Speeding Up Reinforcement Learning With Graphics Processing Units

Jorn H. Postma

Master of Science Thesis

TU Delft Delft University of Technology

Faculty of Mechanical, Maritime and Materials Engineering (3mE)

SPEEDING UP REINFORCEMENT LEARNING WITH GRAPHICS PROCESSING UNITS

JORN H. POSTMA

Master of Science Thesis

BioRobotics
Department of BioMechanical Engineering
Faculty of Mechanical, Maritime and Materials Engineering (3mE)
Delft University of Technology

May 2015

SUPERVISORS:
Dr. Wouter Caarls
Prof.dr.ir. Pieter P. Jonker

PREFACE

This report is the master's thesis of the BioRobotics master specialization at the Delft University of Technology. BioRobotics is a specialization within the BioMechanical Design track of the Mechanical Engineering master's programme.

No prior knowledge of either graphics processing units or reinforcement learning is assumed; the report can serve as an introduction to both subjects. Readers interested in the workings of modern graphics processing units are referred to Chapter 2, or Chapter 3 for GPU programming using OpenCL. An overview of the theory behind reinforcement learning can be found in Chapter 4.

I would like to thank Wouter Caarls and Pieter Jonker for providing me with the opportunity to graduate on the subject of applying reinforcement learning techniques to graphics processing units. I would like to thank Wouter Caarls for his continued support, for stimulating me to thoroughly dive into two subjects with which I was not very familiar when I started, and for helping me write this report to the best of my abilities.

ABSTRACT

Conventionally programmed systems (e.g. robots) are not able to adapt to unforeseen changes in their task or environment. Reinforcement learning (RL), a machine learning approach, could grant this flexibility. Many fields of work could greatly benefit from this, be it in terms of cost, time or some other parameter. With RL, a learning agent tries to maximize its obtained reward during its interaction with a (maybe partially) observable environment. When the environment or even the task changes, the agent notices this and will change its behavior in order to keep its reward maximized. However, in most practical cases with large, if not continuous state and action spaces, converging towards a decent behavioral policy takes too much time to be of real use. Parallelizing RL algorithms might solve this problem. Whereas a modern multi-core central processing unit (CPU) has only a handful of cores, a graphic processing unit (GPU) has hundreds. The goal of this report is to show that fitted Q iteration (FQI), a tree-based RL method, can achieve significant speedups by parallelizing it on a GPU.

The GPU was invented to speed up the generation of images, as this is a process requiring raw computational power rather than flexibility granted by the large memory caches as found on a CPU. A large part of the CPU's caches was therefore replaced by computing cores. As a consequence, memory communications on a GPU are relatively slow and can greatly limit program performance.

Speedups with respect to (multi-core) CPU applications can only be achieved if the application applies repetitive computations to many independent data elements. There should be far more computational instructions than memory transfers. To reduce memory latency, the data can simply be distributed over the on-chip memory (*distribution*), in tiles if necessary (*tiling*), or it can be streamed through it in a predetermined way (*streaming*). Furthermore, multiple cores should be able to use the data of one global memory transaction.

Sequential and parallel implementations of FQI's KD-Trees and Extra-Trees tree-building methods were made using OpenCL and tested using the Puddle World task on an NVIDIA C2075 GPU. KD-Trees has excellent parallelization potential and adequate learning performance, whereas Extra-Trees has excellent learning performance but is more difficult to parallelize. Correspondingly, KD-Trees achieved speedups exceeding 100 times, while Extra-Trees achieved speedups of around 20 times. KD-Trees could furthermore solve much larger problems, achieved greater speedups at small problems and was less memory intensive. Despite this and the fact that learning times with KD-Trees were hundreds of times smaller than those of Extra-Trees, KD-Trees needed many more samples to find optimal solutions. The choice between KD-Trees and Extra-Trees thus comes down to the nature of the problem: if few samples are available Extra-Trees is the better choice, but with more samples and a more time-critical task KD-Trees would be preferred.

Future research could further optimize the parallel implementations (e.g. by combining multiple parallelization strategies). Applications of the implementations in the real world could be researched. Also, the parallelization potential of other RL algorithms could be investigated.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

LIST OF ALGORITHMS

1 INTRODUCTION

Conventional programming consists of writing code that is used to perform a specific, pre-defined task. However, as soon as the nature of the task or any aspect of the environment changes, the application no longer functions correctly. The programming code needs to be altered in order to reflect the change. *Flexibility* in the behavior of a system is something many fields of work would greatly benefit from, be it in terms of cost, time or some other parameter. There is a need for systems that *adapt* to changes.

This is where machine learning comes in. It is one way of solving such problems. Machine learning is a field of study in which it is tried to improve the performance of a system on a certain task using gathered experience. Reinforcement learning is one such machine learning approach. It is a promising technique, concerned with a learning agent interacting with a potentially unknown environment. The agent should be able to sense the state of its environment to some extent. By performing actions, the agent influences the environment. By programmatically defining a reward for certain combinations of environmental states and actions taken by the agent, the agent is instructed to maximize the total obtained reward during its course of actions. By interacting with its environment, the agent is able to learn *by itself* what to do in order to maximize this reward. What's important is that the behavior of the agent is dynamic: as soon as the environment or even the task itself changes, the agent notices this and will change its behavior in order to keep its reward maximized. This allows for applications that have not been possible with pre-programmed systems.

However, the learning speed of real-world reinforcement learning systems often leaves something to be desired. On small problems with small, discrete state and action spaces, reinforcement learning can be applied with good results. But in the real world these limited cases rarely occur. In most practical cases the state and action space will be large, if not continuous. Converging towards a decent behavioral policy then frequently takes too much time to be of real use. Another problem is the fact that learning in the real world might be dangerous, as the agent itself or something in its environment might be harmed.

It is because of this that recently attempts have been made to *parallelize* reinforcement learning algorithms. With today's multi-core central processing units (CPUs), new, powerful parallel programming possibilities have come to be, from which the reinforcement learning field is likely to benefit greatly. But besides the CPU there is also the GPU: the *graphics processing unit*. The architecture of the GPU is such that it can colorize the thousands of pixels of a computer screen in no time. A GPU can potentially perform calculations orders of magnitude faster than any CPU can, but the program's algorithm must suit the GPU's architecture.

What if a reinforcement learning algorithm could take advantage of the *hundreds* of computing cores of a GPU, instead of the four, eight, or possibly twelve cores inside a modern, high-end CPU? The learning speed and accuracy of the result could very well be unparalleled. Learning agents could adapt to a changing environment faster and could act with more accuracy. Current reinforcement learning applications

could be extended to larger and more complex environments. New applications of reinforcement learning could even arise, which would have been infeasible before.

The field of performing general-purpose computations on GPUs (GPGPU) is quite new, but develops rapidly. With the advent of a myriad of programming frameworks and language standards (of which CUDA and OpenCL are notable examples), this field will only continue growing in the near future. Algorithms of many existing fields of study will be implemented on a GPU.

The goal of this report is to show that reinforcement learning can achieve significant speedups by parallelizing it on a GPU. Specifically, the Fitted Q Iteration (FQI) algorithm as described by Ernst et al. [9] is parallelized using OpenCL. An answer to the following question will be given:

> *Can Fitted Q Iteration achieve significant speedups by parallelizing its algorithm and executing it on GPU devices using OpenCL?*

OpenCL version 1.1 is used for all programs. All results are obtained by running the programs on an NVIDIA C2075 (Fermi) workstation GPU and comparing them to a sequential version of the same programs. To measure the performance of the parallel algorithm, a variation of the Puddle World task [34] is used.

The structure of the report is as follows. In Chapter 2 it is explained what a GPU is, why it exists and what the differences with CPUs are. The architecture of modern GPUs is discussed, followed by a description of how the GPU executes programs. This chapter also shows what type of programs typically achieve speedups on GPUs and highlights several programming paradigms to take as much advantage as possible of the great computational power of a GPU. Chapter 3 then explains what OpenCL is and why it was chosen to write GPU programs. OpenCL is used to delegate work to CPU and GPU devices, and internally uses generic models for their architecture. In Chapter 4 the reinforcement learning problem is discussed. RL algorithms suitable to program on a GPU are shown and the choice for FQI is explained. Existing parallel implementations of reinforcement learning are also shown. After that, Chapter 5 shows the inner workings of the FQI algorithm. The reason for its GPU-programming potential is explained in detail, as well as the parallelization strategy used in the rest of the report.

