Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master Thesis
at the Department of Information Technology
and Electrical Engineering

Mapping Optimisation of Streaming Applications on Heterogeneous Platforms

FS 2014

Felix Wermelinger

Advisors: Lars Schor
Andreas Tretter
Professor: Prof. Dr. Lothar Thiele

Zurich
5th September 2014

Abstract

Due to physical limitations of processor speed, modern computing devices feature inherent parallelism. A modern computing architecture is often composed of multiple computing devices, each featuring a different degree of parallelism. This thesis analyses how a streaming application, specified as a Synchronous Data Flow graph, can be mapped onto a modern architecture, such that its performance is optimal. This mapping includes both the binding of processes to devices, as well as parallelisation parameters for each process. We use the throughput as the primary performance metric, while aiming to keep the latency within a given boundary.

Our general approach is to first have a calibration algorithm, which analyses the given application and architecture and extracts benchmark data characterising the execution of the given application and architecture. In a second step, this data can be used by an estimation algorithm, which can estimate throughput and latency for any given mapping. This estimation algorithm is suited to be used by a design space exploration algorithm in order to find an optimal mapping for a given architecture and application, while only requiring one initial execution of the calibration algorithm.

For synthesis and execution we use the Distributed Application Layer (DAL) framework. This framework allows to specify a streaming application as a Synchronous Data Flow graph and synthesises it onto various hardware devices, using the Open Computing Language (OpenCL) or POSIX threads. We develop profiling tools on top of this framework, which allow us to analyse throughput and latency of a network and each process within the network. We propose a model of how performance of a single process depends on its parallelisation parameters. We have developed a calibration algorithm, which will calibrate the process models for any given application and architecture. Experimental evaluation results match very well with the calibrated models. These calibrated process models proof to be useful for estimating the throughput and latency of the entire network, as we show by implementing such an estimation algorithm.

Acknowledgements

I would like to thank Prof. Dr. Lothar Thiele and the Computer Engineering Group for giving me the opportunity to write this master thesis as well as providing good working conditions.

Also I am grateful for the continued support of my advisors Andreas Tretter and Lars Schor, as we have spent many hours discussing the theoretical results of this thesis, as well as determining the focus of my work. Their advice and their knowledge of the research field at hand was a big help in writing this thesis.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Related Work . . . 2
1.3 Contributions . . . 2
1.4 Outline . . . 2

2 Background **5**
2.1 OpenCL . . . 5
2.2 Distributed Application Layer . . . 6

3 Problem & Approach **9**
3.1 Problem Description . . . 9
3.2 Approach . . . 10

4 Performance Evaluation Tools **13**
4.1 Profiling . . . 13
4.2 Analyser . . . 14
4.3 Summary . . . 15

5 Performance Models **17**
5.1 Process Execution Model . . . 17
5.1.1 Execution Time . . . 19
5.1.2 Transfer Time . . . 20
5.1.3 Time per Volley . . . 20
5.2 Process Network Model . . . 21
5.2.1 Process Network Specification . . . 21
5.2.2 Assumptions . . . 23
5.2.3 Steady State . . . 23
5.2.4 Throughput . . . 25
5.2.5 Latency . . . 28
5.3 Summary . . . 36

6 Experimental Evaluation **39**
6.1 Setup 39
6.1.1 Hardware 39
6.1.2 Application 40
6.2 Process Execution Model 40
6.2.1 Average Execution Time per Firing 41
6.2.2 Average Time per Firing 43
6.3 Summary 45

7 Conclusion and Outlook **47**
7.1 Conclusion 47
7.2 Outlook 48

A Appendix **49**
A.1 Presentation Slides 49

1 Introduction

1.1 Motivation

Due to physical constraints, the technology for processing devices has shifted towards parallel architectures. Hardware gets better optimized for its primary purpose over time and is nowadays developed with a specific use case in mind. The result is that a modern computer is a so called heterogeneous system, i.e. a system of multiple compute devices, usually a Central Processing Unit (CPU) and a Graphic Processing Unit (GPU). While most conventional programming is limited to the CPU, GPUs require a lot of effort to program efficiently, due to their complexity and the necessary manual handling of low level details. However recent developments have tried to make programming on GPUs. These developments are also starting to get a foothold in embedded computing, with new hardware which has GPU and CPU integrated on one chip.

In modern programs, which are focused on efficiency, it might be beneficial to move parts of an application to these other devices, as their parallel structure can be beneficial for some processes.

When implementing streaming applications for heterogeneous systems, it has been seen, that it is feasible to specify the application in form of a process network, i.e. a connected network of small programs, called "processes", streaming data to each other over fifo channels. While there exists a framework, which allows to map a process network onto a heterogeneous system, it has so far not been explored, how mapping of a given application can be

optimised. This thesis strives to discover how different parallelisation and mapping of separate processes to different devices affects performance.

1.2 Related Work

Even though throughput optimisation for heterogeneous systems has been considered before [1], so far the consensus was to assume a given constant performance metric for a given binding of a process to a device. This thesis however, will also consider that modern devices allow to parallelise a process and will use the parallelisation parameter as an additional design space dimension, which can be used for optimisation.

Latency has been considered previously as an optimisation goal [2] while optimising the scheduling of processes. This also included a calculation of the latency on a synchronous data flow, when using said scheduling. This thesis will also consider latency optimisation, but has to calculate latency using an undeterministic unchangeable scheduler.

1.3 Contributions

In this thesis we have made the following contributions:

- Development of profiling tools, which allow a programmer to analyse the performance of his mapped process network an a specific target architecture.
- Discovery of a mathematical model, which describes the performance of a process network mapped onto a heterogeneous system.
- Experimental evaluation of the model, by testing it against the results of practical test runs.

1.4 Outline

In Chapter 2 we will introduce the basics on which our thesis builds upon, namely the Distributed Application Layer (DAL) framework and the language used for implementation, Open Computing Language (OpenCL).

We specify our problem and what our approach to solve said problem looks like in Chapter 3.

Chapter 4 introduces the developed profiling tools, which allow to evaluate the performance of a mapped process network running using DAL.

In Chapter 5 we will try and model the behaviour of a process network, again with focus on the performance of said network.

In Chapter 6 we will test our models against some sample applications, comparing the measurements obtained through the profiling tools, with the models.

Lastly Chapter 7 will have a short conclusion, summarising the results of the thesis, as well as a short outlook, suggesting some possible future tasks within the field of this thesis.

2 Background

In this chapter we will review the OpenCL and DAL frameworks. These are the foundations of this thesis, since we use the DAL framework for performance evaluation and our mapping algorithm is designed with OpenCL-compatible systems in mind.

2.1 OpenCL

The OpenCL standard was developed by Khronos [3] in order to allow open, cross-platform, parallel programming for a variety of computing devices, such as CPUs and GPUs. Due to the parallelism implemented in modern architectures, the language has to supply a generic way to specify low level parallelism. This is even more important for Single Instruction Multiple Data (SIMD) parallelism, as it is used in GPUs. The OpenCL standard has done this in form of the two terms work groups and work items on the level of programming, while introducing compute units and processing elements on the hardware level. The idea is that a fixed set of processing elements is grouped as one SIMD compute unit, i.e. all processing elements within the compute unit can only execute the same operation at the same time. Each device usually contains several compute units.

On the programming level, we call the programs written for execution on OpenCL devices kernels. One work group is the set of all work items that get executed on one compute unit. One work item will execute the kernel

on one processing element within the compute unit that the work group is mapped to. As such, the work items within a work group normally run in SIMD fashion. However, since there can be more work items in a work group than there are processing elements in a compute unit, part of the work items must sometimes be stalled and be executed as soon as there are work groups free. This measn that it is not run in true SIMD fashion. The set of work items that get executed consecutively, form a so called wavefront.

The OpenCL framework is responsible for mapping work groups to compute units and work items to processing elements. The programmer can specify the number of work items and work groups as well as the device to map to for each kernel. However one has no control over which work group is mapped one which compute unit.

The hardware that does all of the invoking is called the host and is usually one CPU core. The host will invoke kernels by submitting them to a command queue. The device bound to said command queue will then process the kernels in order. The programmer may register a function as callback, such that it gets executed after the command has finished its execution. This can be used to then enqueue other kernels, which have dependencies on the finished one or to retrieve information about the finished kernel.

The OpenCL standard includes a profiling functionality, which allows to configure any OpenCL command to log important timing information, which can be obtained by registering a profiling callback function.

2.2 Distributed Application Layer

The DAL framework [4] has been developed in order to simplify the development of parallel computing software and to show that implementing parallelism implicitly by specifying a kernel as process network (see Figure 2.2 on page 7) is feasible, compared to the classic explicit method of parallel programming through multithreading. It has been shown, that specifying an application as process network can be beneficial, as the processes are inherently parallel and only coupled over the channels. Later the framework has been extended to be able to map these process networks on a heterogeneous system using OpenCL [5]. The framework will handle all low level details of memory management, scheduling and dependencies, while the programmer only has to supply the functional part of the separate processes in the process network. This allows to run different parts of the network in parallel on different hardware.

The framework requires a specification of the process network in simple XML-form, which specifies all processes and all channels in between them. The user then has to supply source code for the implementation of the pro-

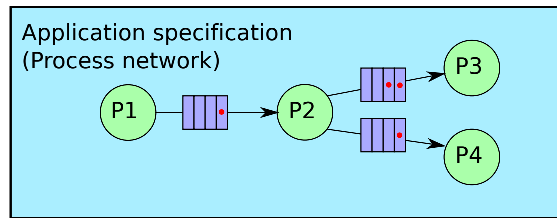


Figure 2.1: Example of a simple process network.

cesses themselves, while the framework will generate all code concerned with the data transfer and invocation management of processes. In the OpenCL environment, one process firing is implemented as one work group invocation. Because the overhead for scheduling and transferring data from and to the device can be quite large, the framework forms a collection of multiple firing operations, which get executed together, to have less relative overhead. We call such a collection a volley and it is implemented as a set of work groups implementing the same kernel. The programmer can specify the number of work groups which are grouped together in one volley for each process.

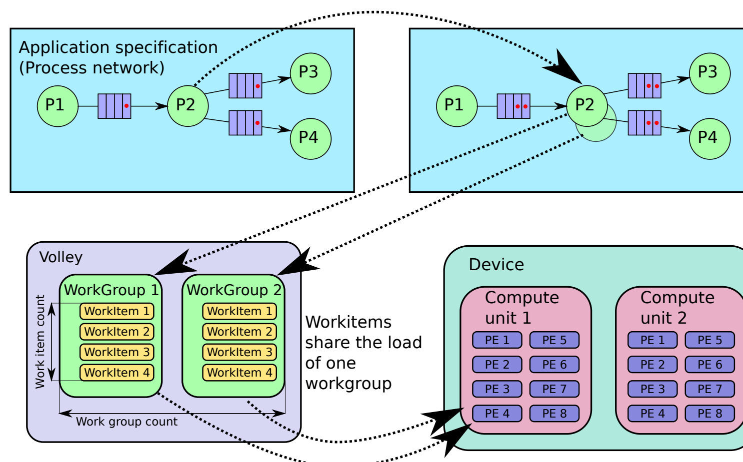


Figure 2.2: Illustration of how DAL maps a simple process network onto the OpenCL structure.

The channels, which have to transport data in between different processes

are implemented as fifos with OpenCL kernels explicitly handling the synchronisation of the fifo content over multiple devices. These kernels are called push and pop kernels and get enqueued in the same fashion as the kernels of volleys. The framework ensures correct ordering of their execution.

The framework also needs an architecture file, detailing the available hardware devices, as well as a mapping file, which statically maps processes to devices and parametrizes the process code itself. These files have to be generated manually. Note that in this context mapping includes the binding of a process to a device as well as setting the paralellisation parameters. This means that the mapping specifies the nubmer of work groups within a volley as well as the number of work items within a work group.

3 Problem & Approach

In this chapter we will describe the problem our thesis is trying to solve. We also detail our general idea and the approach of our solution.

3.1 Problem Description

This thesis has used the DAL framework in order to analyze how the mapping can be used to optimize the performance of a specified process network. The first goal was to extend the DAL framework to analyze the performance of a process network and to enhance our understanding which parameters affect the performance. We are interested in the throughput that a mapped process network may achieve, while keeping the latency bounded. The idea is to develop an estimation, which can predict the performance of a specific mapping for a given process network. This allows the use of an evolutionary algorithm, which will find the Pareto-front using the given estimation.

In order to achieve this it was imperative to develop a performance model of a process network, which captures the architecture, the mapping and the parallelization of the application, and shows how these factors influence the performance.

3.2 Approach

Our approach to obtain a performance estimate starts by calibrating our models for the given application and architecture. We call the collection of our calibrated models the *performance characteristics table*. With the calibrated models, we can then run an estimation algorithm, which estimates the performance for any given mapping. Such an estimation could then be used by a design space exploration algorithm, which is then able to find an optimal mapping without having to run tests for each mapping. This concept can be seen in Figure 3.1.

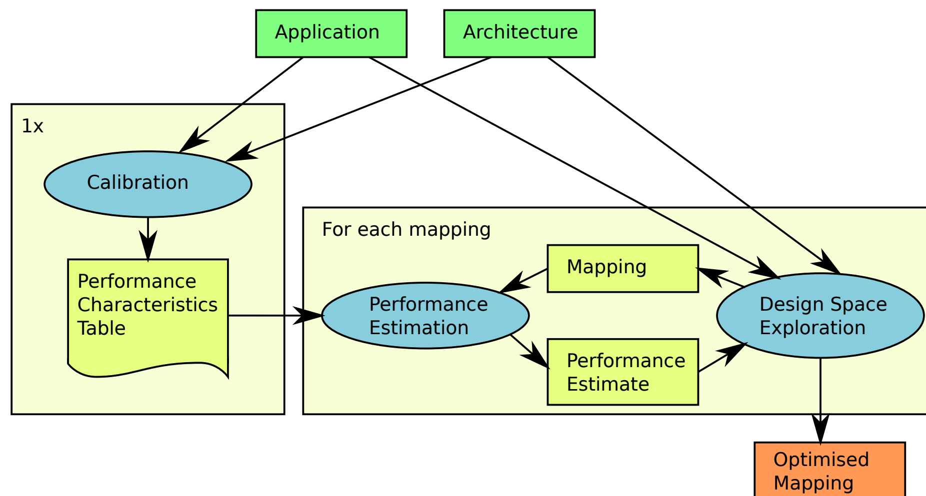


Figure 3.1: Illustration of our approach on how to obtain a reasonable estimate of the performance of a given application and architecture as well as an overall concept on how to find an optimised mapping using said performance estimation.

