem the load of the individual processing units must be taken in to account. An example of the Utilisation Roofline is calculated using the GCBs in table 7.4 and equation 7.5, the result is shown in figure 7.13a. Before the Utilisation Roofline can be calculated must the the map-



(a) CPU Roofline model



(b) GPU Roofline model



(c) CPU-GPU platform that is composed of the processing units depicted in 7.12a and 7.12b

Figure 7.12: Two individual Roofline models and a Roofline model of the combined processing units. The blue lines show the CPUs ceilings and roofs, The red lines show the GPUs ceilings and roofs and the purple show the combined data source ceilings and roofs

ping of GCBs on processing units must be known or assumed. The mapping has an influence on the load of each processing unit and the communication via the interconnect. It is assumed that the communication via the interconnect is in-parallel with the other data sources and operational costs is neglectable.

Table 7.4 lists an example application of which the Utilisation Roofline is calculated for the platform shown in figure 7.11. The mapping is shown in the table and the GCBs communicate is as follows GCB A communicates with B which communicates with C. The utilisation Roofline is calculated with equation 7.5 is depicted as a blue line in 7.13a.

The Utilisation Roofline depicted in figure 7.13a shows the achievable amount processing with re-

	GCB A	GCB B	GCB C
Int	70	10	10
Ext	100	20	10
Bus	1	2	1
Ceil1	30	50	10
Ceil2	20	200	30
Ceil3	10	20	40
	CPU	GPU	CPU

Table 7.4: Example application for platform, Requirements for this application are a frame size of 1280×960 *and a frame rate of* 23

spect to the total processing capacity. This view is less evident as to why this is the achievable amount. To avoid the loss of insight the platform can also be modelled on processing unit level and display the platform load(eqn. 7.9) in each model. Provides a better view on the possible performance bottlenecks. This concept is displayed in figure 7.14 where 7.14a shows the CPU model and 7.14b the GPU model. The GCBs displayed are from table 7.4. This dual view show the same platform as displayed in figure 7.13a

$$pu \overset{def}{=} \text{Number of processing units} \tag{7.4}$$

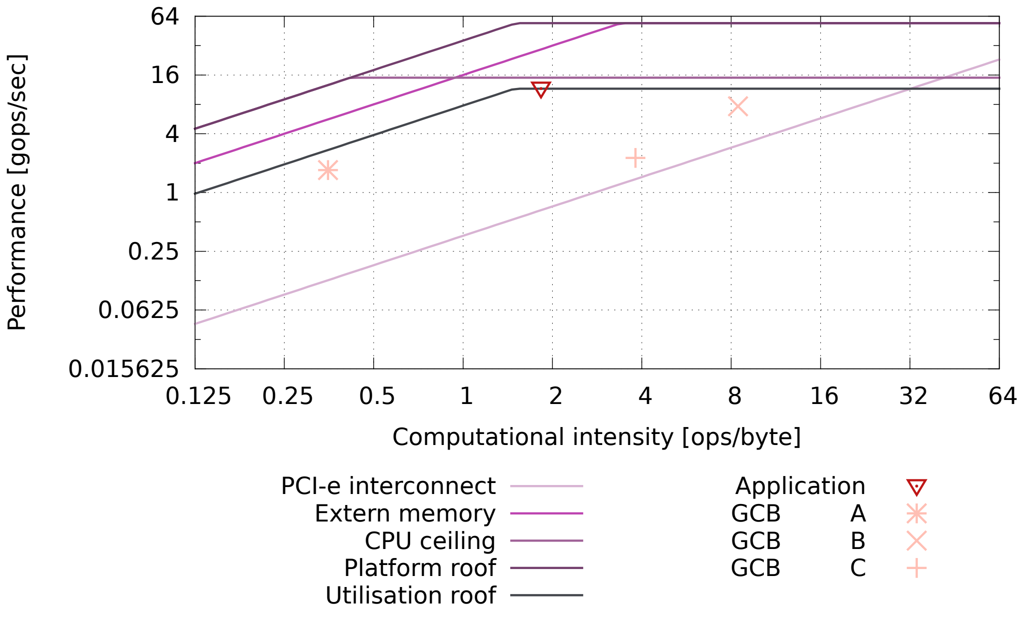
$$\text{Platform\_Utilization} = MIN(Platform_{mur}(x), Platform_{cur}) \tag{7.5}$$

$$Platform_{cur} = \sum_{i=0}^{pu-1} MIN(PuUtil_i, PuLoad_i) \tag{7.6}$$

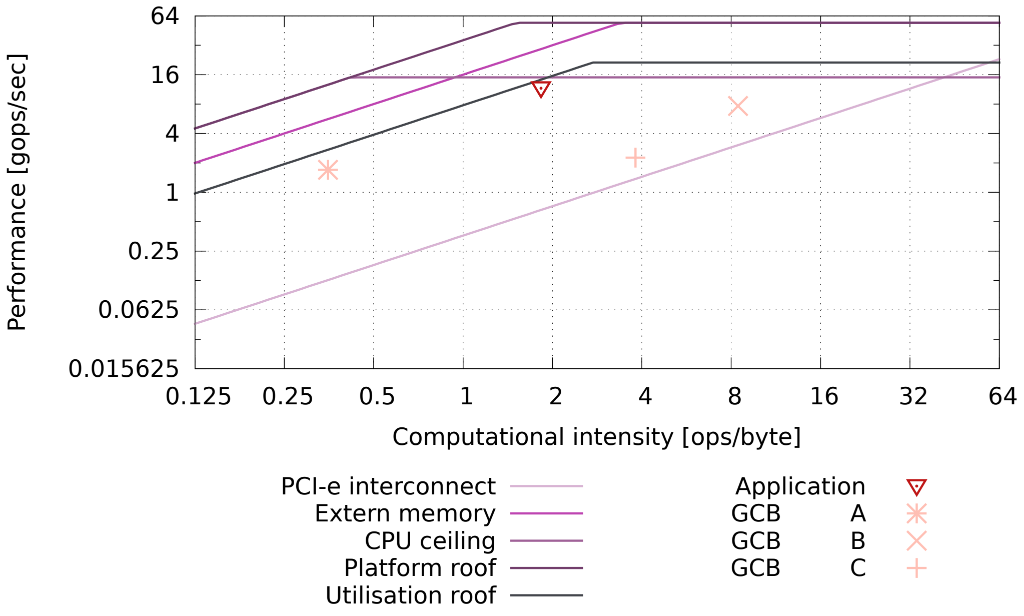
$$Platform_{mur}(x) = \sum_{i=0}^{pu-1} mur_i(x) \tag{7.7}$$