The remaining chapters show the actual implementations of parallel FQI that were made and their performance. Chapter 6, in which the experimental setup is described, starts by showing the sequential and parallel implementations, followed by a description of the Puddle World task and the system that was used for the benchmarks. In Chapter 7 the speedups achieved and other results of the experiments are given, comparing computational times of single-core implementations against their GPU equivalents. A discussion of the results is done in Chapter 8. The report is finished with a list of conclusions and recommendations for future work in Chapter 9.

2

GRAPHICS PROCESSING UNITS

In order to successfully speed up reinforcement learning with GPUs, it is first necessary to look at the purpose and structure of this electronic circuit. This chapter starts with an overview of what a GPU is and why it exists in the first section. Differences with CPUs are highlighted. The second section explains the architecture of modern GPUs. The way a GPU schedules its work is the subject of the third section, followed by an elaboration on writing high-performance GPU programs in the fourth section. The chapter finishes with a summary of what was explained.

2.1 WHAT IS A GPU AND WHY DOES IT EXIST?

To understand why GPUs are built the way they are, it is important to know why they were invented in the first place.

2.1.1 *Transistor Types*

There mainly exist three types of transistors in CPUs and GPUs [29]:

1. **Datapath transistors**
   Transistors performing computations
2. **Control transistors**
   Transistors controlling branching and program flow
3. **Data storage transistors**
   Transistors used for storing data

CPUs are designed such that the user experience for everyday computer tasks is as smooth as possible. The CPU's design is focused on minimizing *latency*: the time between a request and the response of the computer. The CPU should be able to quickly process anything the user throws at it. It needs to efficiently handle a great variety of different tasks. It is therefore important for the CPU to have a large amount of control transistors, so that complex branching structures can be executed rapidly.

To further reduce latency, memory transfer delays should be kept as small as possible. Over the last decades processor clock speeds have increased at a faster rate than memory transfer speeds [41]. This means that memory latency has increasingly become the bottleneck of program performance. The CPU overcomes this by incorporating a large amount of data storage transistors that act as *cache* memory – small layers of memory with very low latency.

The GPU on the other hand is made to perform a task very different from the tasks a CPU performs: it should generate an image to be displayed on the user's display as fast as possible. This basically consists of performing the same computation for every pixel on the screen. So, whereas the CPU is designed to perform many small, complex and different tasks, the GPU needs to perform an enormous amount of the

same computations. The GPU is thus focused on maximizing the amount of computations per time unit (the *throughput*) rather than minimizing the latency. It therefore has a large amount of datapath transistors at the expense of control and data storage transistors.

This difference in transistor usage is shown in Figure 2.1, and it implies two reasons why efficient GPU programming is difficult. Firstly, on a CPU the programmer has a lot of cache space to store the program's data in. Since the cache is much smaller on a GPU, memory usage has to be planned more explicitly. Secondly, the smaller amount of control transistors means that it is harder to efficiently execute elaborate branching structures on a GPU. Dealing with these problems and other common GPU programming considerations is the subject of Section 2.4.

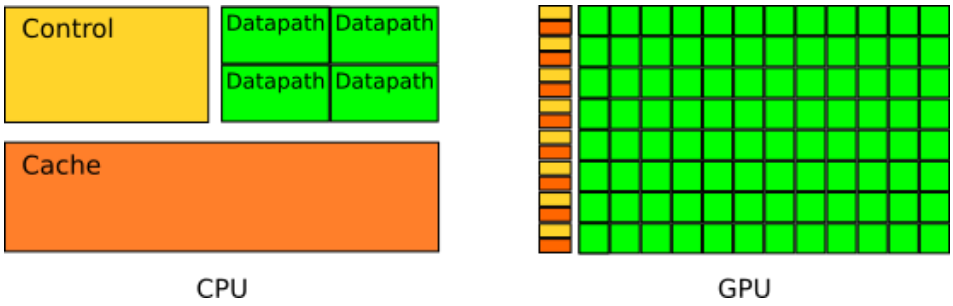


Figure 2.1: Difference in transistor usage between a CPU and GPU.
(Adapted from http://docs.nvidia.com/cuda/cuda-c-programming-guide/graphics/gpu-devotes-more-transistors-to-data-processing.png)

2.1.2 *The Need for Parallel Computing*

Nowadays, an important characteristic of a CPU or GPU is the amount of *computing cores* [1] it has. A computing core is a collection of transistors able to independently execute sets of program instructions (*threads* [1]). The *clock speed* (measured in Hertz) of a core is an important measure of its computational performance. Simple instructions require one clock cycle, whereas more complex instructions require many more. So the greater the core's clock speed, the faster it is able to execute instructions.

In the time that there were only single-core CPUs, increasing the clock speed of this core was the primary method of increasing computational performance. However, an increasing clock speed means an increase in heat generated. Today, clock speeds are so high that increasing them further would result in an amount of heat being generated that prevents the core from functioning correctly [10]. In order to maintain the trend of increasing computational performance, *multi-core* processors were introduced to allow multiple threads to run simultaneously. It lead to the rise of *parallel computing*. A program typically has three types of parallelism it can exploit [29]:

1. **Task parallelism**
   Performing multiple tasks simultaneously. *Example: core A performs task 1, core B performs task 2.*

2. **Data parallelism**
   Performing the same instructions of the same task on multiple parts of data simultaneously. *Example: core A performs task 1 on the first half of array elements, core B on the second.*

3. **Instruction parallelism**
   Performing multiple instructions of a single task simultaneously. *Example: computation of* `a` *and* `b` *is independent in the lines* `a = v+w; b = x+z;` *and can thus be performed simultaneously.*

The amount of cores on a chip depends on the type and amount of tasks to be performed. As mentioned, the CPU needs to handle many different complex tasks. A CPU therefore has a handful of complexly structured cores. The GPU on the other hand needs to perform the same computations for every pixel on the screen. Images could thus be generated faster when data parallelism is exploited. To this end the GPU has hundreds of smaller, simpler cores to colorize many pixels at the same time.

2.2 ARCHITECTURE AND MEMORY

Every GPU is different. For GPU programming it is important to know what the architecture of a modern GPU looks like. This section discusses the NVIDIA Fermi GPU architecture [26]. The NVIDIA C2075 used in the experiments contains the NVIDIA GF110 architecture [24], which is based on Fermi. Although Fermi is just an example of a GPU architecture, many of the concepts shown hold for any modern GPU.

A simplified overview of the Fermi architecture is given in Figure 2.2. The GPU consists of several *streaming multiprocessors* (SMs)[1]. Each SM contains cores, registers and shared memory / L1 cache[2]. Although the figure only shows eight SMs with eight cores each, modern GPUs have many more SMs and cores. The C2075 is equipped with 14 SMs, each containing 32 cores.

Several memory layers exist in this architecture:

1. **DRAM (video memory) (C2075: 6 GB)**
   Large storage space which any core can access.

2. **Level 2 (L2) cache (C2075: 768 kB)**
   Cache which any core can access.

3. **Shared memory / Level 1 (L1) cache (C2075: 64 kB per SM)**
   Shared between cores within an SM – other SMs cannot access it. Used for fast sharing of data within an SM. It can be configured with either 48 kB L1 cache and 16 kB shared memory, or as 16 kB L1 cache and 48 kB shared memory.

4. **Registers (C2075: 128 kB per SM)**
   Each core has its own chunk of registers which is only accessible by that core.

1 NVIDIA terminology.
2 An actual Fermi SM also contains elements like instruction caches, warp schedulers and dispatch units.