For calibration we developed profiling tools, which extract runtime information from a running process network. We focused on obtaining measurements meaningful for throughput and latency, as these are the measures we were later interested in. The important measures are the time a process takes to execute a firing and how long it takes until the process may execute the next firing, as these measures are then later useful for determining the overall throughput and latency of the process network.

For obtaining our models, we analysed the behaviour of processes separately at first. The idea was to develop a mathematical model of the measures mentioned above, which captures their dependencies on all factors, such as

the binding and the parallelization of the process. However there remain some parameters which are normally not known, since they depend on the device architecture, the exact source code or the driver implementation. To obtain these, we execute a small set of tests, using different mappings and parallelizations, to obtain some measures with our profiling tools. Now that we know some measures, we can fit our model to the measured values and find the previously unknown parameters. We call this step the calibration of the models.

After this is done for all processes in the network, we have to concern ourselves, how the overall performance is affected by the separate processes. We model the behaviour of the process network as a whole, based on the individual process models. In this model there is no additional calibration necessary, as it only depends on the performance of the separate processes. So the calibrated process models in combination with our process network model, allows us to predict the performance of a given mapped process network.

4
Performance Evaluation Tools

In this chapter we will introduce the developed profiling tools, which were used for evaluating the performance of a process network running on the DAL framework. First we have a look at the profiling, which extracts some raw timing information from each process and then we will look at the analyzer, which interprets the data from profiling and extracts meaningful measures. We will then use these tools later to calibrate our model of the process network performance.

4.1 Profiling

The profiling tools were integrated directly into the DAL framework (as shown in Figure 4.1 on page 14). When running a process network using DAL, each OpenCL process in the process network has its own profiler, which extracts timing values (see Table 4.1 on page 14) for each volley of said process. OpenCL supports the extraction of the values as listed in Table 4.1 on page 14. Figure 4.2 on page 15 illustrates the timing values which can be obtained.

In this implementation, a volley is always executed as a group, so even though some firings can be finished earlier, they have to wait for the last one to finish, before informing the host about their termination.

Note, that the time in between submit and start will only transfer commands and execution parameters and not the actual data to process. The transfer-

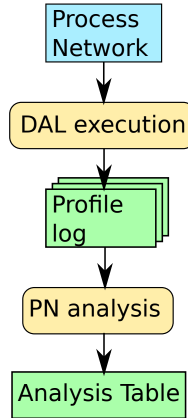


Figure 4.1: Illustration of how the profiling of a process network works.

ring of data from host to device and vice versa is done in a separate kernel, which has again the four timing values. This is called the pop operation, as it pops the tokens from a channel fifo onto the device to use in the next firing. In the same manner there is also a push operation, which will push the tokens from the device onto a channel fifo.

Name	**Value in nanoseconds**
QUEUED	Time when the command is enqueued into a command queue on the host
SUBMIT	Time when the command leaves the host and is submitted to the device
START	Time when the command starts execution on the device
END	Time when the command ends execution on the device

Table 4.1: Table of the timing values that are extracted by the profiler

For accurate profiling, we will simulate a process network for a long time and each profiler will accumulate a long profiling log, which contains all the timing values from Table 4.1 of the respective process. These profiling logs can then be interpreted by the analyser.

4.2 Analyser

The analyser is a modular implemented program, which consists of a multitude of measurers (illustrated in Figure 4.3 on page 16), which are tasked with interpreting the profile logs from the profiling and extracting some meaningful measurements, characterising the execution of the process net-

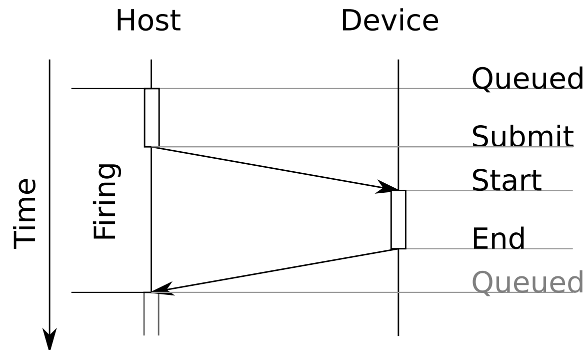


Figure 4.2: Illustration of the points in time during execution, that the profiler allows us to extract. Note that the scale is not representative and that some timings might be much larger than others.

work. These measurers can either be process-measurers, which extract a reasonable measure about a single process (e.g. the average execution time needed for one firing of said process), or a network-measurer, which tries to extract a characteristic about the process-network as a whole by interpreting data about all processes (e.g. the overall latency). There exists a separate process measurer instance for each process, while there exists only one instance of each process-network measurer. The results are saved in analysis tables which are then appended together. The tables are designed to allow multiple measurements from different test runs to be accumulated in a single analysis table.

4.3 Summary

In this chapter we have shown, which timing values the profiler can provide us with. We have also seen how the analyser can interpret the profiling data in order to produce measures, which characterise the execution of the profiled process network.

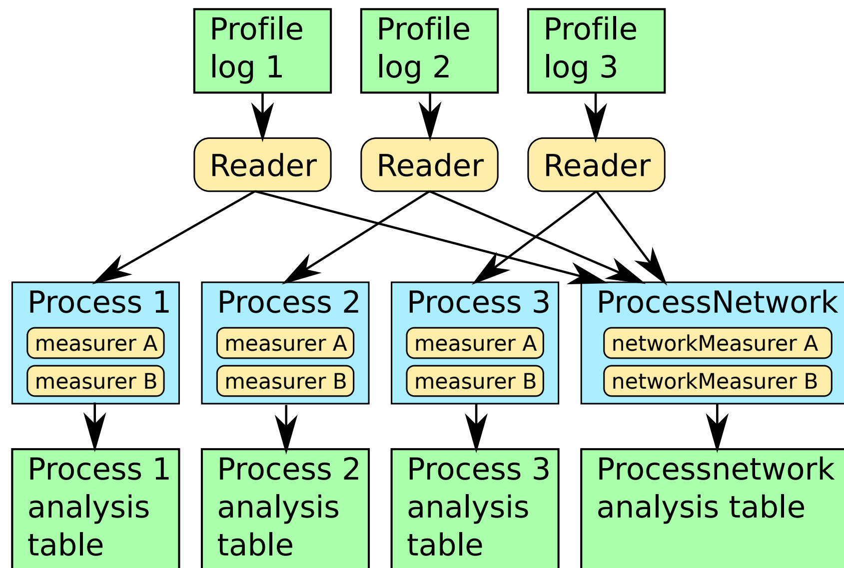


Figure 4.3: Concept of the analyser, that processes the profiling logs.

5 Performance Models

In this chapter we will develop a model of how a process network performs, when executed using the DAL framework. Our goal is to model the overall throughput of a process network. This depends on the network specification, the mapping and the hardware architecture. In a first step we develop a general process model, which models the throughput of a single process. In a second step we will analyse the overall throughput of the network as well as the latency, i.e. the time that a token needs to traverse the network.

The idea is to get a mathematical expression of how performance depends on various parameters, including some unknown constants, which depend on the drivers, source code or hardware architecture. These unknown constants cannot be broken down further, as in depth knowledge of drivers and hardware would be necessary. However we can calibrate these models by running a network multiple times, using different parametrisations (different choices for number of work groups, number of work items and mapping of each process) and use our performance evaluation tools (chapter 4 on page 13) to obtain the performance of these runs. As such we can fit our models to the obtained measurements.

5.1 Process Execution Model

In the following we will model the execution of a single process with a focus on its performance, explicitly its throughput. This means we are interested in how long it takes to execute said process.

As discovered in [5] the execution model looks like Figure 5.1. Each volley has to be accompanied by push and pop operations, which transfer the tokens from and to the device. As illustrated, the total time for each volley can be split in 2 distinguishable parts:

$$t_{pV} = t_{transfer} + t_{execution} \tag{5.1}$$

Where the time per volley t_{pV} is given as the sum of the transfer time $t_{transfer}$, the time needed to transfer the commands and tokens to and from the device, and the execution time $t_{execution}$, the time spent actually executing the volley on the device.

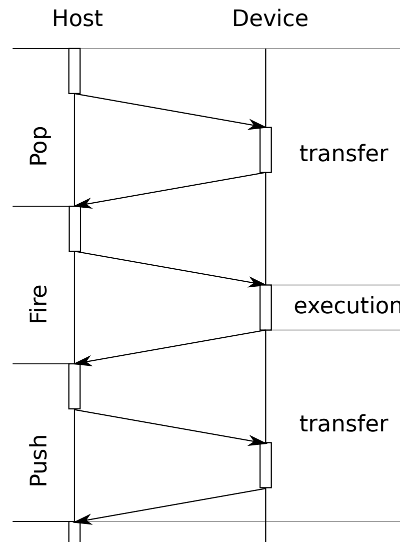


Figure 5.1: The used execution model. Note that the size relations are not representative and that some of these timings can be negligibly small compared to others.

Assumption 5.1.1. *We neglect the time that push and pop operations need to execute on the target device.*

Reasoning. Experiments have shown, that these times are always very small in comparison to execution times of the firing operation and other timing values. Since it makes our upcoming equations easier to understand we will neglect these.

In the following we study different factors which influence the timing values in our model. It is our goal to develop a mathematical description of the model, which can then later be used to develop a model for the whole process

network. For the sake of simplicity we will first only create a model for $t_{execution}$ and $t_{transfer}$ to then combine these two into a model for t_{pV}.

5.1.1 Execution Time

By only looking at the execution time $t_{execution}$ of a volley we can develop a fairly simple model. We can split up the execution time further into the constant overhead $t_{overhead}$ for memory and binding management on the device and the actual time used for calculation $t_{calculation}$.

$$t_{execution} = t_{calculation} + t_{overhead} \tag{5.2}$$

To determine $t_{calculation}$ we have to first consider all parallel work groups. These work groups are executed in parallel, as long as there are enough compute units available, otherwise a serialisation into $\left\lceil \frac{\#WG}{\#CU} \right\rceil$ sequential executions is necessary. The calculation time of a single work group is defined as t_{CpWG}.

$$t_{calculation} = \left\lceil \frac{\#WG}{\#CU} \right\rceil \cdot t_{CpWG} \tag{5.3}$$

The calculation of a work group is then parallelised in SIMD fashion onto different work items. Again executing in parallel as long as there are enough processing elements available. Should there be less processing elements available than there are work items, then $\left\lceil \frac{\#WI}{\#PE} \right\rceil$ wavefronts are formed, which are executed sequentially within the work group.

$$t_{CpWG} = \left\lceil \frac{\#WI}{\#PE} \right\rceil \cdot t_{CpWI} \tag{5.4}$$

We assume, that the process is implemented using a loop, iterating multiple times over the same source code. Since the number of iterations per work group $\#I_{pWG}$ must be constant for a given source code, the number of iterations per work item are given as $\left\lceil \frac{\#I_{pWG}}{\#WI} \right\rceil$. Additionally there is usually a constant overhead for memory accesses, called t_{mem}. Thus, the calculation time of a work item t_{CpWI} can be calculated depending on the calculation time per iteration t_{CpI}.

$$t_{CpWI} = \left\lceil \frac{\#I_{pWG}}{\#WI} \right\rceil \cdot t_{CpI} + t_{mem} \tag{5.5}$$

Both the calculation time per iteration t_{CpI} and the memory overhead t_{mem} now only depend on the source code of the process, the device it is run on and

the tokens processed. These values cannot be further broken down without intricate knowledge of the used hard- and software.

If we re-substitute the above equations into Equation (5.2) on page 19, then the resulting execution time model is given as

$$t_{execution} = \left\lceil \frac{\#WI}{\#PE} \right\rceil \cdot \left\lceil \frac{\#WG}{\#CU} \right\rceil \cdot \left(\left\lceil \frac{\#I_{pWG}}{\#WI} \right\rceil \cdot t_{CpI} + t_{mem} \right) + t_{overhead} \ . \tag{5.6}$$

This means, that if we later want to estimate the execution time $t_{execution}$ we only have to measure t_{CpI} and $t_{overhead}$. This is easy, since these values only depend on the source code and the device the process is bound to. Thus we only have to measure these values once and we can then extrapolate the execution time for all possible choices of number of work groups $\#WG$ and number of work items $\#WI$.

5.1.2 Transfer Time

The transfer time $t_{transfer}$ scales primarily with the number of tokens transferred, since each token has to be read, transmitted over the bus and stored. The amount of tokens is proportional to the amount of work groups used, since the amount of tokens processed by a single work group is given by a constant. The amount of work groups increases transmit time linearly, because we need more tokens for more work groups. However we also have a constant part needed for setting up communication, compiling kernels and scheduling on the device and the host. We call this overhead $t_{overhead2}$.

$$t_{transmit} = t_{TpWG} \cdot \#WG + t_{overhead2} \tag{5.7}$$

Where t_{TpWG} is the transfer time per work group, which can be estimated as a constant for each process and device, since the amount of tokens per work group is constant. $t_{overhead2}$ is again assumed to be a constant. The values for t_{TpWG} and $t_{overhead2}$ can be obtained using our developed profiling tools (chapter 4 on page 13).

5.1.3 Time per Volley

For our calculations of the time per volley t_{pV} we consider again Equation (5.1) on page 18:

$$t_{pV} = t_{transfer} + t_{execution} \tag{5.1 revisited}$$

By just substituting Equations (5.6) and (5.7) we get our final model in Equation (5.8) on page 21:

$$t_{pV} = \left\lceil \frac{\#WI}{\#PE} \right\rceil \cdot \left\lceil \frac{\#WG}{\#CU} \right\rceil \cdot \left\lceil \frac{\#I_{pWG}}{\#WI} \right\rceil \cdot t_{CpI} + \left\lceil \frac{\#WI}{\#PE} \right\rceil \cdot \left\lceil \frac{\#WG}{\#CU} \right\rceil \cdot t_{mem} + t_{overhead} + t_{TpWG} \cdot \#WG + t_{overhead2} \quad (5.8)$$

Note that the constant $t_{overhead2}$ can be absorbed into $t_{overhead}$, when trying to calibrate this model.

5.2 Process Network Model

In this section we define our model of a process network. We will then determine the best achievable throughput and the worst case latency of the process network. Later we will need these metrics for our mapping algorithm, since it strives to maximise throughput, while keeping the worst case latency bounded.