$$PuUtil_i = MIN(mur_i(\frac{MQ^i_{total}}{CQ^i_{total}}), cur_i) \tag{7.8}$$

$$PuLoad_i = \frac{PuUtil_i}{MAX_{k=0}^{pu-1}(PuUtil_k)} \cdot cur_i \tag{7.9}$$

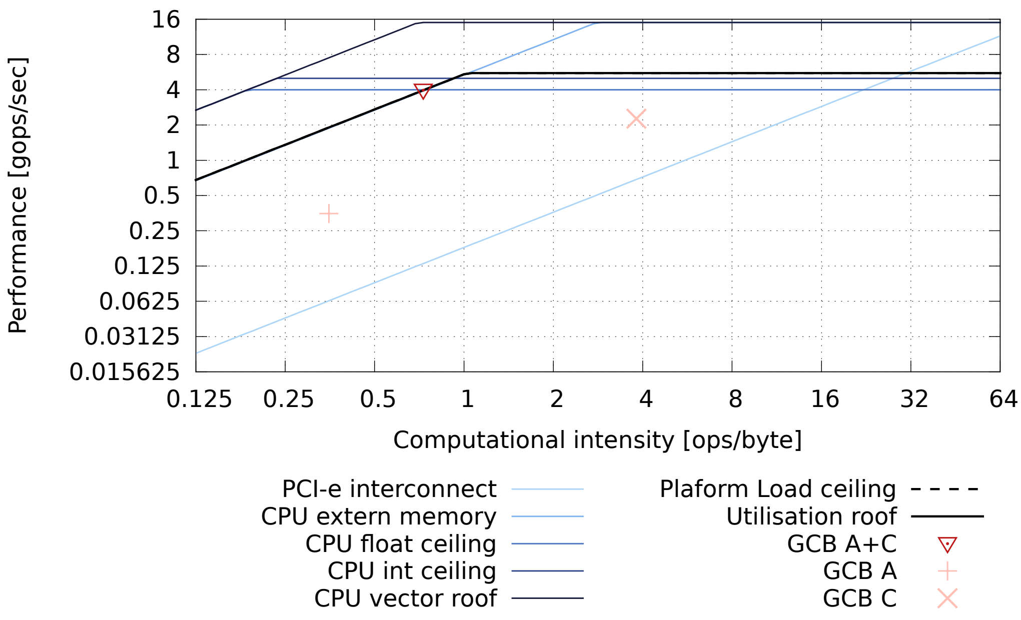


(a) Utilisation Roofline platform

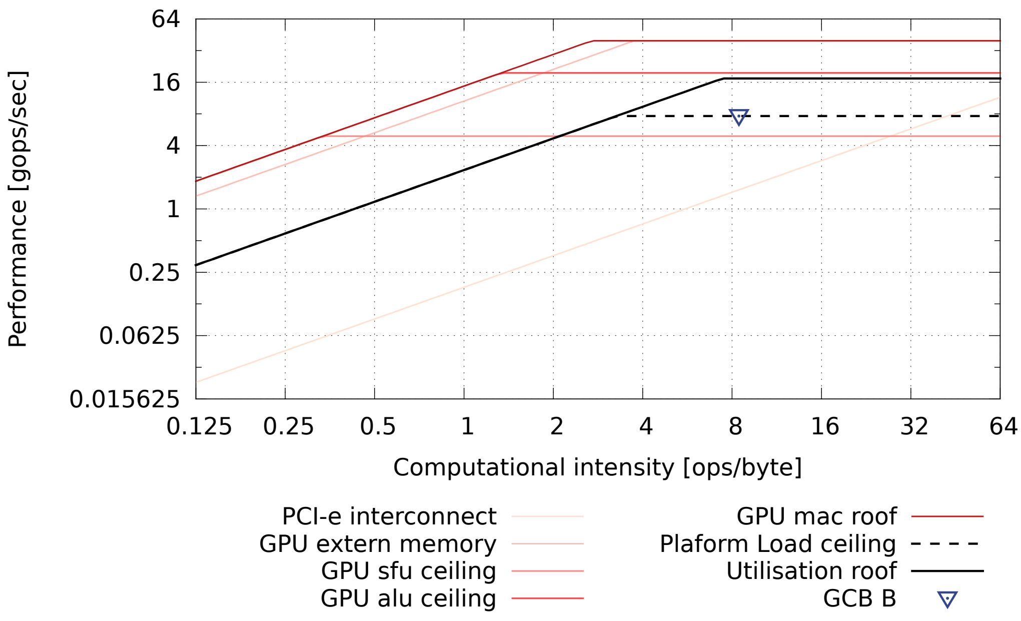


(b) Utilisation Roofline platform without unbalanced loads taken into account

Figure 7.13: This figure shows the difference between a Utilisation Roofline with and without the load of individual processing units taken into account. Figure 7.13b shows there is still a performance increase possible. While in practise there is no performance increase of the platform possible, as shown in figure 7.13a. This is because some GCBs are already on the roof of one of the processing units, this becomse visiable in figure 7.14



(a) The platform Utilisation Roofline shown on CPU level



(b) The platform Utilisation Roofline platform shown on GPU level

Figure 7.14: Modelling the platform on processing unit level provides more insights compared to the platform Roofline model. This view on the platform shows that the sum of GCBs on the CPU have reached the Utilisation Roofline, while GCBs on the GPU are below the Utilisation Roofline. This means that the platform as a whole cannot offer more performance, while the GPU individually can still increase

Chapter 8

From application to platform, a guideline

Throughout this work various aspects of embedded platform selection are presented. This chapter combines these aspects to give an overview of complete process. First, the method is translated to a guideline after which this guideline is used for an example of hardware selection.

8.1 Guideline prerequisites

In order to let a designer apply the method as a guideline some prerequisites are required. First a designer needs to know the exact system requirements especially the requirements regarding data throughput. Furthermore the designer needs information about the kind of application that is executed on the embedded platform. Lastly the designer needs to know some details of the processing units that are available for the selection process. These prerequisites are discussed in the following subsections.

A) Requirements of the system

A designer needs the requirements of the embedded system to be able to select processing units, moreover the designer must be able to derive the required performance in terms of throughput e.g. for video the required number of frames per second and frame size is desired to know. Furthermore requirements like power, cost and availability can form in important factor when selecting processing units for a platform.

B) Application details

To know the kind of processing that is performed on the platform a designer needs information about the application and underlying algorithms. For this an application description, e.g., pseudo-code, application source-code, or assembly code is required from which the required operations and data accesses can be determined.