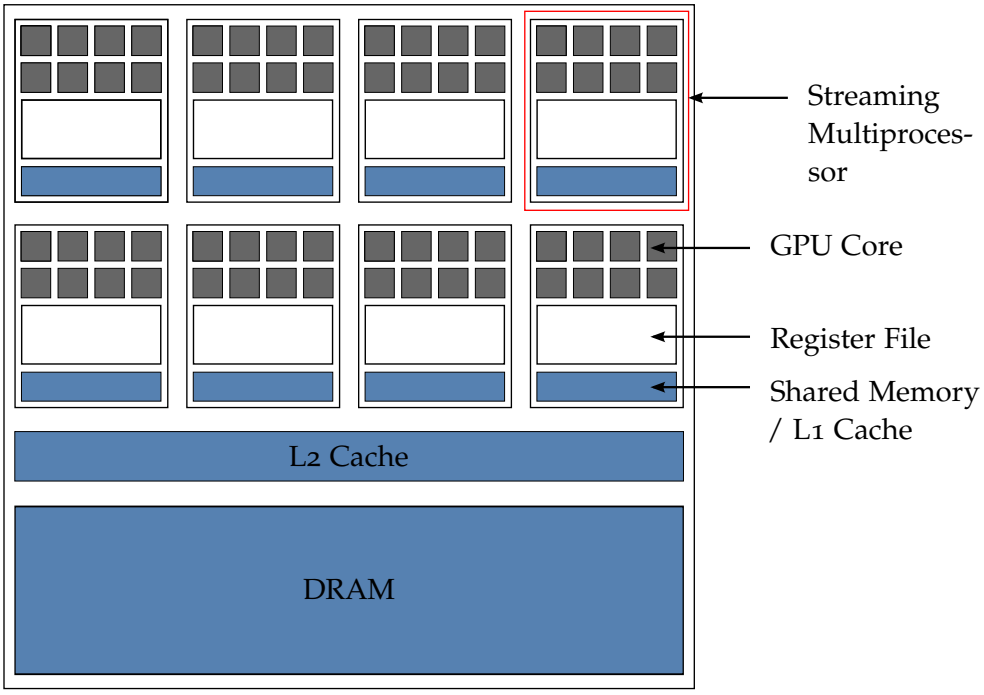


Figure 2.2: Hardware architecture of a GPU. A simplified version of the NVIDIA Fermi architecture [26] is shown.

Note that the L1 and L2 caches cannot be programmed and are filled as the GPU itself sees fit during program execution. The L2 cache stores global memory data that is likely to be used again by any SM in the near future, based on *locality* [13] of the data. This can for example be data that has been used recently (*temporal locality*) or lies directly next to data that was just accessed (*spatial locality*, e.g. surrounding elements in an array). The L1 cache is used for the same purpose, but this time for a specific SM.

An important notion to keep in mind is that most memory transfers on a GPU are slow. When writing a GPU program, the amount of memory transfers should be kept to a minimum, as the high throughput of a GPU is useless when the program's memory communication takes more time than its computations. With memory transfers, in general the following rule holds: the closer to the cores the data is stored, the faster it can be accessed. So, the memory layers shown in the list above are ordered by access speed: DRAM is the slowest layer, whereas the registers are fastest.

Exact figures on the latency of the memory layers are hard to find in literature. Parakh et al. [31] have researched the latency of a Quadro600 Fermi GPU. They found that a DRAM transfer can take 750 clock cycles to complete, L2 cache access took 350 cycles and L1 took 50 cycles. Comparing that to the fact that it only takes two cycles to perform for example an integer operation, it is clear that performance can be limited severely by having too many global memory transfers.

2.3 WORK SCHEDULING

On a GPU the smallest unit of work is not a thread, but a group of threads. On NVIDIA GPUs these groups are called *warps* [26], consisting of 32 threads[3]. Threads within a warp execute the same statement at the same time.

This has a consequence when writing GPU programs that contain conditional statements which might lead to branching. Consider the example code shown in Listing 2.1. What if within a warp the condition `x == 0` is true for threads 1 to 16 and false for 17 to 32? Since all threads within a warp execute the same statement, first *all* 32 threads will evaluate Branch 1, with only threads 1 to 16 actually keeping the result of the statements. Then, all 32 threads will evaluate Branch 2, with threads 17 to 32 keeping the results. So effectively half of the threads are waiting half of the time, dividing program performance by two. This problem can be avoided by limiting the amount of code branches in the program, or making sure all threads within a warp execute the same branch.

Listing 2.1: Example of code branching.

```
1 if (x == 0) {
2     // Branch 1
3 } else {
4     // Branch 2
5 }
```

A GPU can schedule multiple warps on every SM. During program execution, the SMs are able to switch between the warp they are currently executing. So, while Warp 1 is waiting for data to arrive from DRAM, the SM can already execute some instructions of Warp 2. As soon as the data of Warp 1 has arrived, the SM switches back to Warp 1 and continues. This is called *latency hiding* [2]. It reduces the penalty of memory transfers. The more warps are scheduled on an SM, the better the latency can be hidden.

2.4 WRITING HIGH-PERFORMANCE PROGRAMS

The above has shown that the architecture of a GPU is very different from that of a CPU. This means that a specific approach needs to be taken to programming if programs are to be run efficiently on a GPU. Not all programs are suitable to be applied to a GPU. In order for a GPU implementation of a program to be significantly faster than a CPU one (be it single-core or multi-core), the nature of the program has to meet certain requirements. And even then, care should be taken to focus the program on the GPU's architecture. Only then can it benefit from the GPU's full throughput. Not doing so can drastically limit the performance of a program, even to such an extent that a sequential implementation would run faster. This section explains what kind of programs are suitable to be applied to a GPU, and explains how to write programs that take full advantage of one.

3 On AMD GPUs the groups of threads are called *wavefronts*, consisting of 64 threads. In the remainder of this report the term *warp* is used, as an NVIDIA GPU was used during the experiments.

2.4.1 *Characteristics of Parallelizable Programs*

To take full advantage of the extreme throughput of a GPU, a program has to meet the following requirements:

1. **All cores need to be kept busy**
   Unused or waiting cores decrease the potential efficiency of the program.
2. **The amount of memory transfers should be limited**
   In general, memory transfers on a GPU are slow. Waiting for data to arrive slows the program down.

These requirements are met by programs that expose the following three characteristics [14]:

1. **Data parallelism**
   The same operation can be applied to many data elements. Since all threads in a warp execute the same statement at the same time, each operation should be applicable to many data elements.
2. **Data independence**
   There is no (or little) dependence between the processing of different data elements. This way memory transfers between cores and SMs are kept to a minimum. Plus, cores are not waiting for other cores to complete certain computations.
3. **High computational intensity**
   GPU programs have a high *computational intensity*, which is the ratio between computational instructions and memory transfers:

$$\text{computational intensity} = \frac{\text{computational instructions}}{\text{memory transfers}} \tag{2.1}$$

When a problem shows signs of data parallelism and independence, and moreover has a high computational intensity, it is highly likely that speedups can be achieved by programming it for a GPU.

2.4.2 *Approaches to Minimizing Latency*

As mentioned before, it is important to keep data as close to the cores as possible to limit memory latency. This section describes common ways of minimizing latency, loading data from DRAM to the cores, and keeping it as close as possible.

First and foremost, it is important that DRAM transfers use *coalesced access* [27, Section 3.2.1]. Every DRAM transaction has a minimum size of data being transferred. For the NVIDIA C2075 this is 128 bytes when accessing 4 byte elements[4] [27, Section 3.2.1]. So, if a single thread needs to load a 4 byte integer array element, still a memory transaction of 128 bytes (i.e. 32 integers) is made.

If that single thread is the only one using the data transferred, then only 4 of the 128 bytes are used, while the rest of the bytes has been transferred for nothing

4 For single-byte elements the segment size is 32 bytes; 64 bytes for two-byte elements and 128 bytes for four bytes or more.

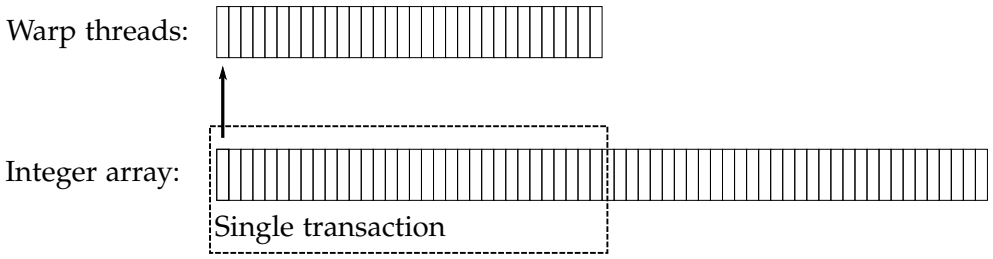


Figure 2.3: A single thread of a warp accessing a single integer in a DRAM transaction. Only one element of the data transfer is used, the other elements have been loaded for nothing.

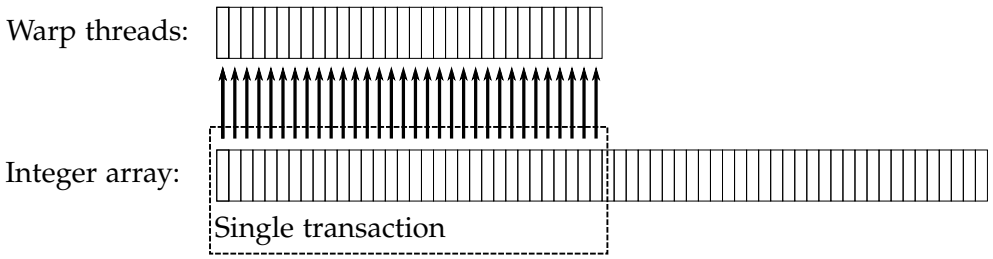


Figure 2.4: All threads of a warp accessing data using a single DRAM transaction. Every element in the transaction is used. This is an example of *coalesced memory access*.