5.2.1 Process Network Specification

We define our process network as the dataflowgraph $g = \langle P, C \rangle$ with the process set P and the set of channels C. Each process $p \in P$ is defined as $p = \langle I_p, O_p \rangle | I_p \subseteq C, O_p \subseteq C$, having an associated set of input channels I_p and a set of output channels O_p. Each channel $c \in C$ is defined as $c = \langle i_c, o_c, c_c \rangle | i_c, o_c, c_c \in \mathbb{N}, i_c \leq c_c, o_c \leq c_c$ where i_c and o_c denote the input- and output firerate of the channel in bytes, while c_c represents the capacity, i.e. the number of bytes the channel can hold at a time. The channel is assumed to be implemented as fifo channel, this means, that no token may leave the channel until all tokens that have arrived before this token have also left.

We model our architecture specification as $a = \langle D \rangle$ where the device set D consists of all devices $d \in D$ available in the architecture. An exact model would also contain the communication-busses, which connect these devices, but we will neglect their resource limitations in this thesis.

We now define the mapping, which binds each process of the process network to a device of the architecture: $d = m(p) : P \to D$

A schedule $\mathbf{s}$ is an ordered list of n process firings $\mathbf{s} \in P^n, n \in \mathbb{Z}^+$. The ordering represents the order of the processes finishing. i.e. s_i finishes before s_k if and only if $i < k | i, k \in \mathbb{Z}^+$. We are only interested in*admissible* schedules, that means schedules, which are executable without violating any

constraints of resources or our assumptions. This means, that no process firing may overlap with another firing from the same process, the resource constraints of the devices have to be fulfilled and no channels may overflow or underflow with tokens at any time during the execution.

Definition 5.2.1. *We define a process firing to be* ready*, if there is no other firing of the same process scheduled, there are enough tokens on all input channels and there is enough free space on all output channels. A firing may only be scheduled to be processed after it has reached ready status.*

We define the buffer state as a ordered list $\mathbf{b} \in \mathbb{N}^{|C|}$, where b_i, $0 < i \leq |C|$, denotes the number of bytes contained within the i-th channel. We introduce the shorthand b_c as the current amount of bytes stored within channel $c \in C$.

When developing models for each separate process in Sections 5.1.1 and 5.1.3 on page 19 and on page 20, we discussed two separate measures, time per firing t_{pF} and execution time per firing t_{epF}, for each process on each device. We formalise this as the functions $t_{pF}(p, d) : P, D \rightarrow \mathbb{R}^+$ and $t_{epF}(p, d) : P, D \rightarrow \mathbb{R}^+$

As suggested in [6] we introduce the topology matrix Γ, where the entry $\Gamma_{i,j}$ represents the amount of tokens that a firing of process j reads from/writes to channel i (reading being represented by negative numbers). We define the *firing vector* $q = \{q_1, 1_2, \cdots, q_{|P|}\}$ as the positive normalised solution to $\Gamma q = \mathbf{0} | \; ||q||_2 = 1$, with q_p denoting the normalised firerate of process p . If each process p in the network fires q_p times, then all channels in the network will hold the same amount of tokens before and after the firings. We call the collection of these firings a *round* of a *PAPS* (periodic admissible parallel schedule). We can express each paps round as a scaled version of the firing vector $q \cdot l$.

The throughput of a process will be specified by the function $\theta : P \rightarrow \mathbb{R}^{+|P|}$, which maps each process to a rational number representing the number of firings per time this process is executed.

Definition 5.2.2. *We define the throughput of a process as the number of firings per second of said process.*

From theory we know, that knowing the amount of firings of one process per time, implies knowledge about the firings of any other process, since the firerates of processes are coupled over the firing vector q in a PAPS round.

Thus we can specify the throughput of our process network as the function $\theta(p)$:

$$\theta(p) = q_p \cdot l \mid l \in \mathbb{R}^+ \tag{5.9}$$

where l is the scalar optimisation factor.

5.2.2 Assumptions

In this thesis we have only confronted ourselves with a subset of process networks. Thus our analysis will reside on the fulfilment of the following properties:

Assumption 5.2.1. *The process network is loop and state free, this means that the execution of a process firing only depends on its current input tokens.*

Assumption 5.2.2. *The process network is connected.*

Reasoning. It is easy to extend our arguments to unconnected networks, by viewing each component as its own process network. The only interaction is that processes from different components might be bound to the same device, which would essentially transform the problem to a multi objective problem, where optimising throughput of one component might decrease the one of the other component. However we will not concern ourselves with this.

Assumption 5.2.3. *The execution time of a process is largely independent of the tokens it reads, or if not, the deviation of the execution time is small enough that arguing with the average time is sufficient.*

Assumption 5.2.4. *The scheduler is fair. All firings which are executed on the same device will be enqueued and executed in FIFO fashion.*

Assumption 5.2.5. *The network is run for a long time. We do not pay special attention to effects when starting up or shutting down the network and are also not interested in the notion of initial tokens on channels.*

Assumption 5.2.6. *The communication capacity in between devices is never fully utilised and does not constrain the throughput of the application.*

Assumption 5.2.7. *The channel capacity c_c of a channel c is divisible by its inputrate i_c and outputrate o_c.*

Reasoning. Channel capacities which do not fulfil this assumption would be less efficient, since the full capacities could not be utilised. This is also the reason why the DAL framework requires this property.

5.2.3 Steady State

In this section we model the steady state, which is the state of the network after running the network long enough.

After starting the network, it will converge towards a specific steady state. This is because all processes will always be scheduled as soon as they are

ready and eventually a network will always be throttled by at least one bottleneck. The throttling will be distributed over the network, because a throttled process will not consume enough tokens for its unthrottled predecessors and not generate enough tokens for its unthrottled successors. Thus eventually these neighbouring processes will also be throttled (because they cannot receive enough tokens or do not have enough space to write their tokens to), in turn eventually throttling their neighbours and so on. Once the whole network has reached a state, where all processes are throttled, such that we can run the network infinitely long without ever needing to make adjustments to the speed of processes, we are in the *steady state*.

Definition 5.2.3. *The* steady state *is a state of the network, where the processes have throttled each other, such that the network can be executed indefinitely.*

Assumption 5.2.8. *Assuming a network is in the steady state, then letting the network run for an arbitrary amount of time will preserve the current buffer state.*

Reasoning. In the steady state we assume the firing rates to be constant, thus all channels will have exactly as many tokens written to them over time as read, preserving the buffer state.

Definition 5.2.4. *A* bottleneck *process is a process, which limits the overall throughput of the process network in the steady state. This can be either due to its maximal achievable firing rate or due to resource limitations on a device.*

Note, that there may be multiple bottenecks within a process network in the steady state. This is due to definition 5.2.4, which defines, that a process may become a bottleneck due to the device it is bound to. If multiple processes share a device and this device's resource limitations are fully utilised, then this implies, that all processes bound to said device must be bottlenecks.

Definition 5.2.5. *A process p is a* predecessor *of process r in the loop-free network graph $g = \langle P, C \rangle$ if and only if there exists a directed path from p to r. Vice versa, r is called the* successor *of p.* Direct successors *and* direct predecessors *are processes, which are immediately adjacent to the respective processes.*

Definition 5.2.6. *A channel is called* drained *if it has not enough tokens on it in the steady state to allow the process, which reads from the channel, to fire.*

This means, that the process writing to the channel cannot provide enough tokens for the reading process to be executed as much as the reading process

would be able to. This will force the reading process to throttle its firing rate to only consume as many tokens per time as the writer provides.

Drained channels generally occur after bottlenecks, since the bottlenecks are throttled and their successors can consume more tokens than the bottlenecks can produce. However they can also occur at other locations in the network, as illustrated in Figure 5.2.

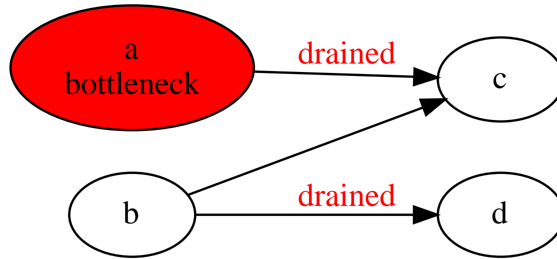


Figure 5.2: Illustration of the buffer states of a steady state. Bottleneck a throttles process c by not providing enough tokens for it to consume. In turn process c throttles process b by not reading as much tokens per time as b writes and thus process b cannot write as many tokens as process d would be able to consume, draining the channel in between b and d.

5.2.4 Throughput

In the following we try to find the best achievable throughput of a given process network, given dataflow graph g, architecture a and mapping $m(p)$. We may also use the obtained measures from the process models t_{pF} and t_{epF}. We will later use this knowledge to find an optimal mapping for a given architecture and dataflow graph.

We want to choose the scalar optimization factor l as large as possible, but we have some limitations:

1. Each process p has a maximal achievable firing rate $\dfrac{1}{t_{pF}(d,p)}$ on each device d, which determines how often the process can run per time. This is because a new firing of the same process can only be enqueued after the last one has terminated and $t_{pF}(d,p)$ denotes the time needed for one firing if we use the best case scenario, where we always enqueue the next firing as soon as the last finished. Thus this fire rate cannot be exceeded.

2. The device resources are finite and a process may have to wait for other processes to finish, to access a device.

3. The bandwidth to the device is limited. The transfer of data and commands of one process to the device may be stalled by other transfers.

As we have already mentioned in the model, we assume, that the bandwidth limit is never reached. Thus we have to only consider optimising l for the two former cases:

Case 1 We know the maximal achievable average time for a firing of a process $p \in P$ as $t_{pF}(p, d)$. This measure cannot be exceeded by the actual firing rate:

$$q_p \cdot l \leq \frac{1}{t_{pF}(p, m(p))} | \forall p \in P \tag{5.10}$$

optimising l yields

$$l = \min_{p \in P} \frac{1}{q_p \cdot t_{pF}(p, m(p))} \tag{5.11}$$

Case 2 We define the utilisation $U(d)$ of a device $d \in D$ as the percentage of time that the device is processing tasks. We can calculate the utilisation by summing up how much time each process bound to the device needs. The time needed by one process we call the usage of the process on the device. We know the amount of firings per second of a process p as $q_p \cdot l$ and the amount of execution time spent per firing as $t_{epF}(p, m(p))$ yielding $Usage(p, d) = q_p \cdot l \cdot t_{epF}(p, d)$ and thus also Equation (5.12).

$$U(d) = \sum_{p \in P | m(p) = d} q_p \cdot l \cdot t_{epF}(p, d) \tag{5.12}$$

By definition we have to fulfil that the utilisation may not be larger than 100%:

$$U(d) = \sum_{p \in R} q_p \cdot l \cdot t_{epF}(p, d) \leq 1 | \forall d \in D \tag{5.13}$$

Where R is the set of processes bound to device d.

After factoring l out and optimising l this yields:

$$l = \min_{d \in D} \frac{1}{\sum_{p \in P | m(p) = d} q_p \cdot t_{epF}(p, d)} \tag{5.14}$$

In this case all processes mapped to device d are bottlenecks of our mapped network.

Solution So our problem can be written as the maximisation problem of l such that Equation (5.10) on page 26 and Equation (5.13) on page 26 hold. For the maximal l, we can just unify Equation (5.11) on page 26 and Equation (5.14) on page 26 by looking which boundary condition is limiting. We do this by taking the minimum of all boundaries once more, yielding:

$$l = \min\left(\min_{p \in P} \frac{1}{q_p \cdot t_{pF}(p, m(p))}, \min_{d \in D} \frac{1}{\sum\limits_{p \in R} q_p \cdot t_{epF}(p, m(p))}\right) \tag{5.15}$$

where R is the set of processes bound to device d.

Note On our testing environment we have some additional considerations to make:

- In our execution, firings are grouped together into volleys. However volleys in our environment have the same properties as firings do in general graph theory. One has to simply adjust all aforementioned formulas to use volleys as firings.
- A volley does not necessarily occupy all resources of a device at a time. This means that there may be multiple processes running on the same time simultaneously. The usage of the device is scaled by factor $\frac{\#WG}{\#CU}$, with $\#CU$ the number of compute units which are available to execute one work group, and $\#WG$ the number of work groups to be executed. The other compute units remain available.

  However this works only for $\#WG \leq \#CU$. The correct modifier in general is $\frac{\#WG}{\left\lceil \frac{\#WG}{\#CU} \right\rceil \cdot \#CU}$. This is because $\left\lceil \frac{\#WG}{\#CU} \right\rceil \cdot \#CU$ are the amount of work groups which could be processed during the execution time (as many as there are work groups rounded up to the next multiple of $\#CU$), while the firing actually uses only $\#WG$ many.

  This changes Equation (5.13) on page 26 to Equation (5.16) and also changes Equation (5.15) to Equation (5.17):

$$U(d) = \sum_{p \in R} q_p \cdot l \cdot t_{epF}(p, d) \cdot \frac{\#WG(p)}{\left\lceil \frac{\#WG(p)}{\#CU(d)} \right\rceil \cdot \#CU(d)} \leq 1 | \forall d \in D \tag{5.16}$$

$$l = \min\left(\min_{p \in P} \frac{1}{q_p \cdot t_{pF}(p, m(p))}, \min_{d \in D} \frac{\left\lceil \frac{\#WG(p)}{\#CU(d)} \right\rceil \cdot \#CU(d)}{\sum\limits_{p \in R} q_p \cdot t_{epF}(p, m(p)) \cdot \#WG(p)}\right) \tag{5.17}$$

where R is the set of processes which are bounded to device d.

5.2.5 Latency

In the following we try to estimate the latency in the system. These findings can later be used to find the Pareto front for different mappings optimising throughput and latency.

Definition 5.2.7. *The latency $lat(s,t)$ of a process pair (s,t) is defined as the time difference from a token being generated at the source s, just after it has been generated until the last tokens which were generated depending on said token have been consumed by the target t.*

Definition 5.2.8. *The latency $lat(g)$ of a process network specified by data-flow graph $g = \langle P, C \rangle$ is defined as the longest time any token spends on the network and is mathematically defined as*

$$lat(g) := \max_{s,t \in P} lat(s,t) + t_{pF}(s) \tag{5.18}$$

For the overall latency we have to add the time that the source process s needs to produce the token, as its existence starts at the beginning of its execution.

We are interested in the latency of the system in the steady state, i.e. we neglect any effects that may occur when starting the network up.

To solve this problem, we will first solve an easier sub-problem and then try to generalise the solution. Thus we will for now assume, that our process network has only 1 bottleneck process and that the token in- and output rates are 1 for all channels.

Definition 5.2.9. *$tok_\psi(s,t)$ is the amount of tokens that process t has to consume in the steady state to empty all channels on a path ψ connecting s and t if s never fires.*

Assumption 5.2.9. *The latency $lat_\psi(s,t)$ in between two processes s and t connected over a path ψ can be calculated as*

$$lat_\psi(s,t) = tok(s,t) \frac{1}{q_t \cdot l}$$

if the last channel c on path ψ is not drained.