C) Processing unit details

From the processing units that are available a description of the supported operation types together with the maximum throughput of these operations needs to be known in advance. This is only applicable for processing units with a fixed datapath like Central Processing Unit (CPU),Digital Signal Processor (DSP) and Graphical Processing Unit (GPU). For the Field Programmable Gate Array (FPGA) the amount of resources and the required resources per operation need to be known. Apart from the speed at which a processing unit can operate on data it must also be known at what speed data can be accessed from the different data sources a processing unit supports.

8.2 Processing unit selection guideline

Figure 8.1 depicts the method as proposed in this thesis. From this figure a step by step guideline is constructed. This guideline is partitioned in four steps. Every step summarizes the necessary actions of the guideline step, while detailed descriptions of these actions can be found in corresponding chapters.

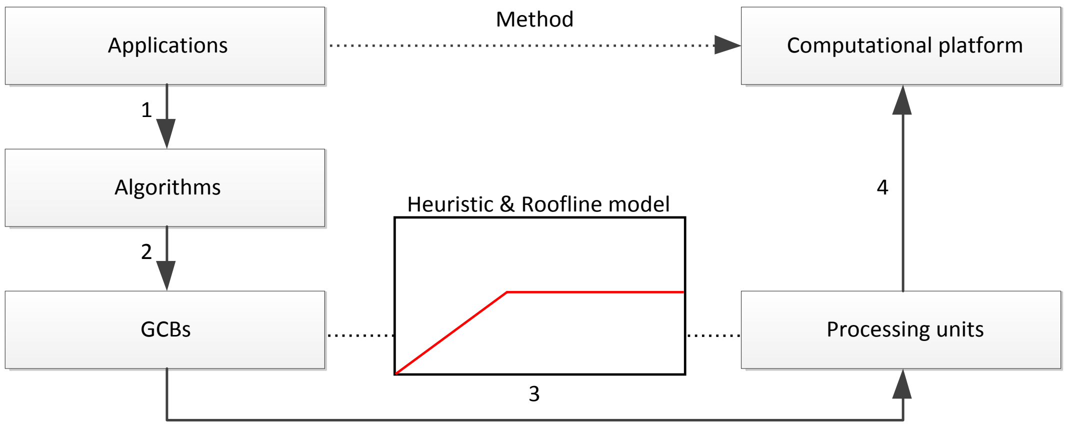


Figure 8.1: Processing unit selection process, step 1 Analyse the application. Step 2 Extract the GCBs. Step 3 Match GCB performance with processing unit models. Step 4 Define the required computational platform

Step 1: Decompose application into multiple algorithms

First step of the selection is the analysis of the application in order to obtain the algorithms which are responsible for the relevant processing. For example to track objects in an environment (as described in chapter 7) algorithms like background detection and segmentation are needed.

Step 2: Decompose algorithms into GCBs

This step decomposes an algorithm into one or more Generic Computational Block (GCB)s. Algorithms often have the same constructs these constructs can be reused in multiple algorithms. Such a construct is a GCB that contains the core processing of an algorithm. By assuming reuse in algorithms a designer only has to define the description of a GCB ones. This description can then be reused in other algorithms. Such a description contains information of the GCB like input size, parallelism and the kind and number of operations. How to apply this step is describe in chapter 7 When this step is explored a designer knows how the GCBs in an application work internally and in what way it operates on data from different data sources. This information is stored into a $GCB_{details}$ list.

Step 3: Apply heuristic and Roofline model

The method described in this thesis uses the Roofline model to select which processing unit matches best with a given GCB. Before applying the Roofline model a heuristic is used that pre-selects processing units for a given GCB. This heuristic selection is based on the GCB properties, e.g., parallelism, used data types. The heuristic is a pre-selection function H on a list of available processing units, and GCB details. The resulting list is a list of candidates $(PU_{candidates})$ for the GCB, these candidates are suited for the type of processing of the GCB. This heuristic prevents bad selections like, trying to select a processing unit that does not support floating point data for a GCB that requires this type of data processing.

$$(PU_{candidates}, GCB_i) = H(PU_{list}, GCB_i) \tag{8.1}$$

For the $PU_{candidates}$ list Roofline can be constructed. The corresponding GCB is then modelled into Roofline model or Utilisation Roofline model of these processing units. Creation of the Roofline model is described in chapter 2, while example Rooflines for different types of processing units can be found in chapter 3 (CPU), 4(DSP), 5(GPU) and 6(FPGA). This selection can result into one or more suitable processing units for a given GCB.

Step 4: select processing units and create a platform

From the previous step a designer acquires a list $PU_{candidates}$ for a given GCB. When the list contains more then one candidate or the application contains more then one GCB, multiple GCBs can be executed on the same processing unit. Using this a designer can decide to run GCBs together on a processing unit, for this extra calculations are necessary that define the ops/byte ratio of this combination. An in depth explanation of these calculations can be found in section 7.5 of chapter 7.

To come to a processing platform GCBs have to be mapped to a processing unit. While this guideline does not deliver a mapping for GCBs on processing units it can supply the designer with hints on which processing unit to select. The complete platform can then be modelled when a designer has selected a mapping of the GCBs from step 3 onto one or more processing units. This allows the designer to compare different Mappings and platforms based on system requirements.

8.3 Example usage of the guideline

This section applies the guideline to an example application. To select a platform, a number of prerequisites is required as described in the previous section. First these prerequisites are listed after which the guideline is applied for the given problem. The example application used throughout this section is object tracking in a video stream.

A) Requirements of the system

The system must perform object tracking on video data with a framesize of 1280x960 and analyse streaming video at 30 frames per second. Further more the power consumption can not exceed 100 watts. Note that all these values are fictive and are only used to depict how the method is used to do processing unit selection for a platform. From the above requirements a requirement list is created that can be used in the processing unit selection process later on.

$$REQ_{list} = (\text{1280x960 @30FPS, Power < 100 Watt years})$$

B) Application details

The application for which a processing platform is selected is an object tracking application. This application identifies and tracks moving objects in a video stream. Source code is available because the example problem and the underlying algorithms are also discussed in chapter 7.

C) Processing unit details

A list of processing units (PU_{list}) from which a selection is made is available and listed in table 8.1. A number of properties such as, cost and bandwidth are listed for processing units in this table.

$$PU_{list} = (A, B, C, D, E, F, G)$$

Processing unit	A	B	C	D	E	F	G
Operations[Gops/sec]	25	20	30	50	50	60	70
Bandwidth [byte/sec]	10	20	124	9	25	150	53
Cost	10	20	30	40	50	60	70

Table 8.1: List of processing units that are available for selection in this example

Step 1: Decompose application into multiple algorithms

The object tracking application consists of two algorithms, a background modelling algorithm and a object tracking algorithm. This concludes this analysis. Note that this step can be more complex when a larger application is analysed.