(Figure 2.3). Fortunately, it is possible to take advantage of the transaction. Other threads inside the same warp can use the other elements that came with the transaction. So in the ideal case all 32 threads inside the warp have loaded data from DRAM in only a single transaction, as shown in Figure 2.4.

This shows the importance of threads inside a warp accessing successive elements in memory. If the threads access data in an uncoalesced manner, at worst 32 transactions of 128 bytes could be made. Since each global memory transaction on a Fermi GPU takes about 750 clock cycles (as was shown in Section 2.2), as many as 24000 clock cycles could be used to transfer the data in a worst-case scenario. If access is coalesced, this could be done in a single transaction of 750 cycles. The consistency with which memory access is coalesced can make the difference between having great speedups or no speedups at all.

To keep the amount of DRAM transfers to a minimum, several approaches exist to keep the data as close to the cores as possible:

1. **Storing data in private memory**
   The registers are the fastest memory layer, so if data used by the cores fits there, it should be stored there.

2. **Storing data in shared memory**
   Data can also be distributed over the shared memory of the SMs if multiple elements are used by multiple threads. Cores can then operate on all data without needing to fetch the data from DRAM. If the data is too big to be distributed over shared memory, *tiling* [27, Section 3.2.2.2] can be used, with which the data is distributed over the SMs in parts (tiles) that fit into shared memory. Cores then operate on a tile, after which the next block is loaded and the operations are performed again.

3. **Streaming data from DRAM**
   If the data to be used is too big for either registers or shared memory, it should be *streamed* from DRAM [14]. It starts with Warp 1 loading data from DRAM. While this warp performs its computations using that data, Warp 2 can load its data from memory in the meantime. As soon as Warp 1 is done with its computations, Warp 2 ideally has its data loaded and can start *his* computations, while Warp 3 loads its data from memory. This process continues until all warps have done their work. This way there is a continuous *stream* of computation and communication in which the memory latency is mostly hidden.

2.5 SUMMARY

Whereas a CPU is designed to be flexible on many different tasks, a GPU is designed for maximum throughput. Many of the CPU's cache transistors are traded for computing cores – a modern GPU has hundreds or even thousands of cores. The GPU's great throughput and the lack of cache space means that memory transfers are costly.

The smallest unit of work on a GPU is a warp, consisting of 32 threads. All threads within a warp execute the same statement at the same time. A GPU can switch between warps to hide memory latency. A program is likely to be fast on a GPU if it has possibilities for data parallelism, has a high computational intensity and if elements are independent of each other. Coalescing memory access is key to minimizing the amount of memory transactions. Smart data distribution, like storing data in registers or using tiling or streaming, can further hide memory latency.

3 OPENCL

OpenCL (Open Computing Language) [12] is the parallel programming framework that was used to write GPU programming code. It is an open language standard which defines programming interfaces to execute code in a parallel fashion on GPUs and multi-core CPUs. Vendors of GPUs and CPUs can implement the OpenCL standard in their drivers, after which programmers can write OpenCL code that can be executed on devices supporting that driver. This means that for example AMD and NVIDIA have separate implementations of OpenCL. The two main reasons for choosing this framework are:

1. **Platform independency**
   OpenCL code does not have to be run on a card of a specific brand or type. Code developed for one system can easily be run on another system. This also facilitates development, as programs can be tested on a development computer first, before deploying the program to the target computer.

2. **Heterogeneous computing**
   OpenCL allows code to be run on GPUs as well as multi-core CPUs. The principle of taking advantage of both types of processors is called *heterogeneous computing*. This results in a wide applicability of the programs, as they can run on almost any device. Plus, even though this report only compares the OpenCL implementations with single-core CPU implementations, multi-core CPU execution times could be compared directly to GPU times.

This chapter explains what OpenCL is, how it operates and what an example program looks like. The first section elaborates on OpenCL's generic device model, which allows for heterogeneous computing and platform independent code. The structure of OpenCL programs is the subject of the second section, followed by a short summary of the chapter in the third section.

3.1 THE OPENCL ARCHITECTURE

Since OpenCL code can be executed on CPUs and GPUs of many vendors, it cannot assume one specific hardware architecture. CPUs and GPUs have fundamentally different hardware architectures, as was shown in the previous chapter. Even two GPUs can be different in this respect. So, if code is to be run on multiple devices, an *abstraction* should be made for the architecture of a device. To this end, OpenCL defines three distinct models in its architecture [12, Chapter 3]:

1. Device model
2. Memory model
3. Execution model

The device model abstractly defines the interconnection of devices and the architecture of a device. The memory model defines the general layout of device memory.

Lastly, the execution model determines how work is mapped onto the devices. To write efficient OpenCL programs, it is important to understand what these models look like, how they operate, and how they communicate with each other. In this section each of these models is described.

3.1.1 *Device Model*

OpenCL relies on the fact that a *host* is connected to a number of *computing devices*. The host is the computer which delegates work to one or more devices, e.g. CPUs or GPUs. The combination of a host and its devices is called a *platform* (Figure 3.1).

The actual device model (i.e. the inner model used for a computing device) is shown in Figure 3.2. Every device has one or more compute units (CUs), which in turn have one or more processing elements (PEs). All instructions of an OpenCL program are executed on the PEs.

Note the differences in nomenclature with the Fermi GPU architecture as shown in Figure 2.2 of the previous chapter. An SM on a GPU is represented by a CU in OpenCL, whereas the GPU cores are called PEs in OpenCL.

3.1.2 *Memory Model*

Figure 3.2 also shows the layers of memory of an OpenCL device. They closely resemble the layers shown in the Fermi architecture:

1. **Global memory**
   DRAM (video memory) on a GPU. Data pushed from the host is stored here.
2. **Constant memory**
   A read-only part of global memory. Devices have optimizations for PEs to access this layer quickly.
3. **Local memory**
   Shared memory on a GPU.
4. **Private memory**
   Registers on a GPU.

As can be seen, the OpenCL memory model does not specify every memory layer of the GPU. The L1 and L2 caches are not modeled. Recall from the previous chapter that these cannot be programmed anyway and are filled by the GPU itself during runtime. Another difference is that OpenCL assumes every PE to have a separate piece of private memory. On a GPU there is one chunk of registers per SM, which is divided over the cores.

It might seem counterintuitive to rely on this model when one knows what the architecture of the actual device looks like. However, to successfully allow heterogeneous computing, the programmer needs to write his code for this model. OpenCL takes care of targeting the optimal parts of the device.

3.1.3 *Execution Model*

A host program is used to interact between the host and a device. The host program selects the platform and devices to be used. It also builds the OpenCL device

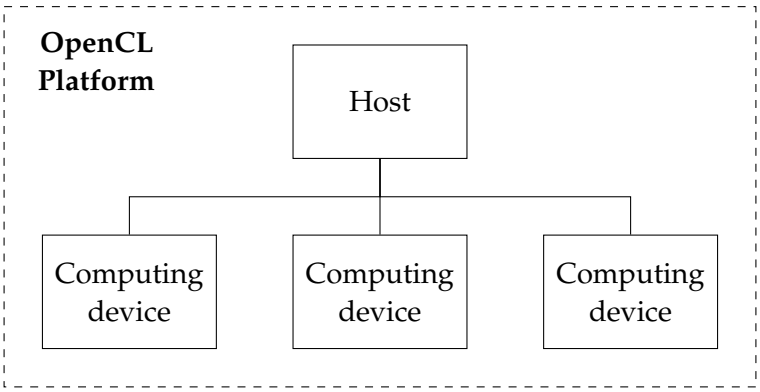


Figure 3.1: An OpenCL platform: a host and connected computing devices.