Reasoning. Because we are in the steady state, we know that the average firing rates are defined by the firing vector $q \cdot l$. Since we assume, that channels are implemented as fifos and never empty, we know, that process t has to process all other tokens in between s and t before reading anything new. Because we know the tokenrate to be 1 token per firing and that t has a

firing rate of $\frac{1}{q_t \cdot l}$ firings per second, we know that process t processes $\frac{1}{q_t \cdot l}$ tokens per second. So to obtain the time needed to process $tok_\psi(s,t)$ tokens is given as $lat_\psi(s,t) = tok(s,t)\frac{1}{q_t \cdot l}$. This counts the time from when the token exists on the very first channel leaving s until it has been processed by t.

Assumption 5.2.10. *The latency $lat_\psi(s,t)$ in between two processes s,t connected over a path ψ can be lower bounded by*

$$lat_\psi(s,t) = \sum_{p \in \Pi} t_{pF}(p)$$

if the channels on ψ are all drained. Where Π denotes the set of processes along the path ψ including t but excluding s.

Reasoning. If the channels are drained, then the time a token takes to travel through is no longer spent waiting in channels, but instead it is mostly spent being executed within the processes. $t_{pF}(p)$ denotes the time of a firing of process p, the token needs to be processed by each process in sequence, so the time the tokens spends executing is given as the sum thereof: $\sum_{p \in \Pi} t_{pF}(p)$.

Note, that we neglected any latency introduced by busy devices. However, since these processes are not bottlenecks, these additional latencies are normally fairly small. Thus we will use this lower bound as an estimation of the real latency for now.

Theorem 5.2.1. *Each path $\psi(s,t)$ can be split into two joining paths $\psi_1(s,r)$ and $\psi_2(r,t)$, such that $\psi_1(s,r)$ has an undrained channel at the end and $\psi_2(r,t)$ consists only of drained channels. The paths ψ_1 and ψ_2 may also have a length of 0.*

Proof. There are 3 possible constellations of drained and undrained channels on $\psi(s,t)$:

- There are only drained channels on ψ:
  In this case we choose $\psi_2 = \psi$ and ψ_1 as empty.
- The last channel on ψ is undrained:
  In this case we choose $\psi_1 = \psi$ and ψ_2 as empty.
- There is at least one channel $c \in \psi$ undrained and the last channel on ψ is drained: In this case we look for the last undrained channel c_l on ψ and choose ψ_1 as the path from the start up to and including c_l and ψ_2 as the rest of ψ.

□

Theorem 5.2.2. *The overall latency over a path $\psi(s,t)$ in a steady state can be calculated as the sum of one path ψ_1 with a undrained channel at the end of it and a path ψ_2 consisting only of drained channels and is thus mathematically defined as*

$$lat_\psi(s,t) = tok(s,r)\frac{1}{q_r \cdot l} + \sum_{p \in \Pi} t_{pF}(p) \tag{5.19}$$

where r is the process, where the two paths join and Π is the set of all processes of the second path, which is .

Proof. This follows directly from theorem 5.2.1 on page 29, since we simply add the latency for path ψ_2 (assumption 5.2.9 on page 28) and the latency for a path of drained channels (assumption 5.2.10 on page 29) up. □

However we actually do not yet know the amount of tokens in between two processes $tok(s,t)$ in the steady state. Thus the next step will be to introduce an algorithm, which finds the amount of tokens on each channel in the steady state.

Definition 5.2.10. *A process $p \in P$ is called* blocked*, if it cannot fire, due to missing tokens on an input channel or not having enough space to write to on a output channel.*

We know, that in the steady state, the whole process network has been throttled due to the limitations of the bottleneck process. We can artificially reconstruct such a state by running the network without ever executing the bottleneck. As such, the bottleneck (and later other blocked processes) will start blocking its direct predecessor processes, as soon as the connecting fifos are full. Similarly blocked processes will block direct successor processes, when the connecting fifo is empty. Thus eventually the whole process network will be blocked. In this state, the whole network has been throttled down to the firing rate of the bottleneck (this rate being zero firings) and is thus representative of the steady state. So we will use this technique in algorithm 5.2.1 on page 31, in order to find the steady state of a process network.

Assumption 5.2.11. *Algorithm 5.2.1 on page 31 terminates and returns the steady state of the associated process network.*

Reasoning. It can easily be seen, that the algorithm terminates, since our loop-free process network, which contains a blocking process b, that is never fired has only a limited amount of overall space to store tokens. Because b

Algorithm 5.2.1 Algorithm to find a steady state of a process network with only one critical process.

Require: All channels are empty
1: P : set of processes
2: $b \in P$: the bottleneck process
3: $U \leftarrow \emptyset$: the set of processes which are certainly not ready
4: **while** $|U| + 1 < |P|$ **do**
5: $p \in P \backslash \{U \cup b\}$ randomly chosen
6: **if** p is ready **then**
7: **while** p is ready **do**
8: fire p
9: **end while**
10: $U \leftarrow p$ *//Remove all other processes from U as they might be ready again*
11: ***else***
12: $U \leftarrow U \cup p$
13: ***end if***
14: ***end while***
15: ***return*** *current fillstates of all channels*

prevents the network from consuming all tokens, we must reach an iteration where no process (except for b) is ready anymore, because all are blocked by full or empty channels. This means that all processes but b are not ready and thus in U, yielding $|U| = |P| - 1$ and terminating the algorithm. Because all processes but b are blocked, the network has been throttled down to the firing rate of b and we thus obtain a steady state.

Now that we have obtained the steady state using algorithm 5.2.1 we now know which channels are drained and can apply theorem 5.2.2 on page 30 to determine the latency. Thus we have now solved our simplified problem with only one bottleneck process and tokenrates of 1.

We will now consider what happens if we allow multiple bottleneck processes. We have to realise, that there is not necessarily one true solution to this problem, since we cannot predict the fillstate of a channel, which connects two bottleneck processes. This is because these processes are supposedly running at the exact same speed, generating and consuming the exact same amount of tokens per second. Thus if in reality one runs just slightly faster than the other, the channel could be completely emptied or completely filled. Thus there are actually multiple steady states possible and we will show algorithms to determine the most extreme cases: The one where all channels in between are empty (algorithm 5.2.2 on page 32) and the one where the channels are full (algorithm 5.2.3 on page 33). Also note the illustrated example in Figure 5.3 on page 37.

Algorithm 5.2.2 Algorithm to find a steady state of a process network with multiple bottleneck process. This state is the one containing the least amount of tokens for any steady state and thus yields the shortest latency.

Require: All channels are empty
1: P : set of processes
2: $B \subseteq P$: the set of all bottleneck process
3: $U \leftarrow \emptyset$: the set of processes which are certainly not ready
4: **while** $|U| + |B| < |P|$ **do**
5: $p \in P \backslash \{U \cup B\}$ randomly chosen
6: **if** p is ready **then**
7: **while** p is ready **do**
8: fire p
9: **end while**
10: $U \leftarrow p$ *//Remove other processes from U as they might be ready again*
11: ***else***
12: $U \leftarrow U \cup p$
13: ***end if***
14: ***end while***
15: ***return*** *current fillstates of all channels*

Theorem 5.2.3. *Algorithm 5.2.2 terminates and finds the steady state with the least amount of tokens of the associated process network.*

Proof. If we block execution of all bottleneck processes, we simulate the case, where the first bottlenecks throttle the ones in the back, as this simulation leaves the channels in between bottlenecks empty. ☐

Theorem 5.2.4. *Algorithm 5.2.3 on page 33 terminates and finds the steady state with the most amount of tokens of the associated process network.*

Proof. This time the algorithm assumes the bottlenecks later in the process network to be the limiting factors, first filling the channels in between them and earlier bottlenecks, which will then throttle the bottlenecks. ☐

We will now consider the changes of the algorithm if we allow tokenrates to be greater than 1. Assumption 5.2.9 on page 28 is no longer valid, because a process may consume multiple tokens per firing. However we can still relate the number of firings of a process to the amount of tokens on the separate channels on paths leading to said process:

Algorithm 5.2.3 Algorithm to find a steady state of a process network with multiple bottleneck process. This state is the one containing the most amount of tokens for any steady state and thus yields the longest possible latency.

Require: All channels are empty
1: P : set of processes
2: $B \subseteq P$: the set of all bottleneck process
3: $T \subseteq B$: the set of bottleneck processes, which have no path starting from them, leading to another bottleneck process
4: $U \leftarrow \emptyset$: the set of processes which are certainly not ready
5: **while** $|U| + |T| < |P|$ **do**
6: $\quad p \in P \backslash \{U \cup T\}$ randomly chosen
7: $\quad$ **if** p is ready **then**
8: $\quad\quad$ **while** p is ready **do**
9: $\quad\quad\quad$ fire p
10: $\quad\quad$ **end while**
11: $\quad\quad$ $U \leftarrow p$ *//Remove other processes from U as they might be ready again*
12: $\quad$ ***else***
13: $\quad\quad$ $U \leftarrow U \cup p$
14: $\quad$ ***end if***
15: ***end while***
16: ***return*** *current fillstates of all channels*

Theorem 5.2.5. *The latency $lat_\psi(s,t)$ in between two processes s,t connected over a path ψ can be calculated as*

$$lat_\psi(s,t) = \sum_{c\in\psi} \frac{b_c}{o_c} \cdot \frac{1}{q_u \cdot l}$$

where u represents the process, which has c as input channel. b_c denotes the number of tokens on channel c in the steady state. This holds if the last channel c on path ψ is not drained.

Proof. Similar to assumption 5.2.9 on page 28 we try again to count the number of firings, that t has to execute, to consume all tokens. We do this for each channel $c \in \psi$ separately. We know the number of firings of u which are necessary as $\frac{b_c}{o_c}$, consuming o_c tokens per firing. Since the number of firings of different processes are correlated over the firing vector, we know the number of firings of t as $\frac{b_c}{o_c} \cdot \frac{q_t}{q_u}$. Now we just have to multiply the number of firings by the period of a firing of t, which is $\frac{1}{q_t \cdot l}$, yielding $\frac{b_c}{o_c} \cdot \frac{q_t}{q_u} \cdot \frac{1}{q_t \cdot l} = \frac{b_c}{o_c} \cdot \frac{1}{q_u \cdot l}$. Now it only remains to sum over each channel $c \in \psi$ □

However assumption 5.2.10 on page 29 is still a fair assumption, as a token travelling through a path of drained channels should still not have to wait on the channels long. It may indeed happen, that due to mismatching tokenrates of reading and writing process, the token has to wait some time on a channel, however we assume this time to be small.

For theorem 5.2.2 on page 30 to still hold we have to replace Equation (5.19) when tokenrates are bigger than 1, since it is based on assumption 5.2.9 on page 28, which is no longer valid. The new equation is an addition over both paths exactly the same as described in the original theorem and given as:

$$lat_\psi(s,t) = \sum_{c\in\psi_1} \frac{b_c}{o_c} \cdot \frac{1}{q_u \cdot l} + \sum_{p\in\Pi} t_{pF}(p) \tag{5.20}$$

where u is the process which reads from c, r is the process where the two paths join and Π is the set of all processes of the second path ψ_2. b_c denotes the number of tokens on channel c in the steady state.

Also we have to consider, whether we can still use the algorithms 5.2.2 and 5.2.3 on page 32 and on page 33 to obtain the steady state, since we have to find the number of tokens on each channel b_c to utilise determine the latency.

Assumption 5.2.12. *Algorithms 5.2.2 and 5.2.3 on page 32 and on page 33 return valid steady states, which determine good bounds for the set of all possible steady states, even if tokenrates are greater than* 1.

Reasoning. Theorem 5.2.3 on page 32 and theorem 5.2.4 on page 32 do not require the tokenrate to be 1 and thus the algorithms still return a valid reachable steady state. However, now there can be a multitude of additional possible steady states, because if the tokenrates from and to a channel mismatch, there might be tokens left behind or token capacity, which is unusable. But it can be seen, that the amount of tokens which we miscount are upper bounded by the input- and output tokenrates. This is because a process will fire as soon as its input channels have enough tokens for one firing. All the same, a process will fire as soon as there is enough space on its output channels. While our error is upper bounded by the tokenrates, the total space in a fifo must be a multiple of its input and output rates (assumption 5.2.7 on page 23). Thus it is fair to say, that the error we introduce by not checking the other steady states is negligible.

Note, that it would be possible to obtain the additional states. Instead of finishing, when a steady state is found, you would simply execute the bottleneck once and then rerun the algorithm to obtain another steady state. However this can become highly complicated with multiple bottlenecks, since one would have to try every constellation of different bottlenecks firing a certain nubmer of times.

Solution Given a dataflow graph $g = \langle P, C \rangle$, an architecture $a =< D >$ and a mapping $m : P \to D$, we find the latency of the mapped process network as follows:

1. Find the worst and best steady states using algorithms 5.2.2 and 5.2.3 on page 32 and on page 33.

2. For each of the obtained steady states:

   (a) For each directed path ψ in the network:

      i. Find the last channel c_r on ψ, which is not drained and has process r as consumer.

      ii. split ψ in a path ψ_1, which has the undrained channel c_r at the end and a path ψ_2, consisting of only drained channels. These two paths join at process r

      iii. determine the latency of the path as $lat_\psi(s,t) = \sum_{c \in \psi_1} \frac{b_c}{o_c} \cdot \frac{1}{q_u \cdot l} + \sum_{p \in \Pi} t_{pF}(p)$, where u is the process which reads from c.

   (b) Maximise over all discovered latencies to obtain the estimate of the process network latency.

Note that we can optimise this procedure in multiple ways:

- We do not have to check all paths within the network, since a path ψ_a, which is sub-path to another path ψ_b cannot yield the worst case latency, as latency may only increase for longer paths.
- We can thus easily compute the subset of possible end processes of such a path, by searching for all processes which have no output channels.

5.3 Summary

In this chapter we have shown how to model a process network execution in the DAL framework. We did this by first only considering the execution time of one volley on a device $t_{execution}$ and developing a separate model for it. We were then able to build a model of the overall time needed for one volley t_{pV} based on the model for the execution time $t_{execution}$. Properly defining the scope of our network model enabled us to model throughput and latency of a process network.