Step 2: Decompose algorithms into GCBs

In the previous step two algorithms are identified for the example application, in this step GCB, smaller processing blocks are identified. Background modelling contains two GCBs namely background selection by using mixture of gaussians that separates important moving objects from the static background and morphological filtering (Erosion) to remove unwanted noise from the background model. After this the object tracking algorithm is applied on the objects on the foreground (identified by the background model). Two GCBs are used for this purpose first labelling is applied that selects a group of pixels and identifies them as a single object, when the objects have been identified blob tracking can track objects across different frames in the video stream. The GCBs are analysed in chapter chapter 7 the information from this chapter is summarized in table 8.2.

GCBs are then analysed by defining the parallelism inside these GCBs. for this purpose independent processable components (ipc) are defined within a GCB . For image processing this often comes down to the number of pixels or frame components that can be processed in parallel. The GCBs used as an example in this chapter have the following value for ipc. A detailed explanation of these algorithms can be found in chapter 7.

• **Mixture of Gaussians (MoG):** $ipc = framesize \cdot k$ in which k represents the number of gaussians used in the algorithm

• **Erosion:** $ipc = framesize$, with erosion all pixels can be calculated in parallel
• **Connected Component Labelling:** $1 < ipc < framesize$, ipc depends on the labelling approach, worst case every pixel depends on the previous pixel and only 1 pixel can be labelled in parallel. Best case an approach is used in which every pixel gets an index, all pixels can then be processed in parallel but more operations are required per pixel to let the indexes converge and get connected components
• **Blob tracking:** In blob tracking the ipc is defined as the number of blobs (components) that are tracked and compared, notice that this ipc can have very dynamic behaviour in case of a video stream

Table 8.2 list the GCB and their properties. Besides properties such as bytes per pixel, operations per pixel and ipc this table also lists the data types used inside the GCB. This information is used when pre-selecting processing units for GCBs.

	MoG 5 Gaussians	Erosion	Labelling	Blob tracking
OPP	300	127	160	25
BPP	100	60	75	15
ipc	$1280 \cdot 960 \cdot 5$	$1280 \cdot 960$	$1 \vee 1280 \cdot 960$	$\#blobs$
$double$	-	-	-	-
$float$	+	-	-	+
$integer$	+	-	+	+
$char$	-	+	-	-
$uncommon$	-	-	-	-

Table 8.2: The GCBs and their properties, + uses this type of processing within the GCB - doesn't use this type of processing. uncommon data types are data types that are not 8,16,32,64 bit in size. BPP number of bytes accesses per pixel, OPP operations per pixel, to get the total number of ops and bytes the framesize has to be accounted for

Step 3: Apply heuristic and Roofline model

Before Roofline models can be created a heuristic is applied on the available GCBs and processing units. In this heuristic the properties from a GCB (table 8.2) are matches with the properties of processing units to see which processing units are best suited. One part of the heuristic is matching the parallelism of the GCB with the parallelism inside a processing unit. This is done using the independent processable component (ipc) measure defined in Step 2. Table 8.3 shows the required ipc for different types of processing units.

Furthermore the GCBs are matched with a processing unit based on the type of data types and sizes that are processed. This results in table 8.4 where GCBs are matched with a processing unit. Note the the above procedure is just an example of how an heuristic can be applied to do some pre-selection. However it is important that the data types used in the application match the data types and parallelism in the processing unit.

Processing unit A and D can run all GCB that is why these processing units are selected for further

type	ipc requirement
CPU	$ipc \geq C_{\#cores} \cdot C_{ops}MAX$
DSP	$ipc \geq C_{\#cores} \cdot C_{ops}MAX$
GPU	$ipc > C_{\#cores} \cdot C_{ops}MAX \cdot PD$
FPGA	$ipc > arch$

Table 8.3: Processing unit and their capabilities, every GCB consists of number of independent processable parts (ipc), to efficiently use an architecture this ipc should match the number of available cores to be able to exploit the parallelism inside a processing unit. For an FPGA this requirement depends on the implemented architecture ($arch$).PD is defined as the pipeline depth of a GPU. The maximum number of operations that can be executed in parallel is defined as $C_{ops}MAX$.

Processing unit	A	B	C	D	E	F	G
MoG 5 Gaussians	+	-	-	+	-	+	-
Erosion	+	-	+	+	-	-	-
Labelling	+	-	-	+	-	-	+
Blob tracking	+	+	+	+	-	-	-

Table 8.4: List of processing units and GCBs when a processing unit is suited for a GCB it is marked with a '+' else it is marked with a '-'

platform selection. Notice that all processing units with a '+' can be taking into account. However to keep the example below comprehensible A and D are the only processing units used in the rest of the selecting procedure.

Step 4: Select processing units and create a platform

After applying the heuristic that results in table 8.4, the required performance of the GCBs is estimated. Calculations of this performance requirement can be found in chapter 7.

name	GCB_1	GCB_2	GCB_3	GCB_4
type	MoG 5 Gaussians	Erosion	Labelling	Blob tracking
OPP	300.00	127.00	160.00	25.00
BPP	100.00	60.00	75.00	15.00
Gbyte/sec	11.31	4.68	5.89	0.92
Gops/sec	3.68	2.21	2.76	0.55

Table 8.5: GCB requirement of bandwidth [Gbyte/sec] and performance [Gops/sec], which can be calculated using Operations Per Pixel (OPP) and Byte Per Pixel (BPP) and The requirement that the GCBs process frames of 1280x960 pixels at a rate of 30 frames per second. This results in the requirement in GB/sec and Gops/sec, using the given rate of $1280 \cdot 960 \cdot 30 = 36864000$ *pixels per second*

Risk assessment

A Risk is defined that is used in selecting a combination of processing units. Assume the requirement of a given GCB$_X$ has a computational performance of P_{gcb} [ops/sec] and a bandwidth of B_{gcb}

[bytes/sec]. A Processing Unit (PU)$_X$ is defined with memory bandwidth B_{pu} [bytes/sec] and a maximal attainable computational performance P_{pu} [ops/sec]. Instead the bandwidth and maximal attainable computational performance of the processing unit the bandwidth and performance from the utilisation Roofline can be used. Then two risk ratios can be defined as depicted in equations 8.2 and 8.3 ,that define the risk of mapping GCB$_X$ on processing unit PU$_X$. r_P is the computational risk and r_B the bandwidth risk. When one of the risk ratios is greater or equal to one, the mapping is not feasible because the required bandwidth of the GCB or computational performance exceeds the available bandwidth or computational performance of the processing unit (equation 8.4).