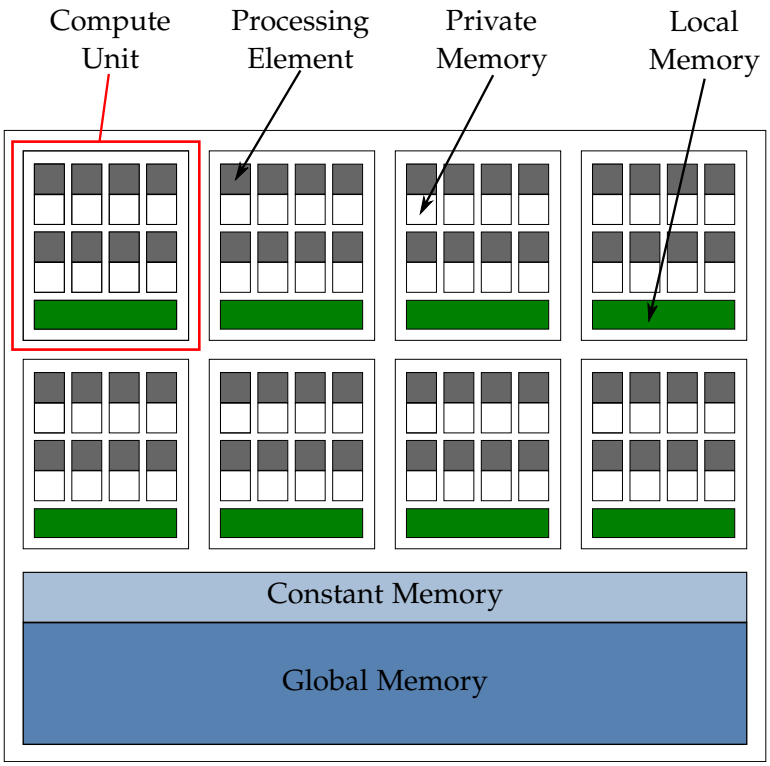


Figure 3.2: The device model of OpenCL.

code (code to be executed on for example the GPU) and pushes it to the device for execution. It is furthermore used to read and write data to and from the device.

When device code is to be executed, the amount of threads to be used on the device needs to be specified. In OpenCL terms, a GPU thread corresponds to a *work-item*. Each work-item is executed on a processing element of a compute unit. Work-items are grouped into *work-groups*.

With OpenCL, groups of threads are specified using *NDRanges* (n-dimensional ranges). Threads can thus be specified in multiple dimensions – most devices support up to three. Before executing device code, two NDRanges need to be specified by the programmer in the code:

1. **Global NDRange**
   The global NDRange specifies how many work-items are to be used in total.

2. **Local NDRange**
   The local NDRange specifies the size of the work-groups inside the global range.

Example: Consider device code performing a computation on every element of a 12x12 matrix. To ensure every element is processing in its own work-item, a global NDRange of 12x12 is specified. This global grid can be split up into work-groups of 4x4 work-items by setting the local NDRange to 4x4.

The above shows that the programmer is able to specify the work-group size to use. OpenCL will then schedule the device code with that specific size. This scheduling by *software* as done by OpenCL should not be confused with scheduling by the GPU *hardware* as was explained in Section 2.3. OpenCL distributes work-groups over the CUs of the GPU, after which the GPU itself makes the right warps execute at the right time. The work-group size should in principle always be a multiple of the warp size. A work-group with 20 work-items will be scheduled with one warp of 32 threads on the GPU, as the GPU itself cannot schedule anything smaller. Of those 32 threads only 20 will do actual work – the others are waiting. A work-group with 65 work-items will be scheduled using three warps, one of which only having a single thread doing work.

3.2 OPENCL PROGRAMS

The architecture described above is used by OpenCL programs, which consist of two components: host code and device code. But what does this code look like?

3.2.1 *Host Code*

First there is host code, which is executed on the host. It selects and prepares the devices to be used. It acts as a conductor for OpenCL programs. The device only does work if the host instructs it to. The host code specifies which pieces of device code should be executed, when they should be executed and what NDRanges should be used. It also determines when and which data should be pushed from the host to the device and vice versa. Typical host code executes the following steps [12]:

1. Fetch a list of OpenCL-compatible devices connected to the host and select the ones to use
2. Create one or more contexts based on these devices
3. Create a command queue for each device
4. Compile and build a device program
5. Create input and output data buffers and push them to the devices' memory
6. Set the data buffers as the arguments for the device code
7. Execute the device code
8. Listen for events and read the output from the devices' memory
9. Free the allocated memory.

3.2.2 *Device Code (Kernels)*

The device itself is only capable of executing functions that the host dictates. These functions are called *kernels* and are written in a subset of the ANSI C99 programming language, extended with extra functionality for parallel programming [12, Chapter 6].

Example Kernel

OpenCL kernel code can best be explained by showing a minimal example. A kernel used to square the elements in an array might look as shown in Listing 3.1.

Every kernel starts with the `kernel` keyword. Kernels cannot return values and therefore always return `void`. The arguments of this kernel are `in` and `out`, both pointers to a float (array) in global memory of the device, as specified by the `global` keyword.

In the body of the kernel the global ID of the current work-item is stored in a variable `i` with `get_global_id(0)`. The `0` parameter denotes that the ID of the work-item in the first dimension should be accessed. After that, the `i`'th element of the output array is set to the square of the `i`'th element of the input array.

Listing 3.1: A squaring kernel

```
1 kernel void square(global float* in, global float* out) {
2         int i = get_global_id(0);
3         out[i] = in[i] * in[i];
4 }
```

3.3 SUMMARY

This chapter explained what OpenCL is and how it works. OpenCL is an open language standard allowing parallel programming on GPUs as well as multi-core CPUs. Since the code should be capable of running on any type of device with any architecture inside, OpenCL uses generic models for the computing devices and their inner architecture. The programmer always programs against these models instead of the actual architecture of the device. OpenCL itself takes the architecture of the device and maps this to its models. An OpenCL program consists of host code

and device code. Device code consists of small functions called *kernels*, written in a subset of ANSI C99. A device only executes kernels when the host instructs it to.

4 REINFORCEMENT LEARNING

The OpenCL language standard has now been explained along with how a GPU interacts with it. This chapter explains the main theory behind the reinforcement learning (RL) problem, a machine learning problem that can be solved in many ways.

The first section formally defines the problem and highlights several of its main components. The second section explains the workings of Markov decision processes, a type of problem to which most RL problems belong. The third section explains what categories of RL methods exist, followed by common ways of tackling an RL problem in the fourth and fifth section. Section six shows the results of a literature research that was performed before writing this report. There, RL algorithms that are suitable and unsuitable for GPU implementations are shown, and previous parallelization attempts are highlighted. The chapter finishes with a summary in section seven.

4.1 DEFINITION AND KEY COMPONENTS

In this section the reinforcement learning problem is defined, and several of its key components are elaborated on.

4.1.1 *Definition*

Solving a reinforcement learning problem [35, Section 1] is the act of learning what actions to take in order to maximize a certain reward function. A reward is assigned to each possible situation (state of the environment) or state-action pair. The learner (agent) is not instructed what to do; it can only receive rewards from the environment.

Key aspects of the reinforcement learning problem are shown in Figure 4.1. Reinforcement learning is about *interaction* between the agent and its environment. The agent should be able to observe the state of the environment to an extent. Based on its observations, it performs an action which influences the environment. The agent is then given a reward (which can be zero or negative as well), depending on how the reward function is defined.

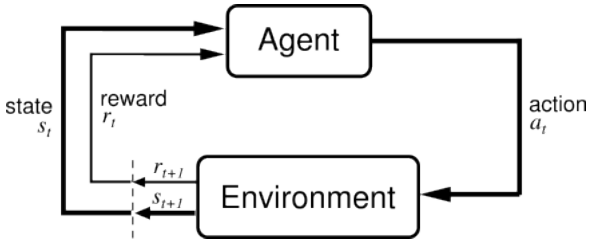


Figure 4.1: Interaction between the agent and environment.
(From [35, Section 3.1])

4.1.2 *Key Components*

The following components play a central role in solving a reinforcement learning problem:

1. **Reward function** - defines the immediate reward for reaching a state.
2. **Return** - defines the expected cumulative future reward for a state, when starting from that state.
3. **Value function** - stores the return for each state.
4. **Policy** - defines how the agent chooses its actions.
5. **Model** - returns the next state, given a current state and an action.
6. **Sample** - defines an experience datum.

Each of these will now briefly be explained.

Reward Function and Return

The reward function defines the desirability of states as a single number. It is defined by the environment. Whenever the agent reaches a certain state (or state-action pair), the reward defined by the reward function for that state is given. As such, the goal state might for example grant a reward of 100, a good intermediate state 50. A bad state (e.g. a walking robot falling over) might result in a negative reward being given.