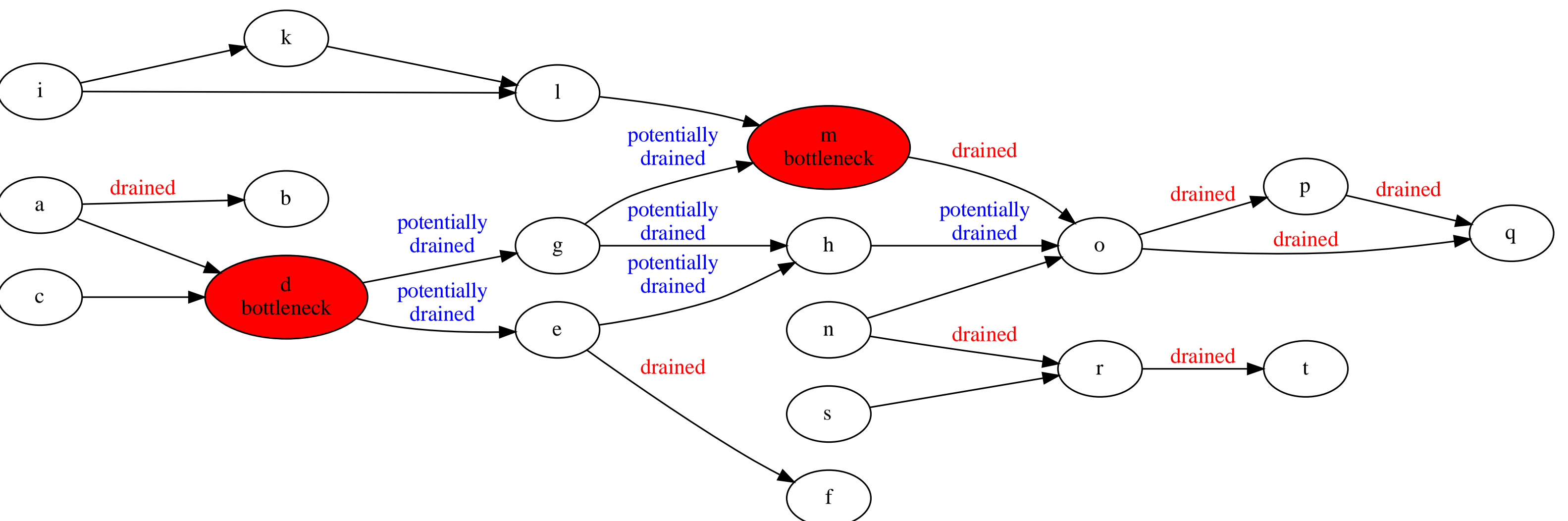


Figure 5.3: Illustration of the buffer states of a steady state of a fairly complex process network with two bottlenecks. Note that the channels labelled "potentially drained" will be drained, if we assume the bottleneck d to not quite generate enough tokens for m to consume (This state is obtained by running algorithm 5.2.2 on page 32). The channels will be not drained, if we assume m to consume slightly more than d may generate (This state is obtained by running algorithm 5.2.3 on page 33).

6
Experimental Evaluation

In this chapter we will run some experiments to test against our developed process execution model from chapter 5 on page 17. This means we will create some sample application process networks and evaluate them on different heterogeneous systems. In parallel we will calibrate our models of the processes for each heterogeneous system and compare the calibrated model to the measurements obtained by our profiling tools (chapter 4 on page 13).

6.1 Setup

In this section we will shortly describe the surrounding conditions of our testing environment. At first we will have a look at the used hardware systems and then shortly explain our sample applications, that we used for testing.

6.1.1 Hardware

The considered hardware platform consists of two CPU's, and a GPU. There are two Intel CPU's "Xeon(R) CPU E5" with a total of 8 cores using the Intel driver version 1.2.0.82248, which is OpenCL 1.2 compatible. The GPU is an AMD graphics card "Radeon HD 7900 Series", with 32 work groups, each containing 64 work items using the AMD driver version 1445.5 (VM), which is OpenCL 1.2 compatible.

6.1.2 Application

We will now shortly describe the functionality of our used testing example.

Single Modifier Example

This example is a simple process network, consisting of one OpenCL process, the modifier, as well as two POSIX threads, the generator and the consumer. The generator has an internal counter and each firing will send the counter value as an integer token and increment the counter. Because this would generate a lot of firings with little tokens, the generator can be configured to send a vector of multiple integer tokens per firing.

The modifier will distribute the integer tokens evenly among its assigned work items and each integer will be multiplied multiple times with its original value. The number of multiplications can be chosen freely, but should be chosen as a prime number, since according to Eulers theorem [7] it holds that:

$$a^{\phi(n)} \equiv 1 (\text{mod } n) \tag{6.1}$$

where $\phi(n)$ is the Eulers totient function. If n is chosen prime, then $\phi(n) = n - 1$ which can be reformulated as $a^n \equiv a (\text{mod } n)$. This means that the modifier will execute a lot of division and multiplication functions, but in the end its result, i.e. $a^n \mod n$, will be the same as its input a. So the modifier will be busy and always output its input values. The modifier can fully profit from parallelization, since each output token only depends on a single input token and is independent of all others.

The consumer will read the integer tokens produced by the modifier and has an internal counter, which predicts the value it is supposed to receive. It will always check whether the received values are correct when consuming them.

The generator and consumer are synthesized as POSIX threads, because their internal counter requires these processes to be stateful. The modifier is synthesized with OpenCL and can be executed on graphics cards. In our experiments we will thus always analyze the modifiers performance.

6.2 Process Execution Model

In the following we will test whether our process execution model reflects the reality.

6.2.1 Average Execution Time per Firing

In the following we will evaluate the average execution time per firing, that a separate process needs. Note that we observe the execution time averaged over many firings. Since multiple firings are executed in parallel as a volley, this average execution time depends on the number of work groups used. In Section 5.1.1 on page 19 we have modelled the execution time per volley, so dividing Equation (5.6) on page 20 by the number of work items yields the estimation Equation (6.2).

$$t_{epF} = \frac{\left\lceil \frac{\#WI}{\#PE} \right\rceil \cdot \left\lceil \frac{\#WG}{\#CU} \right\rceil \cdot \left(\left\lceil \frac{\#I_{pWG}}{\#WI} \right\rceil \cdot t_{CpI} + t_{mem} \right) + t_{overhead}}{\#WG} \tag{6.2}$$

In the following we show an experiment using the single modifier example (see Section 6.1.2 on page 40) on the server architecture on the GPU. We are focusing on the impact of the number of work items and thus leave the number of work groups constant for each experiment.

In Figure 6.1 on page 42 we see an example, where the number of tokens per firings has been chosen large, as 512. This corresponds to $\#I_{pWG}$ in Equation (6.2), which means that the term $\left\lceil \frac{\#I_{pWG}}{\#WI} \right\rceil$ looks like a hyperbola function for small $\#WI$, while the term $\left\lceil \frac{\#WI}{\#PE} \right\rceil$ is just constantly equal 1. This results in the observed overall hyperbola function. We see, that the measurements of the real system fit very closely to our model.

In Figure 6.2 on page 42 we see an example with a much smaller number of iterations per firing of 32, but instead we have artificially increased the time it takes per iteration t_{CpI}. Now the term $\left\lceil \frac{\#I_{pWG}}{\#WI} \right\rceil$ is not a smooth function anymore, but jumps, whenever $\#WI$ is a divisor of 32. Again we see, that the measurements match our theoretical model very closely.

These experiments show, that increasing the number of work items can improve throughput heavily, especially, when the number of work items is smaller than the number of available processing elements per compute unit. Also latency profits from the additional work items, as the overall execution takes less time. However, we also see that processes, which do not feature enough internal parallelism cannot profit from this and are thus less suited for devices with parallel architectures.

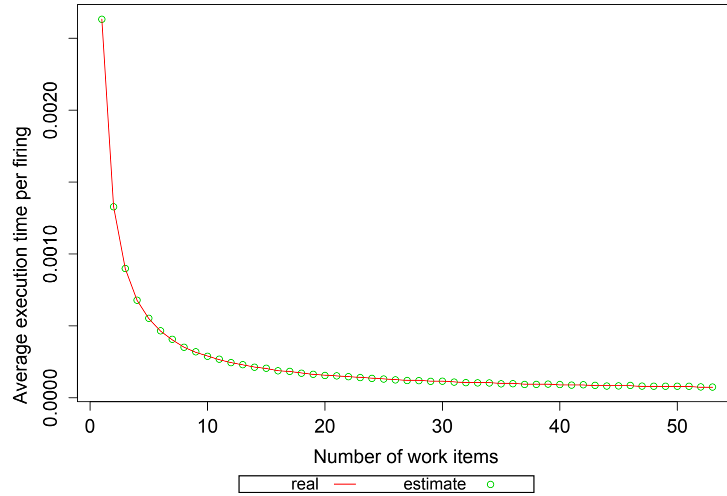


Figure 6.1: Test-run of a single process on the AMD GPU with 32 compute units and 64 processing elements per compute unit, working on 1 work group, spanning 512 iterations.

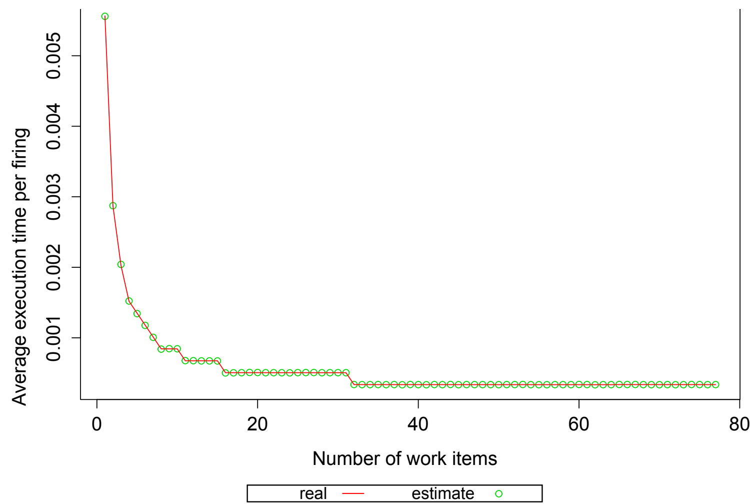


Figure 6.2: Test-run of a single process on the AMD GPU with 32 compute units and 64 processing elements per compute unit, working on 16 work groups, each spanning 32 iterations.

6.2.2 Average Time per Firing

We will now look at the average time used for a firing. This includes the time for the memory transfer and can be inferred from the model of the time per volley in Equation (5.8) on page 21 by again dividing by the number of work groups, see Equation (6.3). Note, that the number of work items actually has no influence on the additional transfer and overhead times at all, which is why we focus our observations on the number of work groups.

$$\begin{aligned} t_{pF} = \Bigg(& \left\lceil \frac{\#WI}{\#PE} \right\rceil \cdot \left\lceil \frac{\#WG}{\#CU} \right\rceil \cdot \left\lceil \frac{\#I_{pWG}}{\#WI} \right\rceil \cdot t_{CpI} \\ & + \left\lceil \frac{\#WI}{\#PE} \right\rceil \cdot \left\lceil \frac{\#WG}{\#CU} \right\rceil \cdot t_{mem} + t_{overhead} \\ & + t_{TpWG} \cdot \#WG + t_{overhead2} \Bigg) \cdot \frac{1}{\#WG} \end{aligned} \quad (6.3)$$

The first experiment will execute the single modifier example (see Section 6.1.2 on page 40) on the Intel CPU, varying the amount of work groups. With a fixed amount of 32 work items per work group, we can clearly see the effect of the term $\left\lceil \frac{\#WG}{\#CU} \right\rceil$ causing spikes, whenever $\#WG$ reaches a multiple of $\#CU = 8$. We can see, that our estimate is fairly close to the actual values.

In a second experiment, executing the same single modifier example on the AMD GPU, we can see that the term $\left\lceil \frac{\#WG}{\#CU} \right\rceil$ has no large impact anymore, since the number of compute units is now 32. Thus we can see the effect, that with increasing number of work groups, the time per firing decreases diminishingly. Our model seems to match the measurements quite accurately once more.

In these experiments we see, that an increase in the number of work groups can improve throughput significantly, as the transfer time is a constant time used for all work groups together. We also note that choosing the number of work groups as a multiple of the number of compute units is beneficial, because otherwise some work groups have to wait for others to finish, even though there are additional compute units available. However the drawback is, that the latency increases with the number of work groups, because the execution of an entire volley takes much longer.

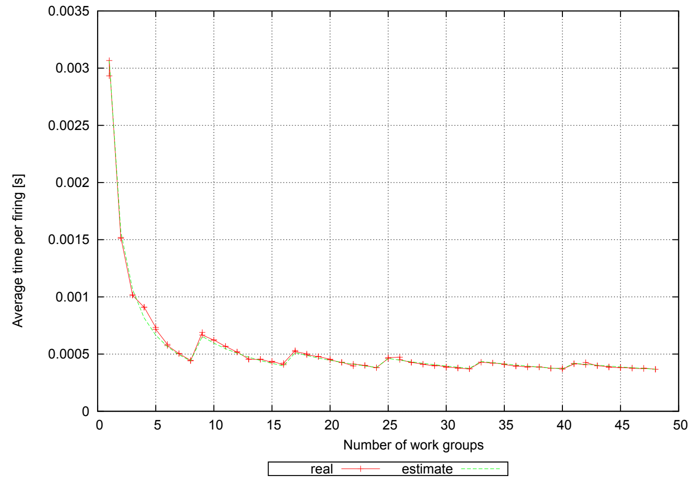


Figure 6.3: Test-run of a single process on the Intel CPU with 8 compute units and 32 processing elements per compute unit, working on 32 Work Items per Work Group.

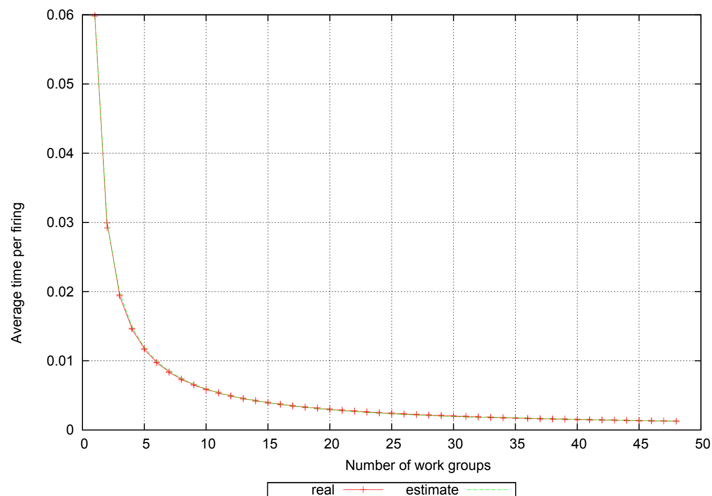


Figure 6.4: Test-run of a single process on the AMD GPU with 32 compute units and 64 processing elements per compute unit, working on 64 Work Items per Work Group.