$$r_P = \frac{P_{gcb}}{P_{pu}} \tag{8.2}$$

$$r_B = \frac{B_{gcb}}{B_{pu}} \tag{8.3}$$

$$(r_P \geq 1 \vee r_B \geq 1) \rightarrow \text{infeasible mapping} \tag{8.4}$$

When a mapping is feasible the total risk of this mapping R is assumed to be the maximum of the bandwidth risk and the computational risk (equation 8.5). This assumption can be made because an application is memory bound (r_b is dominant) or computational bound (r_p is dominant). However how r_b and r_p are related to R depends on the type of architecture ,mapping and weights a designer adds to these risks. For example reaching the memory bandwidth can be harder then reaching the computational performance.

$$R = MAX(r_P, r_B) \tag{8.5}$$

A Pareto Space

Multiple processing units can be suited for a GCB for this multiple GCBs can be executed on a single processing unit. Furthermore a combination of processing units can be selected to run the GCBs. To select the final platform, configurations ($conf$) can be defined that select processing units and a mapping of GCBs on these processing units. For example GCB_{123} is a combination of GCB_1, GCB_2, GCB_3 as defined in table 8.5. To calculate the bandwidth [byte/sec] and performance [ops/sec] values for GCB_{123} the values for bandwidth and performance of GCB_1, GCB_2, GCB_3 bandwidth is summed and performance is summed. In the pre-selection (step 3) only two candidates are defined processing unit A and processing unit D. A larger space can be selected but to keep this example comprehensible only two processing units are selected and only three out of four GCBs are used namely GCB_1, GCB_2, GCB_3. For every combination of GCBs a risk can be defined for running it on processing unit A and running it on D. This risk depends on the required bandwidth B and performance P of the combined GCB and the processing available on the processing unit. Table 8.6 depicts the performance, bandwidth, computational intensity and risks for processing unit A and D for a number of combined GCBs.

$comb$	P [Gops/sec]	B [Gbyte/sec]	[Ops/Byte]	Risk on PU A	Risk on PU D
123	21.89	8.66	2.52	0.87	0.43
12	15.99	5.52	2.89	0.63	0.31
13	17.21	6.45	2.66	0.68	0.34
23	10.57	4.97	2.12	0.49	0.21
1	11.31	3.68	3.07	0.45	0.22
2	4.68	2.21	2.11	0.22	0.09
3	5.89	2.76	2.13	0.27	0.11

Table 8.6: Risk of running a combination of GCBs on processing unit A or D

A configuration can then be defined to be a mapping of one or more GCBs on a processing unit. For example $conf_1 = (A, GCB_{123})$ is a configuration in which GCB_{123} is mapped on processing unit A. In this way the configurations of table 8.7 can be defined, when 4 processing units can be selected of two types A, B. This table only lists the risk, BOM an power of the mapping using the processing units. Interconnect cost and risk etc. are not taken into account.

$conf$	A1	D1	A2	D2	Total Risk	BOM	Power
1	123				0.87	10	20
2	123				0.43	40	35
3	12	3			0.63	50	60
4	13	2			0.68	50	60
5	23	1			0.49	50	60
6	3	12			0.31	50	60
7	2	13			0.34	50	60
8	1	23			0.45	50	60
9	1	2	3		0.45	60	60
10	1	3	2		0.45	60	80
11	2	1	3		0.27	60	80
12	3	2	1		0.45	60	80
13	2	3	1		0.45	60	80
14	3	1	2		0.27	60	80
15	1	2		3	0.45	90	100
16	1	3		2	0.45	90	100
17	2	1		3	0.22	90	100
18	3	2		1	0.27	90	100
19	2	3		1	0.22	90	100
20	3	1		2	0.27	90	100

Table 8.7: Different configurations $conf$ of processing units and GCB, and the bill of materials (BOM), Risk of implementing and power consumption of these configurations. The total risk is the maximum of the risk of running a GCB or a combination on the processing unit. The BOM is the sum of the cost of the used processing units, and the Power is the sum of the power of the used processing units in a configuration.

A pareto space can then be defined in which the Bill of Materials can be plotted versus the Risk involved with implementing this mapping. This results in figure 8.2. In this figure the pareto optimal points are connected by a red line. Another pareto space can be defined in which Risk, Cost and Power are compared. This results in a 3D graph (figure 8.2 that represents the pareto space. Pareto optimal points in this space are marked with a filled circle.