The agent's objective is to maximize the total reward during its course of actions. This is defined mathematically as maximizing the sum of the expected reward after time step t: the **return**. The return is given by:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \quad . \tag{4.1}$$

Here, r is the reward and γ is a discount factor to prevent the return from going to infinity.

Value Function

The value function V then defines for each state the maximum possible return that can be obtained from the reward function over the agent's course of actions, starting from that state. The experiences of the agent build and refine this function. It is the agent's current knowledge of the environment. A certain state might not give the agent an immediate reward, but another state very near to this one could. In that case, this state receives a high value as it contributes to maximizing the total reward. Values can be assigned to states, in which case the value function is described by $V(s)$, or to state-action pairs by $Q(s, a)$.

Policy

The policy defines the behavior of the agent. It is the method used by the agent to choose its next action. The policy can be affected by the value function as well. For

example, a *greedy* policy means that the agent always chooses the action that will put it in the state with the greatest value.

An important consideration in RL is the policy's balance between *exploitation* and *exploration*. *Exploitation* means choosing the actions that result in the maximum expected total reward (a greedy action), by exploiting the current knowledge of the environment (i.e. the current value function). In other words, actions that result in states with high current value are preferred. *Exploration* on the other hand means performing an exploring action (e.g. randomly), despite the current knowledge of the environment. When the value function has not converged sufficiently, there is always a possibility that the current greedy action might not be the best action to take. An exploring action takes this into account. It is used to get a more accurate and complete value function.

Model

A model for the environment may or may not be available when solving a reinforcement learning problem. Having this model means that values and rewards can be estimated before a state is actually experienced. It provides a means for predicting the reaction of the environment to the agent's actions.

4.1.3 *Performing Trials*

Most reinforcement learning tasks have a fixed number of trials. During each of these trials, the agent attempts to reach its goal. Each set of (s, a, r, s') (i.e. at state s action a was taken, resulting in immediate reward r and new state s') is called a **sample**. The course of states or state-action pairs then results in a total reward for that trial. The value function can be updated using the rewards it has been given. After a sufficient number of trials, the value function will start to converge to the real (or optimal) value function. Based on this a policy can be chosen.

4.2 MARKOV DECISION PROCESSES

The great majority of RL problems are Markov Decision Processes (MDPs) [35, Section 3]. An MDP is a process that has the Markov property, which states that a future state of a system is only dependent on the system's current state and the policy. The sequence of states that happened before the current one is irrelevant. In terms of RL this means that the next state and reward are solely dependent on the current state and chosen action. For a system with stochastic transitions this means that the probability of reaching state s' by choosing action a in state s is:

$$\mathcal{P}^a_{ss'} = \Pr\left\{s_{t+1} = s' \mid s_t = s, a_t = a\right\} \quad , \tag{4.2}$$

with its corresponding expected reward:

$$\mathcal{R}^a_{ss'} = \mathrm{E}\left\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\right\} \quad . \tag{4.3}$$

Now, let $V^\pi(s)$ be the maximum possible return when starting a trial from state s, following policy π. For an MDP this can be written as $V^\pi(s) = \mathrm{E}_\pi\{R_t \mid s_t = s\}$. Substituting Equation 4.1 into this and rewriting then gives:

$$V^\pi(s) = \sum_a \pi(s, a) \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right] \quad , \tag{4.4}$$

which is known as the *Bellman equation*. From this, the very similar *Bellman operator* [19] T_π can be found:

$$(T_\pi V)(s) = \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right] \quad , \tag{4.5}$$

which can be used to iteratively let any initial V converge to the actual V^π. V^π is called the *fixed point* of the Bellman operator, as applying T_π to V^π would again give V^π since the solution has converged already:

$$T_\pi V^\pi = V^\pi \quad . \tag{4.6}$$

This process of finding the value function for a specific policy is called *policy evaluation* [35, Section 4.1]. However, the policy that was evaluated might not be the optimal policy possible – i.e. there might be states in which the action given by $\pi(s)$ is not the best action to take. The goal of learning is to find an *optimal* policy, $\pi^*(s)$. The corresponding optimal value function V^* can be found with the *Bellman optimality operator* T [35, Section 3.8]:

$$(TV)(s) = \max_{a \in \mathcal{A}} \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right] \quad . \tag{4.7}$$

If this operator is applied repeatedly to a value function, the value function will ultimately converge to V^*, such that:

$$TV^* = V^* \quad . \tag{4.8}$$

So, V^* is the fixed point of the Bellman optimality operator. Once V^* is known, finding an optimal policy π^* is trivial: in every state the agent should perform the action that gives the highest value (the $\max_{a \in \mathcal{A}}$ in the Bellman optimality operator).

4.3 TYPES OF REINFORCEMENT LEARNING METHODS

Solving the Bellman equation, or finding the fixed point of the Bellman operator, forms the basis of many RL methods that try to solve for the value function. These methods are called *value based methods*. On the other hand, instead of finding V, there can also be searched for an optimal policy immediately. These methods are called *policy search methods*.

Both value based and policy search methods can be either model free, model based or model learning. Model free methods do not use a model at all. Model based methods on the other hand rely on a perfect model of the system to solve the RL problem. Lastly, model learning methods are methods that try to learn and update the model of the environment during the learning process.

The way the agent stores its value function, reward function or policy (their *representation*) also differs between RL methods. Before diving into the actual RL methods it is useful to briefly discuss several representations, as they make up an important part of the methods and are used in value based and policy search methods alike.

4.4 REPRESENTATIONS

A property of RL methods that greatly affects the potential of a possible GPU implementation, is how the value function, reward function or policy will be stored within the agent: their *representation*. In this section the case of representing the value function will be discussed. However, the principles demonstrated can just as well be applied to represent the others.

4.4.1 *Lookup Tables*

Using a lookup table is the most straightforward way of storing the data. For each element (be it a state for V or π, or a state-action pair for Q), a value is stored. This is especially suitable for small problems with small sets of states and actions. When the problem gets bigger, it becomes too computationally expensive to sweep over the entire state-action space and update every value.

4.4.2 *Function Approximation*

In many practical applications of RL, the sets of actions and especially states are large. Furthermore, there is likely to be only a very small subset of the state-action space to be visited during a trial. So the question is: how can a full value function or policy be reconstructed from only a very small set of samples? *Function approximation* [35, Section 8] is used to extend this small set of experiences over the entire state-action space. The value function or policy is assumed to be a parametric function with weight vector θ_t. Therefore, the goal of function approximation is to find the weights that let the approximated function be as close to the optimal function as possible. Below, a couple of different ways of representing a function approximator are described.

Tile Coding

Assume the value function is a *linear sum* of weighted basis functions $\boldsymbol{\phi}_s$:

$$V = \theta_t^T \boldsymbol{\phi}_s \quad . \tag{4.9}$$

One representation which uses this assumption is *tile coding* [35, Section 8], which overlays a state-action space with several *tilings*. Each of the *tiles* inside a tiling are assigned a parameter and a binary basis function (zero if state s does not belong to the tile; one if it does). For example, take the two-dimensional state-action space shown on the left hand side of Figure 4.2. This space can be overlaid with a grid-like tiling structure as shown in the right in the figure. In this case, the point marked with 'x' belongs to the two highlighted tiles and its value is equal to the sum of the parameter values of these tiles.

Examples of different tiling structures are given in Figure 4.3. The structure determines the resolution of the value function in regions of the state-action space.