6.3 Summary

In this section we have seen the setup used for experimental evaluation, as well as the comparison of our experiments and the calibrated models. In our experiments, we note that the calibrated models are a very close match to the measured values from our experiments.

It can be seen, that increasing the number of used work groups or work items can be very beneficial up to some point, but that the throughput gained will decrease logarithmically. In the case of the number of work groups, we noted a trade off between latency and throughput, as both increase with larger amounts of work groups.

7
Conclusion and Outlook

7.1 Conclusion

In this thesis we proposed an approach, which uses a calibration algorithm for extracting performance characteristics for a given application and architecture, based on which we could perform an accurate estimation for any given mapping. We have considered both the binding and the parallelisation parameters as part of our mapping.

We have developed profiling tools, which allow analysis of a given Synchronous Data Flow graph and its separate processes.

The calibration algorithm, which extracts the data needed for estimation, has been implemented. We have also performed experimental evaluations , which show that its calibrated performance models match the test results very well. Thus we know that the created models for the execution time and the time per volley are very accurate.

We have developed an algorithm, which can estimate throughput and latency of any mapping. We have proposed methods for determining the throughput and latency of a process network, only knowing the execution time and the time per volley of each process.

7.2 Outlook

As this thesis provides the tools to find an optimal mapping it would now be easily possible to use a design space exploration algorithm in order to find an optimal mapping for a given architecture and application.

Additional research into the exact mechanics of memory management could yield some further improvements to our process models, which might enable further improvements in performance.

Testing the algorithms with real applications and experimentally evaluating the estimation algorithm would also further improve the usefulness of this thesis.

A
Appendix

A.1 Presentation Slides

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Mapping Optimisation of Streaming Applications on Heterogeneous Platforms

Master Thesis

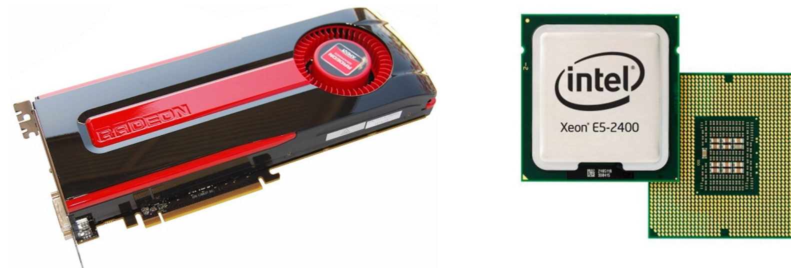
Felix Wermelinger

Advisers: Andreas Tretter & Lars Schor

3rd September 2014

Motivation

- Modern architectures feature multiple devices
- Modern devices feature inherent parallelism
- How can we use these devices efficiently?



Goals

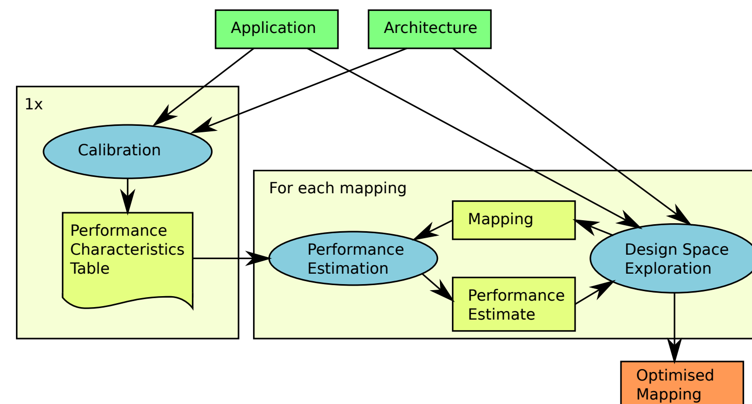
- Analyse performance of different mappings of streaming applications on heterogeneous platforms
  - What parameters of mapping affect performance?
  - Can we model the dependency?

Perspective

- Find a mapping which optimizes performance
  - Find it within reasonable time

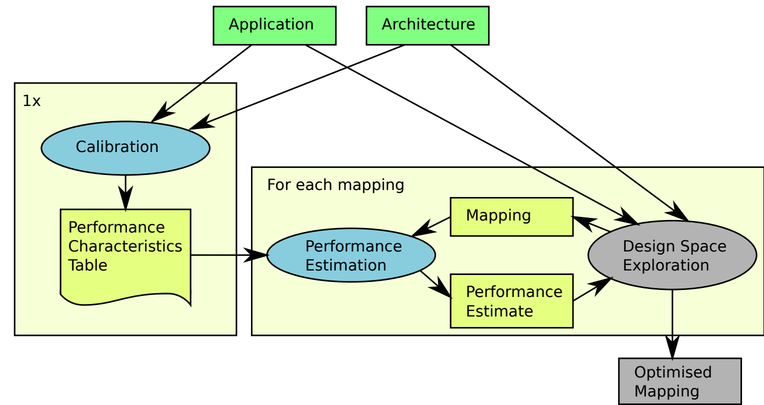
Approach

- Calibration creates data table, characterizing performance
- Estimator calculates performance for a given mapping
- Design space explorer explores mappings to find optimised mapping



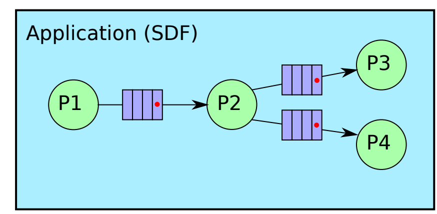
Contributions

- Developed calibration
- Developed estimation
- Experimental evaluation of implementations



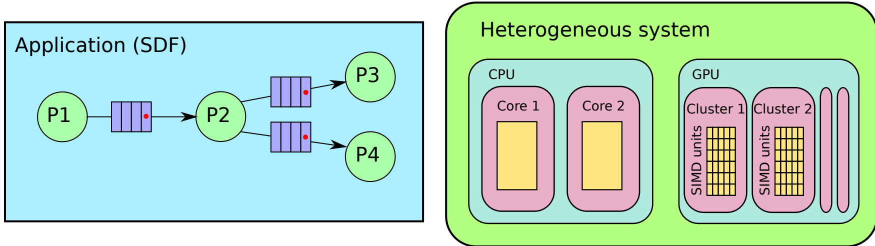
Problem Description

- Streaming application given as Synchronous Dataflow Graph (SDF)
- Architecture is a set of devices with multiple levels of parallelism
- The returned mapping has to include binding of processes to devices and setting of parallelism used



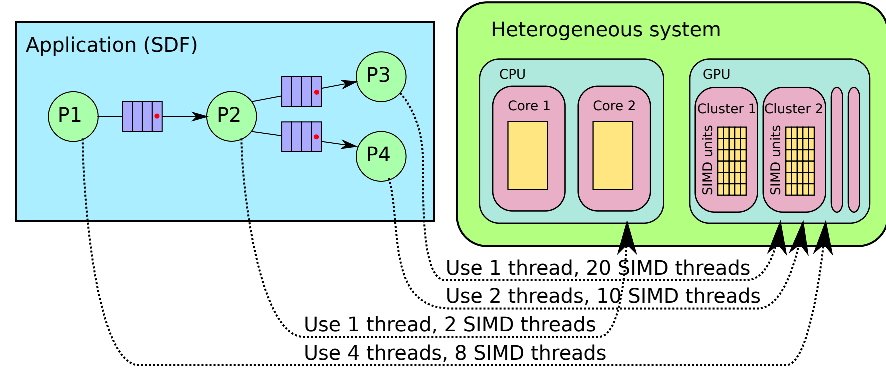
Problem Description

- Streaming application given as Synchronous Dataflow Graph (SDF)
- Architecture is a set of devices with multiple levels of parallelism
- The returned mapping has to include binding of processes to devices and setting of parallelism used



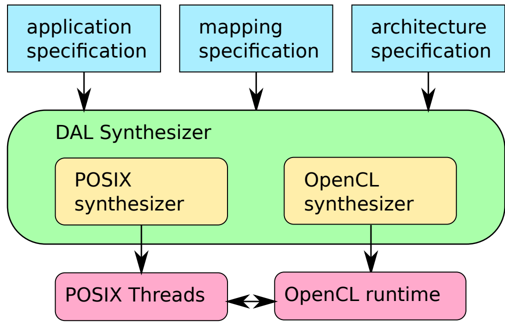
Problem Description

- Streaming application given as Synchronous Dataflow Graph (SDF)
- Architecture is a set of devices with multiple levels of parallelism
- The returned mapping has to include binding of processes to devices and setting of parallelism used



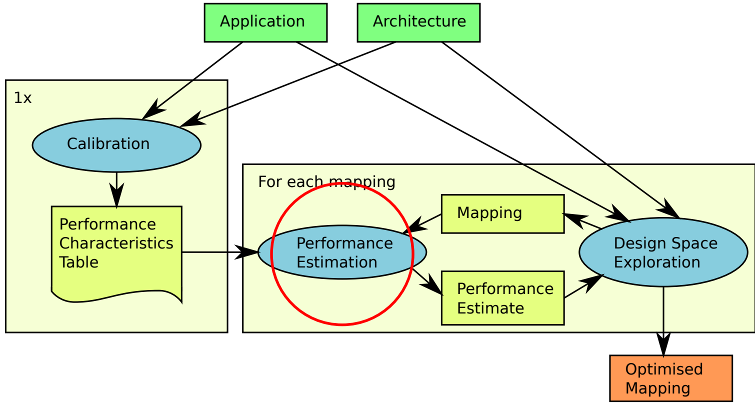
Context

- The Distributed Application Layer framework was used for synthesis and execution
  - Uses OpenCL or POSIX threads for processes
  - Manages scheduling and memory transfer

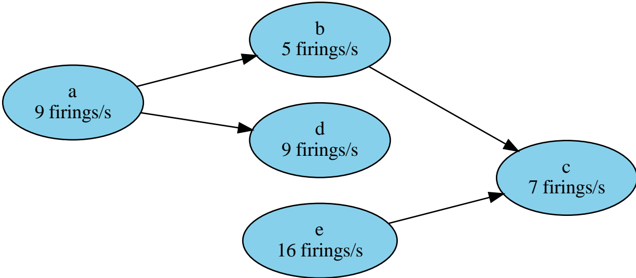


Performance Estimation

- Optimized performance: maximal throughput with minimal latency
- Throughput: the number of tokens processed per second
- Latency: longest time a token may need to traverse the network

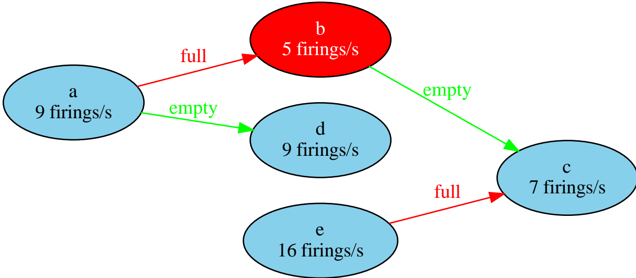


Performance Estimation - Throughput



- The bottleneck limits overall throughput
- Bottleneck throttles all other processes $\longrightarrow$ Steady state

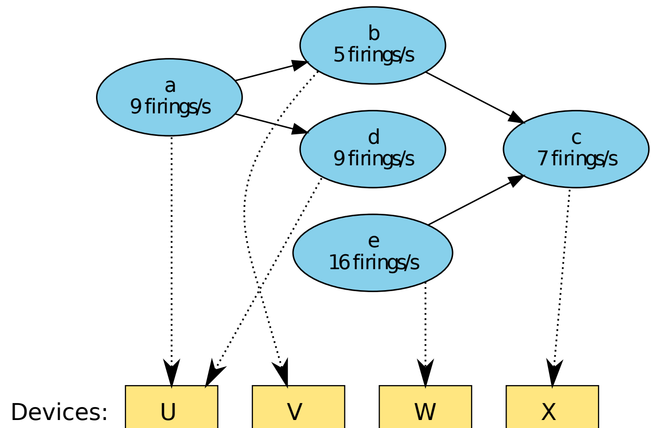
Performance Estimation - Throughput



- The bottleneck limits overall throughput
- Bottleneck throttles all other processes $\longrightarrow$ Steady state

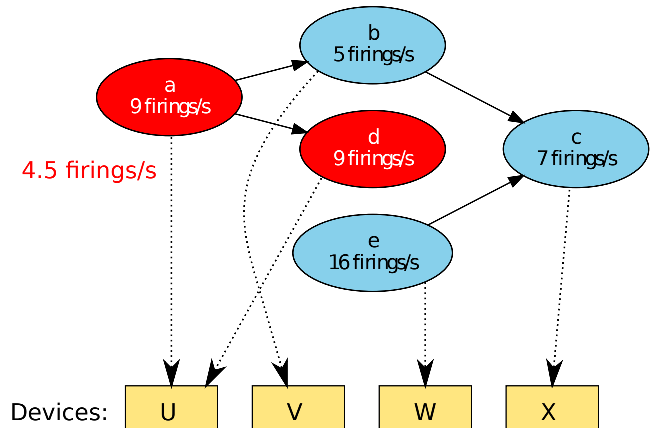
Performance Estimation - Throughput

- Processes are bound to devices



Performance Estimation - Throughput

- Processes are bound to devices

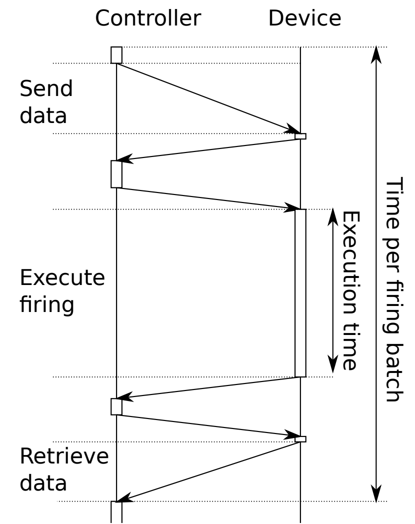


Performance Estimation - Throughput

- Time per firing batch ≠ Execution time
- Device can execute multiple threads in parallel

For throughput estimation we need to know

- Execution time
- Time per firing batch

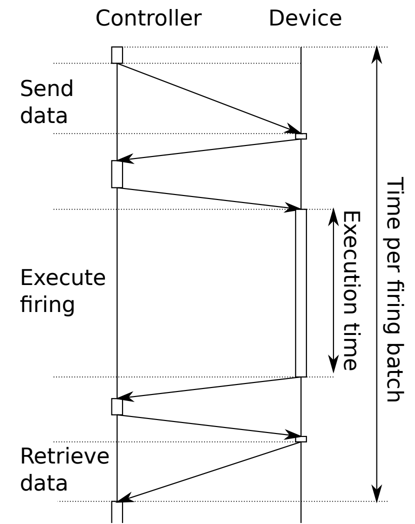


Performance Estimation - Throughput

- Time per firing batch ≠ Execution time
- Device can execute multiple threads in parallel