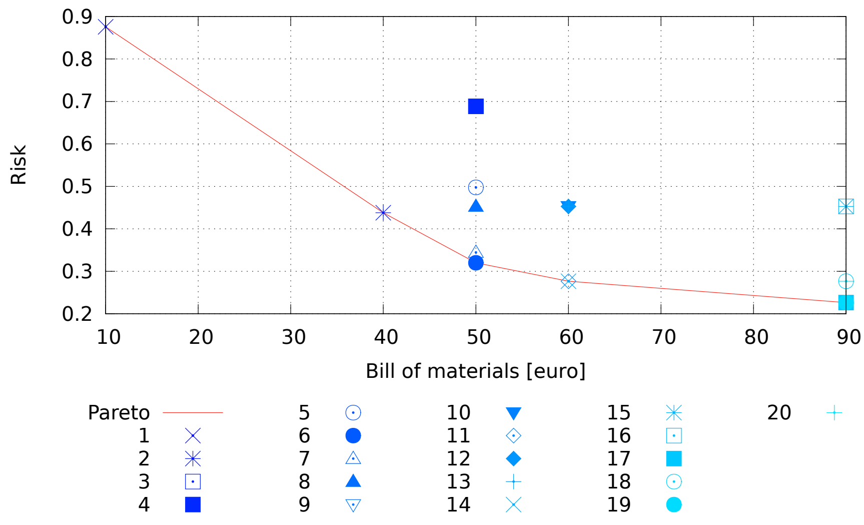


Figure 8.2: Pareto space of BOM vs Risk of the configurations

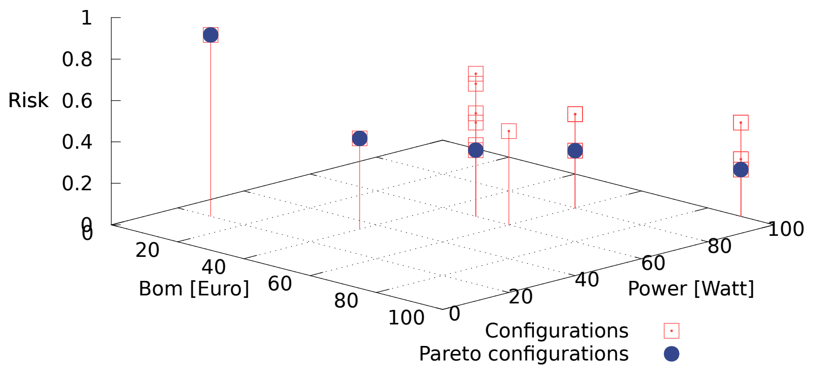


Figure 8.3: Pareto space of Risk, Power and BOM of the configurations

Chapter 9

Conclusions

The goal of this thesis is to provide a means for quickly estimating which processing units are needed for a set of tasks, given system requirements. This is delivered by the method presented in chapter 2. Furthermore this method can be applied as a guideline as discussed in chapter 8. The method consist of four steps:

1 Initially the application is decomposed into one or more algorithms.
2 The algorithms are then decomposed into one or more Generic Computational Blocks (GCBs). This are reusable constructs within an algorithm.
3 The GCBs are analysed for various properties, e.g., amount of operations and parallelism. These properties are used with the properties of the processing unit to create a model that provides an upper bound on the performance the GCB.
4 After modelling the processing units can be selected to form the embedded platform.

In order to perform the modelling, the Roofline model [40] is adopted to model the processing capacity of a processing unit in conjunction with the required processing of an application. This model provides a visual insight in the performance of a parallel architecture. The Roofline model describes performance in terms of floating point operations and external memory data transfers. Furthermore, it defines the model ceilings that illustrate different levels of parallelism and memory usage. Moreover the model depicts an application on a vertical line, where the position on the line is determined by the programming efficiency, hereby visualising how well the performance of an architecture is harnessed.The Roofline model is intended to assist programmers in reaching the maximum performance of a parallel architecture. In the current work, this model is extended to make it suitable for processing unit selection. The extended model has a number of benefits:

- The model incorporates multiple data sources, i.e., internal memory, external memory and interconnect. Whereas the original Roofline model only considers external memory transfers.
- Operations on multiple data types and a mix of these types can be modelled, instead of only modelling floating point operations like in the original model.

- In addition to the hardware based performance bound, named roof, a new bound is proposed. That is based on the ratio of different operations in an application and is named the utilization Roofline.

The extensions make the model better suited and more accurate for performance estimation of an application on a processing unit. In addition to the new model other contributions and observations of this thesis are summarised below:

- The performance of several processing units are modelled with the Roofline model. These models are verified using micro benchmarks to confirm the practical performance in the Roofline model. The verification in chapters 3, 4 and 5 show that computational performance can almost reach the computational roof, the achieved performance is on average 95% of the theoretical performance. The memory ceilings and roofs are harder to reach, the results of external memory show that the practical performance varies between 40% and 80%. This variation is due to the unknown internals of the memory controllers.
- This method does not require access to the processing unit in order to create the model, the essential information is acquired from its documentation. DSPs and FPGAs are in general well documented. The completeness of CPU documentation varies per vendor but is sufficient to construct a Roofline model. Documentation of GPU architectures is limited. This is expected to change in the future as GPUs will become more common for general purpose computations. The documentation is an important factor for the applicability and accuracy of the method.
- The information required to model an application in the Roofline model can be determined using a pen en paper approach or a high-level description of the application. Determining the amount of operations is the most challenging. Deviations between the estimated operations and the actual operations result in an error in the indicated performance. In section 7.3 and 9.1, several possible approaches are discussed which can be used to increase the accuracy of the estimation.
- Little effort and input are required to create an insightful model. This model can be used in an early stage of a project for a course selection. A more accurate model can be created in the case where there is a detailed description of processing unit and application available.
- The Roofline model can model the performance of a complete computational platform. However this provides less insight on performance bottlenecks compared to modelling individual processing units. This is overcome by modelling on both platform level as well as on processing unit level. The platform level presents an overview of the total capacity, while modelling on processing unit level allows the identification of local bottlenecks and imbalanced loads.

In addition to a fast selection the model offers a view on the possible risks concerning the processing units of choice. The Roofline model shows the required application performance with respect to the available performance. The smaller the difference between the required and available performance the higher the risk of failure. The Roofline model can also be used to find a trade-off between this risk ,which can be related to Non-Recurring Engineering (NRE) cost, and the Bill Of Materials (BOM). Using a pareto space a designer can select trade-offs and determine the desired mapping of GCBs onto processing units, based on performance predictions done by the extended model of this thesis.

9.1 Future work

The method presented in this work can be extended in some areas to increase the accuracy and usability. In this section five suggestions are discussed which can contribute to a higher usability and accuracy in terms of estimated performance.

i. As discussed in section 7.3, estimating the required operations and bytes for a GCB is a challenging task. Estimating the number of operations can be done on different levels of abstraction, e.g., pseudo code, Low Level Virtual Machine (LLVM) and assembly code. The LLVM method seems the most promising because it can generate an architecture independent low level code. LLVM is also able to perform architecture independent optimizations and automatic detection of loop trip counts. This makes it an useful approach to increase the accuracy of the estimated operations. Another advantage of the LLVM front-end is that it is able to process C, C++ and Java sources, which makes it possible for third parties to process their source code with the LLVM front-end and send the statistics to the electronics designer, hereby avoiding intellectual property issues.

ii. Embedded systems often use an Operating System (OS) for resource management and scheduling. This OS is most likely to execute on the CPU and requires a certain amount of memory bandwidth and processor time. If these requirements of the OS are known, then the OS can be placed in the Roofline model. This will yield a more accurate estimation of the available performance for an application.

iii. The abstract model of a processing unit introduced in chapter 2 comprises an internal memory, external memory and an interconnect. The interconnect can be of any type and kind, and it is assumed that the overhead of communicating via the interconnect is neglectable. However cases may exist where operations are require to communicate via the interconnect, e.g. in the case of an Ethernet interconnect where a TCP/IP stack has to run on the CPU to handle the communication. If the amount of operations are known then this can be modelled in the Roofline model to create a tighter bound on the available performance. Determining the required operations of Ethernet communication is addressed in the work of van der Oest[26].

iv. In chapter 6, a function is proposed that models the performance of an FPGA in terms of $[ops/sec]$ using the available resources in the FPGA. The performance indicated by the Roofline model for the FPGA should be verified using micro-benchmarks to increase the accuracy of the model. In addition to micro-benchmarks, some GCBs could be implemented on the FPGA. The performance of these implementations must then be compared with the predicted performance of the model. This will increase the accuracy of the Roofline for FPGAs

v. To improve the usability of the method for electronics designers, it is desired to have a tool-chain which can be used to easily and quickly create Roofline models. This tool-chain should also be able to place several GCBs in a model and calculate the utilization Roofline. This can also be combined with a source code analyser or a processing unit database.