Tree-Based Models

Tree-based models are also used to represent a value function [9]. A tree splits up the state-action space into a number of segments which are subsequently split further

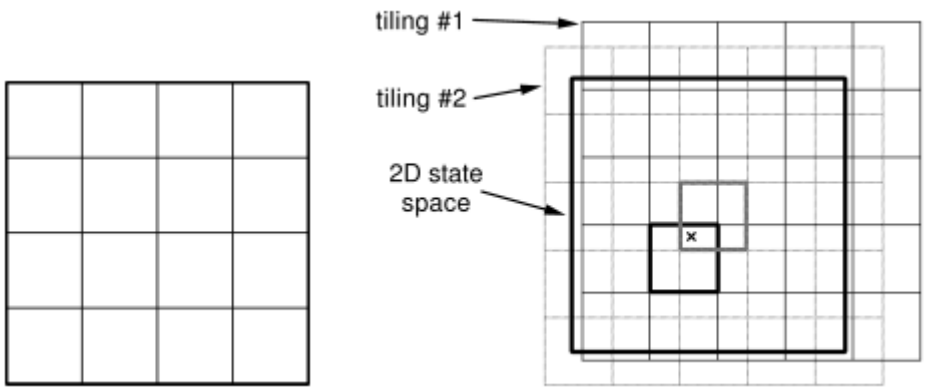


Figure 4.2: Tile coding: overlaying a two-dimensional state-action space (left) with grid-like tilings (right).
(From [35, Section 8])

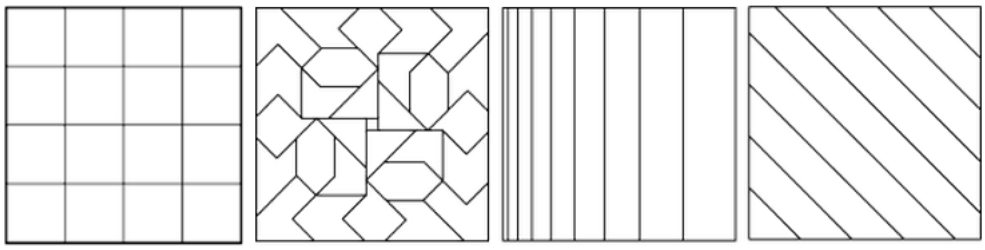


Figure 4.3: Four examples of tiling structures that could be used with tile coding.
(Adapted from [35, Section 8])

into smaller regions. An example is shown in Figure 4.4. The tree splits up the state-action space by defining a binary test at every node. These tests have format *cut-parameter* $<$ *cut-point*. The cut-parameter is a state- or action dimension, while the cut-point is a value for that dimension. The state-action space region for which the test is true makes up the left child node, whereas the region for which it is false makes up the right. A prediction of a region's value can be made by averaging the reward of the samples that belong to that region of state-action space. Tree-based supervised learning (described in Section 4.5.3) is used to determine a test at every node, and therefore determines *how* the tree splits up the state-action space.

Neural Networks

Neural networks [21] can also be used to approximate V. A neural network is a massively parallel model. A basic overview of its structure is given in Figure 4.5. An input pattern is propagated to one or more hidden layers, which process the input and form an output pattern. The hidden layers contain sets of nodes. Each node (or computational element) receives a combination of input elements, which are weighted, after which a mathematical function is applied to obtain an output element. This process is shown in Figure 4.6. After each node has computed its output value, the output values are weighted to form a final output pattern.

It should be clear that the used weights are the variables influencing the process from input to output. The goal of a neural network is thus to optimize the weights such that an optimal output pattern is generated from the input pattern.

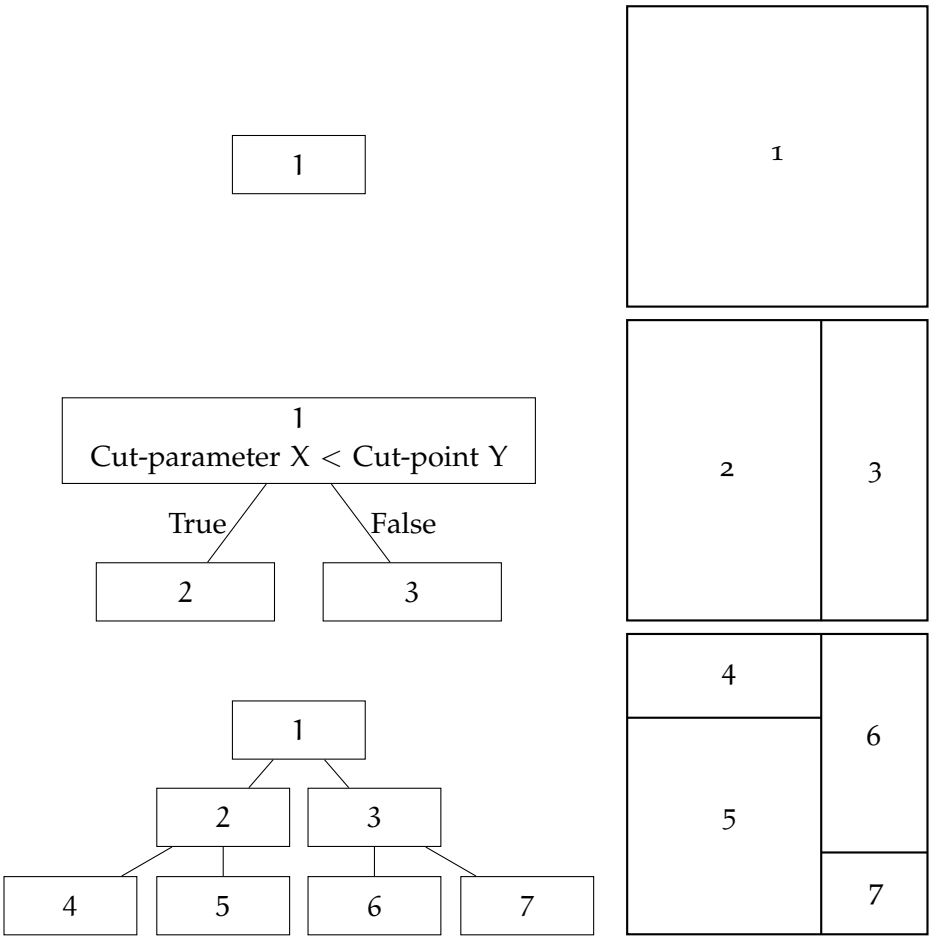


Figure 4.4: Example of a tree model representing regions of state-action space.

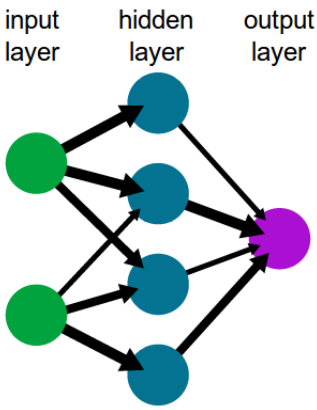


Figure 4.5: Basic layout of a neural network.
(From http://upload.wikimedia.org/wikipedia/commons/9/99/Neural_network_example.svg)

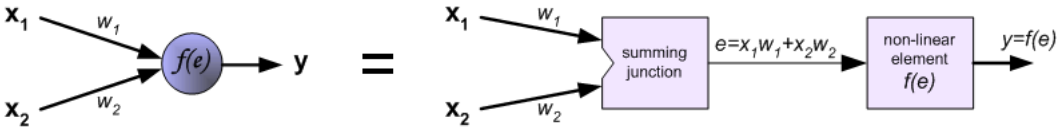


Figure 4.6: Inside a node, a function is applied to the weighted sum of input elements to determine an output element.
(Adapted from http://home.agh.edu.pl/~vlsi/AI/backp_t_en/backprop_files/img01b.gif)

4.5 SUPERVISED LEARNING METHODS

Supervised learning methods are ways in which samples are used to update the weights of a function approximation. When a representation has been chosen, it is important to decide how to map experience (samples) onto it. With lookup tables this might be trivial, since every state is assigned its own value. However, with function approximation this is not the case. In this section three important methods are described: gradient descent methods, least squares solutions and tree-based regression. Again, the example of approximating a value function will be described, but the principles demonstrated hold for a policy just as well.

4.5.1 *Gradient Descent*

Gradient descent methods [35, Section 8] work by minimizing the mean squared error (MSE) over the available samples. The MSE is given by:

$$\mathrm{MSE}(\theta_t) = \sum_{s \in S} P(s)\left[v_t(s) - V_t(s)\right]^2 \quad , \tag{4.10}$$

in which $P(s)$ are weights for the errors on the different states, $v_t(s)$ is the learning target for state s at time t, and $V_t(s)$ is the current value found for that state. After obtaining a sample, a new, improved set of parameters can be found using:

$$\theta_{t+1} = \theta_t + \alpha\left[v_t - V_t(s_t)\right]\nabla_{\theta_t} V_t(s_t) \quad , \tag{4.11}$$

in which $\nabla_{\theta_t} V_t(s_t)$ is the gradient of $V_t(s_t)$ with respect to θ_t, and α is a step-size. This changes θ_t such that the error of this sample is reduced most. Note that θ_t cannot simply be changed such that the error of this sample is reduced to zero. The approximated function should make an optimal fit through *all* the samples; not just this single one. It is for this reason that the parameters are only altered slightly after each sample.