For throughput estimation we need to know

- Execution time
- Time per firing batch

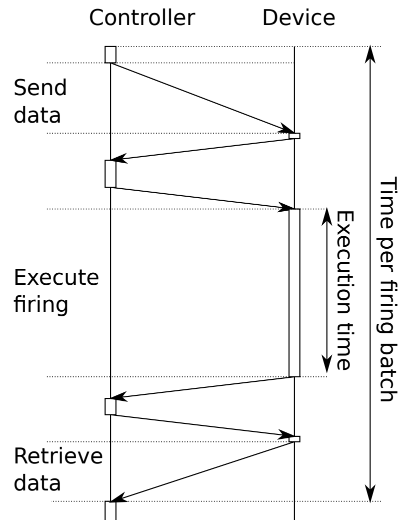


Performance Estimation - Throughput

- Time per firing batch $\neq$ Execution time
- Device can execute multiple threads in parallel

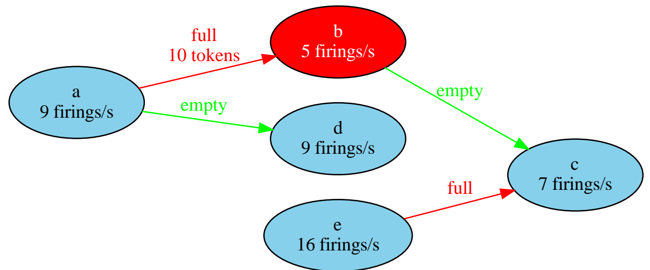
For throughput estimation we need to know

- Execution time
- Time per firing batch



Performance Estimation - Latency

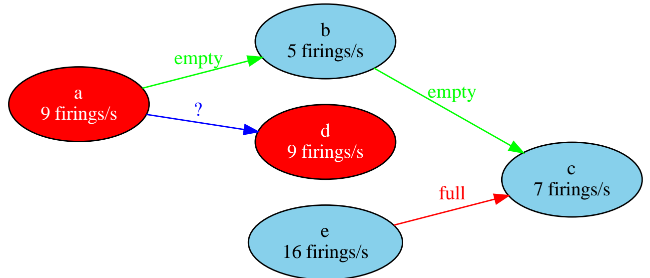
- Number of tokens on channels in steady state is known



- Latency to traverse channel: determined by number of tokens in steady state on channel
- Latency to traverse process = Time per firing batch

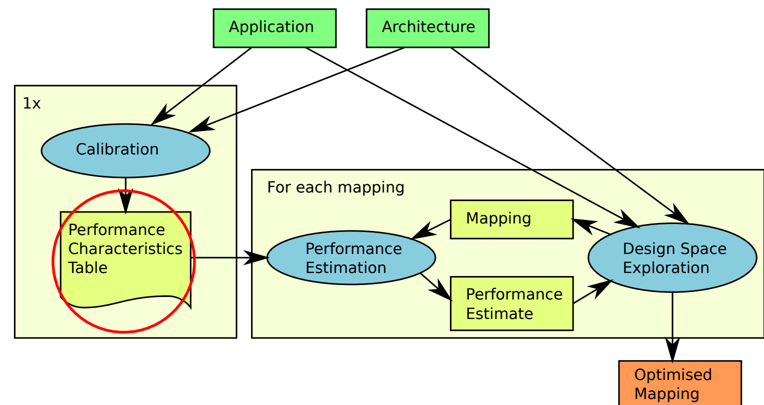
Performance Estimation - Latency

- Multiple bottlenecks $\longrightarrow$ unknown channel fillstates



- Steady state is unknown
  - We can still give boundaries for latency

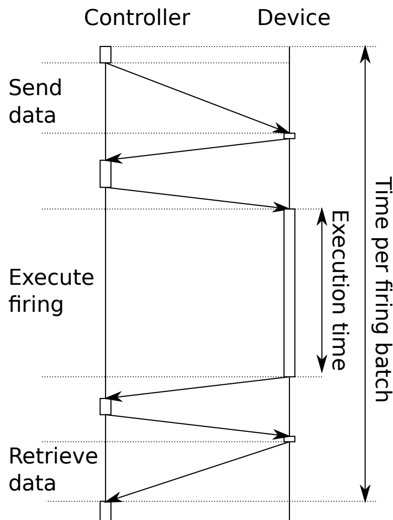
Performance Characteristics Table



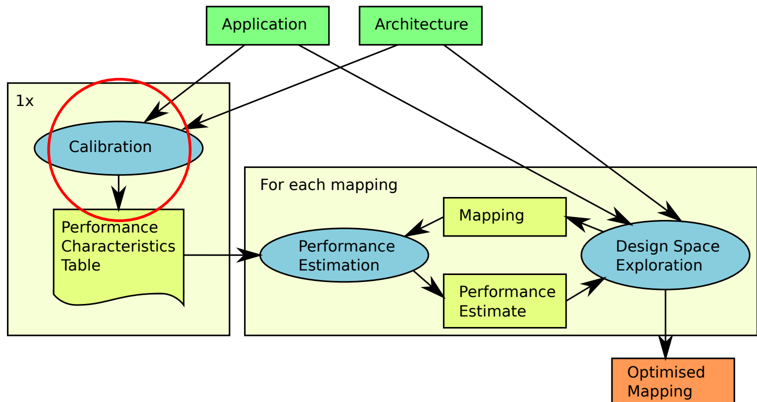
Performance Characteristics Table

For throughput estimation we need to know

- Execution time
- Time per firing batch



Calibration

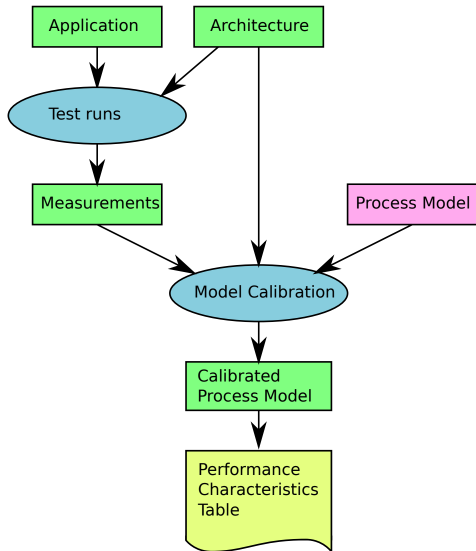


Goal:

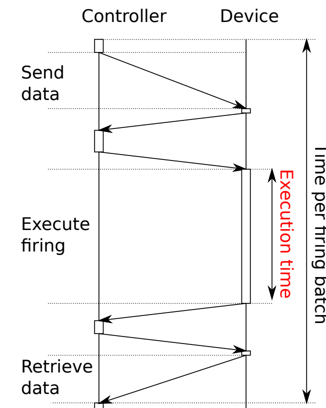
- Create table with timing values for each mapping (binding & parallelisation) of each process
  - This design space is still very large

Calibration - Concept

- Test each process binding separately
- Measure needed timing values
  - Time per firing batch
  - Execution time
- Instead of exploring whole design space:
  - Only execute few test cases
  - Interpolate values according to process model



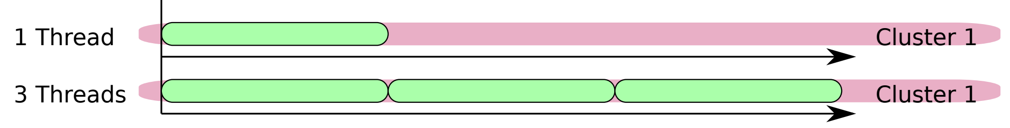
Process Performance Model - Execution Time



$$\text{Execution time} = \left\lceil \frac{\#\text{Threads}}{\#\text{Clusters available}} \right\rceil \cdot \left\lceil \frac{\#\text{SIMD threads}}{\#\text{SIMD units}} \right\rceil \cdot \left(\left\lceil \frac{\#\text{Operations}}{\#\text{SIMD threads}} \right\rceil + t_{mem} \right) \cdot \text{time per Operation} + t_{overhead}$$

Process Performance Model - Execution Time

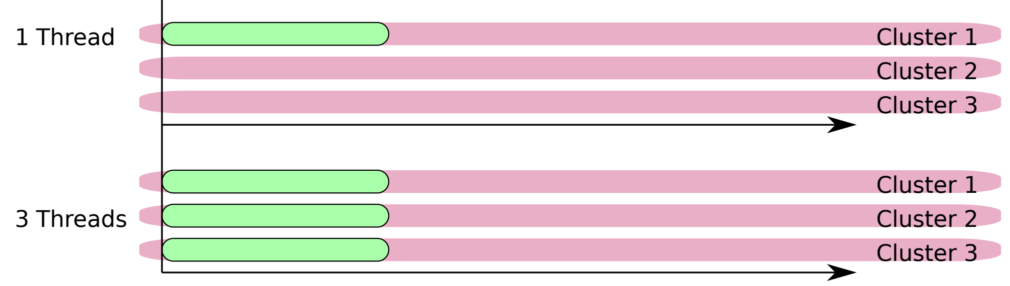
- We can configure multiple threads to be executed together



$$\text{Execution time} = \left\lceil \frac{\#\text{Threads}}{\#\text{Clusters available}} \right\rceil \cdot \left\lceil \frac{\#\text{SIMD threads}}{\#\text{SIMD units}} \right\rceil \cdot \left(\left\lceil \frac{\#\text{Operations}}{\#\text{SIMD threads}} \right\rceil + t_{mem} \right) \cdot \text{time per Operation} + t_{overhead}$$

Process Performance Model - Execution Time

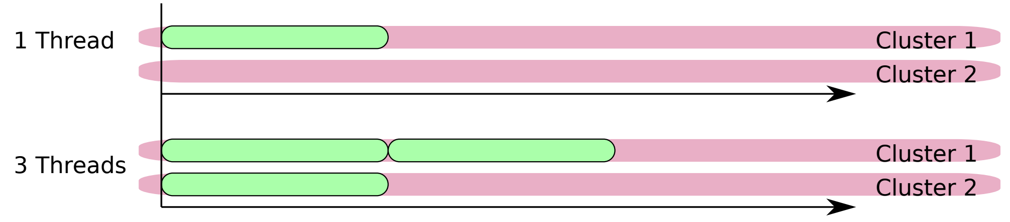
- We can configure multiple threads to be executed together



$$\text{Execution time} = \left\lceil \frac{\#\text{Threads}}{\#\text{Clusters available}} \right\rceil \cdot \left\lceil \frac{\#\text{SIMD threads}}{\#\text{SIMD units}} \right\rceil \cdot \left(\left\lceil \frac{\#\text{Operations}}{\#\text{SIMD threads}} \right\rceil + t_{mem} \right) \cdot \text{time per Operation} + t_{overhead}$$

Process Performance Model - Execution Time

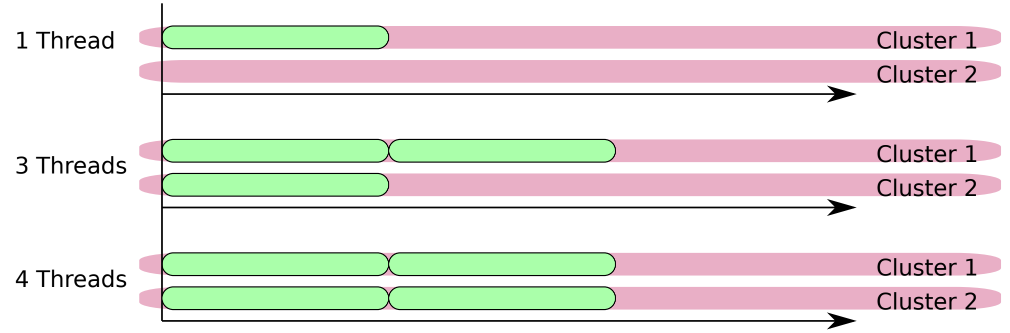
- We can configure multiple threads to be executed together



$$\text{Execution time} = \left\lceil \frac{\#\text{Threads}}{\#\text{Clusters available}} \right\rceil \cdot \left\lceil \frac{\#\text{SIMD threads}}{\#\text{SIMD units}} \right\rceil \cdot \left(\left\lceil \frac{\#\text{Operations}}{\#\text{SIMD threads}} \right\rceil + t_{mem} \right) \cdot \text{time per Operation} + t_{overhead}$$

Process Performance Model - Execution Time

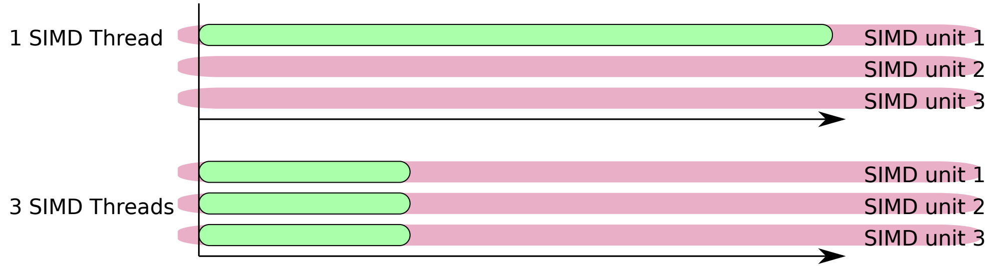
- We can configure multiple threads to be executed together



$$\text{Execution time} = \left\lceil \frac{\#\text{Threads}}{\#\text{Clusters available}} \right\rceil \cdot \left\lceil \frac{\#\text{SIMD threads}}{\#\text{SIMD units}} \right\rceil \cdot \left(\left\lceil \frac{\#\text{Operations}}{\#\text{SIMD threads}} \right\rceil + t_{mem} \right) \cdot \text{time per Operation} + t_{overhead}$$

Process Performance Model - Execution Time

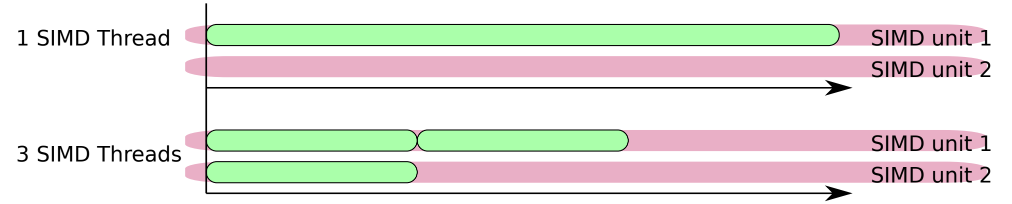
- Work of a cluster gets split onto SIMD units



$$\text{Execution time} = \left\lceil \frac{\#\text{Threads}}{\#\text{Clusters available}} \right\rceil \cdot \left\lceil \frac{\#\text{SIMD threads}}{\#\text{SIMD units}} \right\rceil \cdot \left(\left\lceil \frac{\#\text{Operations}}{\#\text{SIMD threads}} \right\rceil + t_{mem} \right) \cdot \text{time per Operation} + t_{overhead}$$