Glossary

Ethernet physical transceiver is a chip that implements the hardware send and receive function of Ethernet frames, it interfaces to the line modulation at one end and binary packet signalling at the other. 58

geometric vertex is a point that describes the corners or intersections of geometric shapes. Vertices are commonly used in computer graphics to define the corners of surfaces in 3D models, where each point is given as a vector. 59

JPEG A acronym for Joint Photographic Experts Group that refers to a type of digital image compression.. 10

Mahalanobis distance is a measure for the similarity of an unknown sample set compared to a known one. 90

MP3 MPEG-2 Audio Layer III is a patented digital audio encoding format. 10

pipelining is the division of processing a instruction into a series of independent steps, with storage at the end of each step. This allows the control circuitry to issue instructions at the rate of the slowest step. 5

PKZIP Archiving tool originally written by Phil Katz and marketed by his company PKWARE Inc.. 10

system-type programs non numeric programs such as operating systems, compilers and editors. 3

TCP/IP The TCP/IP model describes a set of general design guidelines and implementations of specific networking protocols to enable computers to communicate over a network. 58, 119

texture is an image describing a surface, which mapped is on a shape, or polygon. 59

Acronyms

ALU Arithmetic Logic Unit. 30–32, 34, 39, 40, 43, 66

API Application Programming Interface. 59

BOM Bill Of Materials. 119

BRAM Block RAM. 75, 76, 78, 81

CAL Compute Abstract Layer. 66

CCL Connected Component Labeling. 93–95, 97

CFS Completely Fair Scheduling. 37

CISC Complex Instruction Set Computing. 34, 49

CLB Configurable Logic Block. 77

CPU Central Processing Unit. 5–7, 19, 21, 29, 30, 32–39, 44, 49–51, 55, 60, 76, 95, 102, 104, 108, 109, 118, 119

CUDA Compute Unified Device Architecture. 60, 62, 66

DDR Double Data Rate. 40, 75, 78, 79

DIMM Dual In-line Memory Module. 43, 75

DLP Data Level Parallelism. 85

DMA Direct-Memory Access. 58

DRAM Dynamic Random-Access Memory. 11, 16, 17

DSE Design Space Exploration. 2

DSP Digital Signal Processor. 5–7, 19, 37, 53–55, 58, 73, 78, 95, 108, 109, 118

ENO Estimated number of Operations. 86, 91–94, 96

FF Flip Flop. 71

FIFO First In First Out. 32, 37

FPGA Field Programmable Gate Array. 5–7, 12, 17, 19, 37, 71–81, 91, 93, 108, 109, 113, 118, 119

FPS Frames Per Second. 84, 90, 93

GCB Generic Computational Block. 7, 9, 10, 23, 24, 36, 70, 80, 81, 83–89, 91–96, 98, 100, 102–104, 108–115, 117, 119, 120

GCC GNU Compiller Collection. 88, 89

GPIO General Purpose Input-Output. 58, 72, 73, 75

GPU Graphical Processing Unit. 5–7, 19, 21, 37, 46, 59–70, 91, 93, 102, 104, 108, 109, 113, 118

ILP Instruction Level Parallelism. 61, 85

IO Input-Output. 16, 75

IOB Input-Output Buffer. 72, 73

ISA Instruction Set Architecture. 33–35, 49–51, 55

LB Logic Block. 71

LLVM Low Level Virtual Machine. 88, 89, 119

LRU Least Recently Used. 32

LUT Look Up Table. 71, 77

MAC Multiply-Accumulate. 64, 66

MIMD Multiple Instruction Multiple Data. 5, 6

MISD Multiple Instruction Single Data. 5

MoG Mixture of Gaussians. 90, 91, 97

MPSoC Multi Processor System On Chip. 16, 21

NOP No Operation. 54

NRE Non-Recurring Engineering. 119

OS Operating System. 37, 38, 50, 119

PCB Printed Circuit Board. 55

PCI-e Peripheral Component Interconnect Express. 75, 76, 78

POOSL Parallel Object-Oriented Specification Language. 4

PRU Programmable Real-time Unit. 58

PU Processing Unit. 9, 10, 15, 16, 20, 102, 114

RISC Reduced Instruction Set Computing. 33, 34

SDF Synchronous Data Flow. 5

SIMD Single Instruction Multiple Data. 5, 6, 15, 33, 43, 44, 49, 54, 55

SIMT Single Instruction Multiple Threads. 61

SISD Single Instruction Single Data. 5, 6

SoC System-On-a-Chip. 2, 48, 50, 55

SPEC Standard Performance Evaluation Corporation. 3

SPU simplified processing unit. 19, 21, 29, 34, 35, 72, 76, 79

TI Texas Instruments. 4, 53–57

TLP Thread Level Parallelism. 61, 85

VCA Video Content Analysis. 83, 84, 86

VLIW Very Long Instruction Word. 54, 55, 61, 62, 66

Bibliography

[1] Tom Vander Aa, Murali Jayapala, Francisco Barat, Henk Corporaal, Francky Catthoor, Geert Deconinck, *A High-level Memory Energy Estimator based on Reuse Distance* IMEC, Belgium and Electrical Engineering, TU Eindhoven 2005

[2] ARM *ARM Instruction Set Architecture* `http://www.arm.com/products/processors/technologies/instruction-set-architectures.php` August 2011

[3] Shuichi Asano, Tsutomu Maruyama and Yoshiki Yamaguchi *Performance Comparison of FPGA, GPU and CPU in Image Processing* Systems and Information Engineering, University of Tsukuba 2009

[4] Bammi, J.R. and Harcourt, E. and Kruitzer, W. and Lavagno, L. and Lazarescu, M.T. *Software performance estimation strategies in a system-level design tool* Cadence Design Systems, Philips Research Laboratories and Politecnico di Torino

[5] T. Bouwmans, F. El Baf and B. Vachon *Background modelling using mixture of gaussians for foreground detection - a survey*. Laboratoire MIA, Universite de La Rochelle, France 2008.