Process Performance Model - Execution Time

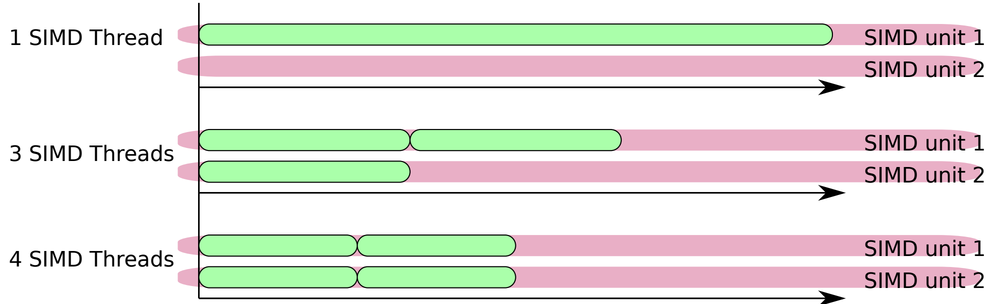
- Work of a cluster gets split onto SIMD units



$$\text{Execution time} = \left\lceil \frac{\#\text{Threads}}{\#\text{Clusters available}} \right\rceil \cdot \left\lceil \frac{\#\text{SIMD threads}}{\#\text{SIMD units}} \right\rceil \cdot \left(\left\lceil \frac{\#\text{Operations}}{\#\text{SIMD threads}} \right\rceil + t_{mem} \right) \cdot \text{time per Operation} + t_{overhead}$$

Process Performance Model - Execution Time

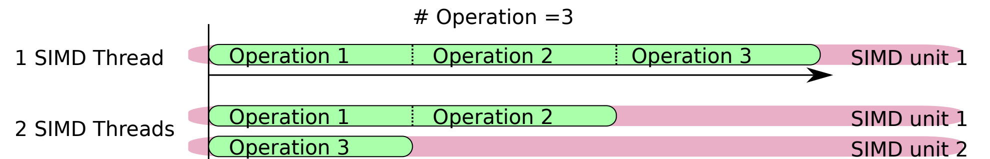
- Work of a cluster gets split onto SIMD units



$$\text{Execution time} = \left\lceil \frac{\#\text{Threads}}{\#\text{Clusters available}} \right\rceil \cdot \left\lceil \frac{\#\text{SIMD threads}}{\#\text{SIMD units}} \right\rceil \cdot \left(\left\lceil \frac{\#\text{Operations}}{\#\text{SIMD threads}} \right\rceil + t_{mem} \right) \cdot \text{time per Operation} + t_{overhead}$$

Process Performance Model - Execution Time

- The thread cannot be segmented arbitrarily



$$\text{Execution time} = \left\lceil \frac{\#\text{Threads}}{\#\text{Clusters available}} \right\rceil \cdot \left\lceil \frac{\#\text{SIMD threads}}{\#\text{SIMD units}} \right\rceil \cdot \left(\left\lceil \frac{\#\text{Operations}}{\#\text{SIMD threads}} \right\rceil + t_{mem} \right) \cdot \text{time per Operation} + t_{overhead}$$

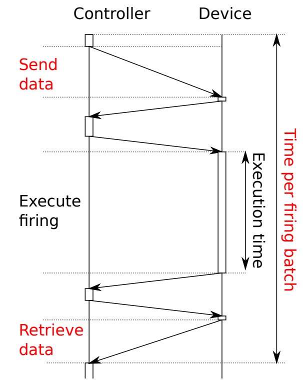
Process Performance Model - Execution Time

$$\text{Execution time} = \left\lceil \frac{\#\text{Threads}}{\#\text{Clusters available}} \right\rceil \cdot \left\lceil \frac{\#\text{SIMD threads}}{\#\text{SIMD units}} \right\rceil \cdot \left(\left\lceil \frac{\#\text{Operations}}{\#\text{SIMD threads}} \right\rceil + t_{mem} \right) \cdot \text{time per Operation} + t_{overhead}$$

- All values but our parallelisation variables are constant for a given binding.

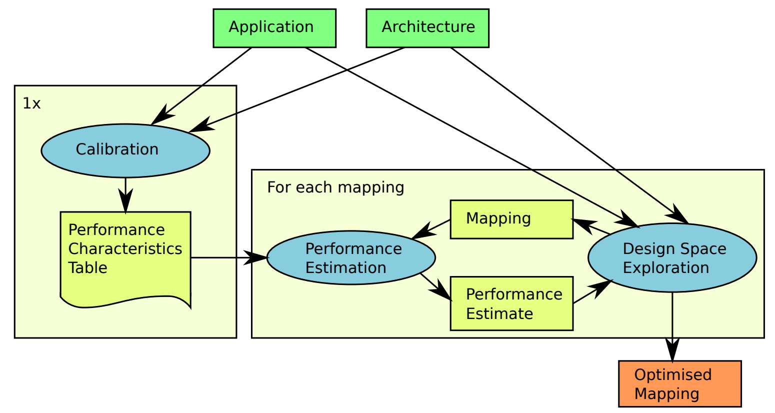
Process Performance Model - Time per Firing Batch

- Additional time for communication & data transfer
  - Most timeintervals only depend on device binding
  - "Send" & "Retrieve" scale with data transfered



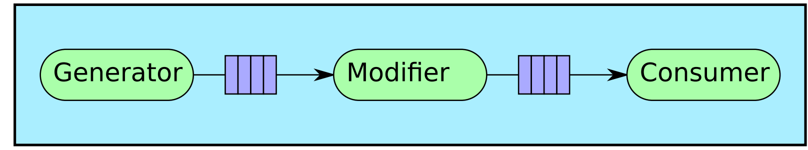
$$\text{Time per firing batch} = \text{Execution time} + t_{overhead} + t_{transfer} \cdot \#\text{Threads}$$

Overview

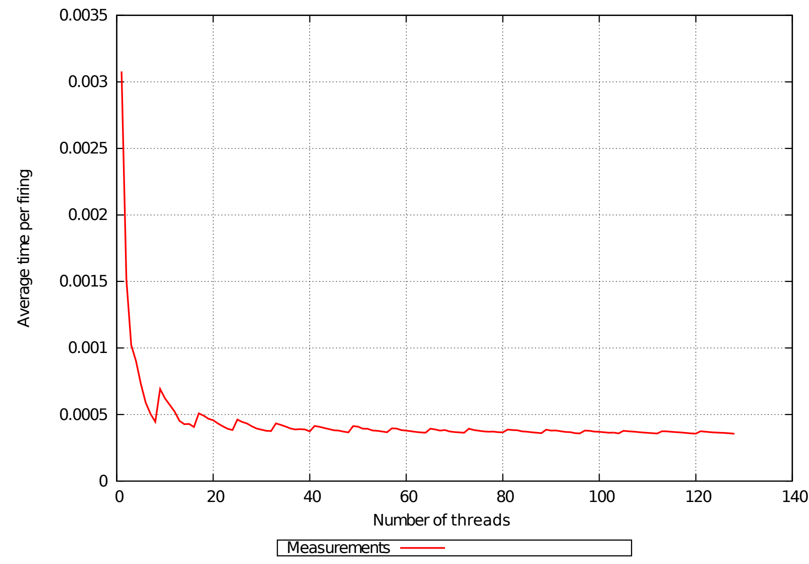


Experimental Evaluation - Setup

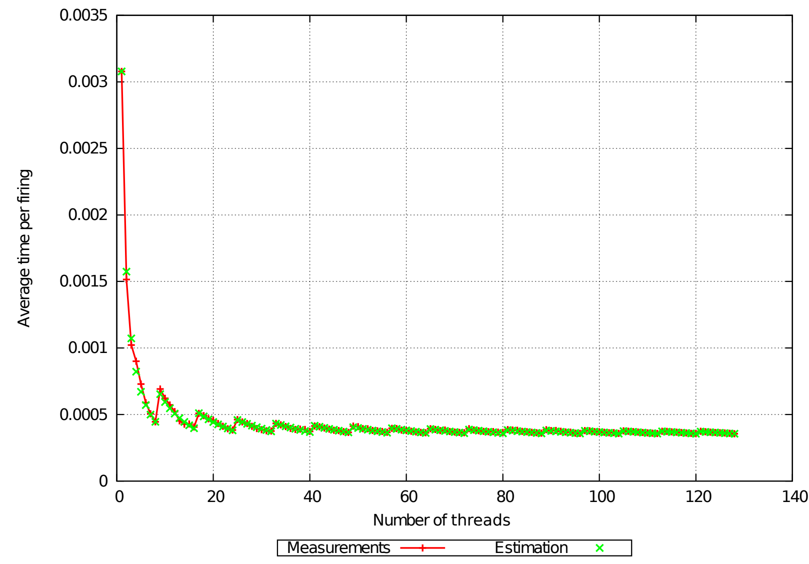
- Hardware:
  - 2 CPU's "Xeon CPU E5"
    - 8 cores in total
  - AMD graphics card "Radeon HD 7900 Series"
    - 32 clusters
    - 64 SIMD units each
- Software



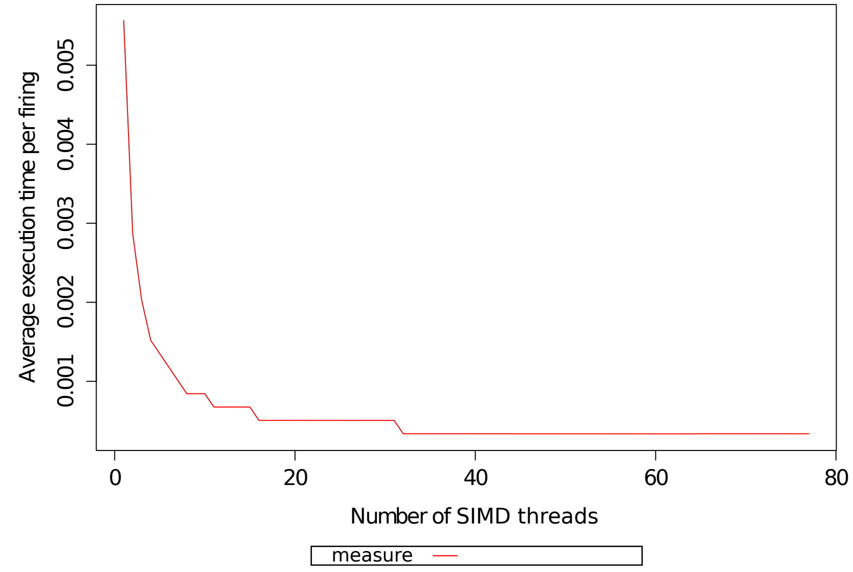
Experimental Evaluation - Process Performance Model



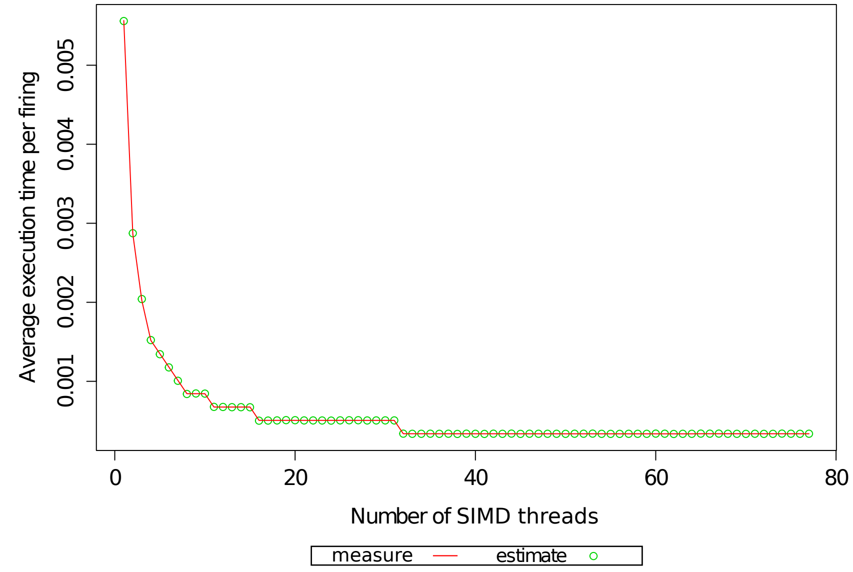
Experimental Evaluation - Process Performance Model



Experimental Evaluation - Process Performance Model



Experimental Evaluation - Process Performance Model



Summary

- Performance estimation is possible if execution time and time per firing batch known
- Execution time and time per firing batch can be modelled
- The developed models match experimental evaluation

Conclusion

- Throughput depends logarithmically on parallelism
- Latency scales linearly with parallelism

Bibliography

[1] L. Thiele, I. Bacivarov, W. Haid, and K. Huang, "Mapping applications to tiled multiprocessor embedded systems," in *Application of Concurrency to System Design, 2007. ACSD 2007. Seventh International Conference on.* IEEE, 2007, pp. 29–40.

[2] A. H. Ghamarian, S. Stuijk, T. Basten, M. Geilen, and B. D. Theelen, "Latency minimization for synchronous data flow graphs," in *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on.* IEEE, 2007, pp. 189–196.

[3] Khronos. Opencl documentation. [Online]. Available: http://www.khronos.org/opencl/

[4] L. Schor, I. Bacivarov, D. Rai, H. Yang, S.-H. Kang, and L. Thiele, "Scenario-based design flow for mapping streaming applications onto on-chip many-core systems," in *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems.* ACM, 2012, pp. 71–80.

[5] L. Schor, A. Tretter, T. Scherer, and L. Thiele, "Exploiting the parallelism of heterogeneous systems using dataflow graphs on top of opencl," in *Embedded Systems for Real-time Multimedia (ESTIMedia), 2013 IEEE 11th Symposium on.* IEEE, 2013, pp. 41–50.

[6] E. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *Computers, IEEE Transactions on*, vol. 100, no. 1, pp. 24–35, 1987.

[7] G. H. Hardy, E. M. Wright, D. R. Heath-Brown, and J. H. Silverman, *An introduction to the theory of numbers.* Clarendon press Oxford, 1979, vol. 4.