[6] W. Caarls, *Automated Design of Application-Specific Smart Camera Architectures* Delft University of Technology, 2008

[7] Henk Corporaal *Microprocessor Architectures, from VLIW to TTA* published by John Wiley & Sons 1997

[8] *Cortex-A9 NEON Media Processing Engine Revision: r3p0 Technical Reference Manual* `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0409g/DDI0409G_cortex_a9_neon_mpe_r3p0_trm.pdf`

[9] *C64x+ Cycle Accurate Simulator* `http://processors.wiki.ti.com/index.php/C64x%2B_Cycle_Accurate_Simulator`

[10] Lieven Eeckhout, Mark D. Hill *Computer Architecture Performance Evaluation Methods* 2010 ISBN: 9781608454679

[11] Oana Florescu, Menno de Hoon, Jeroen Voeten, Henk Corporaal *Performance Modelling and Analysis Using POOSL for an In-Car Navigation System* Eindhoven University of Technology, Chess Information Technology BV, Embedded Systems Institute 2006

[12] Marc Geilen, Twan Basten *A Calculator for Pareto Points* Department of Electrical Engineering, Eindhoven University of Technology 2007

[13] Tony Givargis, Frank Vahid, and Jörg Henkel *System-Level Exploration for Pareto-Optimal Configurations in Parameterized System-on-a-Chip* IEEE Transactions on Very Large Scale Integration (VLSI) Systems 2002

[14] Matthias Gries *Methods for Evaluating and Covering the Design Space during Early Design Development* Electronics Research Laboratory, University of California at Berkeley 2003

[15] John L. Hennessy and David A. Patterson *Computer Architecture - A Quantitative Approach (4. ed.)* Morgan Kaufmann Publishers 2007

[16] M.Holzer, F. Schumacher, I. Flores, T. Greiner and W. Rosenstiel *A real time video processing framework for hardware realization of neighbourhood operations with FPGAs* Center for Applied Research, Pforzheim University, Germany 2011.

[17] W. Hu, T. Tan, L. Wang and S. Maybank *A Survey on Visual Surveillance of Object Motion and Behaviors* School of Computer Science and Information Systems, Birkbeck College, London National Laboratory of Pattern Recognition, Institute of Automation, Chinese Academy of Sciences, Beijing 2003.

[18] *Intel® 64 and IA-32 Architectures Optimization Reference Manual* Order Number: 248966-025, June 2011

[19] O. Kalentev, A. Rai, S. Kemnitz, R. Schneider *Connected component labeling on a 2D gid using CUDA* Max-Plack-Institut fur Plasmphysik, Greifswald, Germany 2010

[20] Lina J. Karam, Ismail AlKamal, Alan Gatherer, Gene A. Frantz, David V. Anderson, and Brian L. Evans *Trends in Multicore DSP Platforms* IEEE Signal Processing Magazine, November 2009.

[21] V. Kastrinaki, M. Zervakis, K. Kalaitzakis *A survey of video processing techniques for traffic applications* Department of Electronics and Computer Engineering, Technical University of Crete 2003

[22] Marco Lattuada Fabrizio Ferrandi *Performance Modeling of Embedded Applications with Zero Architectural Knowledge* Politecnico di Milano, Dipartimento di Elettronica e Informazione 2010

[23] Paul Lieverse, Pieter van der Wolf, Ed Deprettere and Kees Vissers *A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems* Dept. of Information Technology and Systems, Delft University of Technology 2001

[24] C. Nugteren *Algorithm Classification and Performance Prediction for Heterogeneous and Parallel Computing* Eindhoven University of Technology 2011.

[25] Leonid Ryzhyk *The ARM architecture* `http://www.cse.unsw.edu.au/~cs9244/06/seminars/08-leonidr.pdf` June 2006.

[26] Bas van der Oest *10 Gigabit Ethernet and Rapid IO* Prodive, Son and Electrical Engineering, TU Eindhoven to appear

[27] Varun Sampath *CBench: Analyzing Compute Performance for Modern NVIDIA and AMD GPUs* University of Pennsylvania 2011

[28] A Senior, A Hampapur, YY Tian, L Brown, S Pankanti and R Bolle *Appearance Models for Occlusion Handling* IBM T. J. Watson Research Center, 2006.

[29] A. Shabbir, A. Kumar, S. Stuijk, B. Mesman, H. Corporaal *CA-MPSoC: An Automated Design Flow for Predictable Multi-processor Architectures for Multiple Applications* Eindhoven University of Technology Eindhoven, bNational University of Singapore 2010

[30] A. Shoham and J. Bier. *TI aims for floating point DSP lead* Microprocessor Report 12(12), September 1998.

[31] M.G.M Spierings and R.W.P.M. van de Voort, *Pre-study Document of DICAPO (distributed camera processing)* PSD6001101487R03.pdf , Prodrive B.V. Son 2011.

[32] C. Stauffer and W.E.L. Grimson *Adaptive background mixture models for real-time tracking* The Artificial Intelligence Laboratory, Massachusetts Institute of Technology 1999.

[33] W. Strauss *DSP Silicon Strategies '09* Forward concepts, `http://www.fwdconcepts.com/DSP\begingroup\let\relax\relax\endgroup[Pleaseinsert\PrerenderUnicode{âĂŹ}intopreamble]09` 2009.

[34] Edwin J. Tan and Wendi B. Heinzelman *DSP Architectures: Past, Present and Future* Department of Electrical and Computer Engineering, University of Rochester 2003.

[35] *Performance Modelling for System-Level Design* B.D. Theelen. Eindhoven University of Technology 2004

[36] *The Quintessential Linux Benchmark: All about the "BogoMips" number displayed when Linux boots* Linux Journal Volume 1996, Issue 21es, Jan. 1996

[37] TI *EDMA Background Activity for OMAP-L1x/C674x/AM1x Throughput Measurements* `http://processors.wiki.ti.com/index.php/EDMA_Background_Activity_for_OMAP-L1x/C674x/AM1x_Throughput_Measurements` March 2010

[38] J.A.Vijverberg and P.H.N de With *Hardware acceleration for tracking by computing low-order geometrical moments* Eindhoven University of Technology 2008.

[39] *An Overview of Common Benchmarks* Reinheld P. Weicker, Siemens Nixdorf Information Systems December 1990

[40] S.W. Williams, A. Waterman and D.A. Patterson *Roofline: an insightful visual performance model for floating-point programs and multicore architectures* Electrical Engineering and Computer Sciences, University of California at Berkeley 2008.

[41] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi and A. Moshovos *Demystifying GPU Microarchitecture through Microbenchmarking* Department of Electrical and Computer Engineering, University of Toronto 2010.