4.5.2 *Least Squares*

When experience is gathered, the current approximation of the value function is updated. Very frequently it is the case that the target of the updated function does not lie on the plane spanned by the basis functions. A way of solving this problem is by *projecting* it onto this plane. Least squares [19] solutions update the weights of the function approximator by performing an orthogonal projection. An illustration of this principle is shown in Figure 4.7. The projection can be used to find $\widehat{Q}^{\pi}$ directly from a set of samples. It requires the inversion of a matrix with size $N \times N$, with N being the amount of basis functions used to approximate Q multiplied by the amount of actions (as every action has its own set of basis functions).

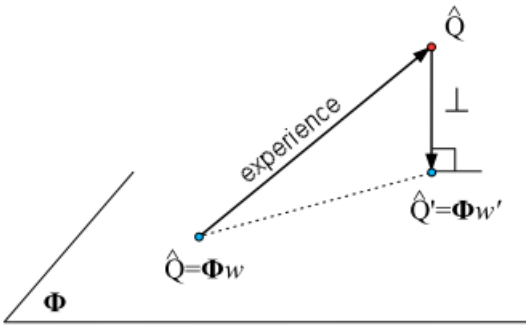


Figure 4.7: Orthogonal least squares projection.
(Adapted from [19])

4.5.3 *Tree-Based Regression*

In Section 4.4.2 tree-based representations were shown. Tree-based regression methods [9] are used to build these trees. They generate a binary test at every node and split up the state-action space into regions. The parameter to split on at each node can be pre-determined (the layers can for example cycle through the state-action dimensions sequentially), but can also be deduced from data inside the node. The latter can be done for example by maximizing *information gain* [8], using which the test is chosen that lowers the entropy of the data to the greatest extent. Choosing the test that minimizes the variance of the data could be a strategy as well.

For RL, several tree-based regression methods exist. They can be categorized as shown in Table 4.1. The two important properties of the categories are:

1. **The number of trees generated**
   The representation of Q can either consist of a single tree or an ensemble of trees. In the case of a tree ensemble, the predicted Q-values will be an average

of the predictions of each individual tree. Building more than one tree is a computationally expensive operation, but does result in more accurate learning.

2. **Dependence of the tree on the value function**
   If the tree structure is dependent on the current value function, the trees need to be re-built every time the values change, for example during iterations of most RL methods. This is a computationally expensive operation, but does result in more accurate learning.

Table 4.1: Categorization of tree-based regression methods.

	Single tree	Multiple trees
Tree(s) independent of value function	Category 1	Category 2
Tree(s) dependent on value function	Category 3	Category 4

4.6 ALGORITHMS SUITABLE FOR GPU PROGRAMMING

Before writing this report, the author performed a literature research to find out which RL algorithms are suitable to be programmed on a GPU [32]. The parallelization potential of several RL algorithms was estimated and an investigation of existing parallel implementations was made.

4.6.1 *Unsuitable Algorithms*

Several RL algorithms were found unsuitable for GPU implementations. Firstly, Temporal Difference learning (TD) [35, Section 6] is one such method. With TD, after the agent takes an action, the value function is updated immediately using the observed reward and the current value of the successor state. It is unsuitable for a GPU implementation since its computational intensity is too low: few calculations have to be performed after each step and the step time is likely much larger than the computational time.

Another unsuitable algorithm is Least-Squares Policy Iteration (LSPI) [19]. It approximates the value function using a linear combination of basis functions, using least-squares supervised learning. Large transition matrices are used in each value function update. Busoniu et al. [4, Section 5.6.2] showed a typical learning example (the two-link manipulator). The experiment used matrices that, on a GPU, would take up more space than cache layers or local memory would be able to hold. A tiling strategy could be used (Section 2.4.2), but tile switching latency would be difficult to hide. Plus, the nature of the algorithm leads to it that increasing the problem size means that the number of computations increases by a power of three. For these two reasons speedups can be expected only with very small problems, and thus LSPI was deemed unsuitable for GPU implementations.

In general, policy search methods (e.g. policy gradient methods [37]) were found unsuitable as well. They typically rely on updating the weights of the policy after complete trials. This means that computations are few and infrequent, which limits the computational intensity and with that the possible speedups. PILCO [7], a policy search method with more computations, was found unsuitable for a similar problem as with LSPI: a tiling strategy is required with switching latency that is hard to hide.

4.6.2 *Suitable Algorithms*

There were also RL algorithms which are suitable for GPU implementations. Dynamic programming (DP) [35, Section 4] uses a model of the environment. This means that transitions can be *computed* without the agent actually experiencing them. The Bellman optimality operator (Equation 4.7) can be used to simply calculate the optimal value function, based on which an optimal policy can be determined. Using the model, all states can update their value *simultaneously* by simulating actions. This is an embarrassingly parallel process and thus very suitable for GPU implementations.

Horde [36] is suitable for GPU implementations too, as it was specifically designed for multi-core processors. With Horde, one agents interacts with the environment. Inside the learning system multiple sub-agents called *demons* learn their own task using the information. So, multiple tasks are learned at the same time. Each GPU core could process its own demon.

Dyna [35, Section 9.2] is another method that is suitable for multi-core implementations. Besides updating the value function using samples, it also tries to learn a model of the environment based on those samples. This way, the value function can be updated by gathering new samples *and* by generating simulated samples (i.e. evaluating the model) at the same time. The CPU could be used to gather and store real samples. The GPU cores could then be divided into three parts: one part that updates the model, a second part that generates simulated samples, and a third part that updates the value function using those real and simulated samples.

Fitted Q Iteration (FQI) [9] is another suitable method. It approximates the value function by iteratively applying a supervised learning method to a set of samples, *after* the agent interacts with the environment. In the original paper tree models and tree-based regression were used. First, a *training set* is generated from the samples, based on the current approximation of the value function. After that, the approximation is updated using the training set. Many pieces of the algorithm can be parallelized. All GPU cores could be used to convert samples to training samples and multiple trees could be generated in parallel. Because of this, this method is suitable to be programmed for a GPU.

While the original FQI algorithm used tree models, neural networks can also be used to represent the value function. This is done in Neural Fitted Q Iteration (NFQ) [33]. Instead of distributing nodes of tree models over the cores, the nodes of the neural network could be distributed. The GPU speedup potential is preserved.

4.6.3 *Existing Parallel Implementations*

There have been several attempts at parallelizing RL algorithms, most of which are on multi-core CPU systems.

Multi-Core CPU

The great majority of the research done for multi-core CPU implementations has been with model-based methods. They assume a complete and accurate model of the environment is available. For example, Wingate et al. [40] parallelized dynamic programming. They distributed regions of the state space over the cores. Their

speedup was about an order of magnitude with respect to a sequential implementation.

Tan et al. [38] also parallelized dynamic programming. Unique about their approach was the fact that two helper threads were used for memory transfers: one for transfers between the CPU's cache and DRAM and vice versa. Their speedup was about two orders of magnitude.

Instead of performing computations faster, processor cores can also be used to learn multiple tasks at the same time, as was shown with Horde. Learning can then become even faster when multiple learning agents share information. Tan [39] researched this by letting agents share samples and policies. Kretchmar [18] builds upon that by implementing a multi-agent system which shares Q-values between agents. He shows that important parameters in multi-agent systems are the number of agents, what type of information is shared, and the frequency and timing of sharing the information. A problem with this approach was that there was wasted experience: the agents might learn the same samples. To avoid this problem the communication frequency could be increased, but then memory latency would become the speedup limiting factor. Trying to find a balance between these two, Kretchmar ultimately found that with ten agents, an order of magnitude fewer trials were needed to learn a task.

Research in parallelizing model-free or model-learning methods is much more scarce. Caarls et al. [5] evaluated Dyna on multi-core CPUs for robotics applications. One thread was used to obtain real-world experience, another to update the learned model, and a set of P threads to generate simulated experiences. They reported a speedup of two orders of magnitude using 16 shared-memory threads.

Li et al. [20] used MapReduce [6] to investigate possibilities for parallel implementations of TD and LSPI. Their parallelizations are mostly done on the level of matrix and vector operations (e.g. finding a maximum, or multiplication). The authors did not implement their suggested parallelizations, so the possible speedups remain unknown.

GPU

Very few attempts have been made to actually implement RL algorithms on GPUs. An example attempt is made by Gehring [15] who published code with a Horde implementation in OpenCL. He used a thread for every demon and states that at least 1000 demons are required for the algorithm to achieve speedups over CPU implementations.

Oh et al. [28] showed a GPU implementation of a neural network. They demonstrated that each layer of a network can be described by a matrix multiplication, which can be computed efficiently using a GPU by distributing its elements over the cores.

Furthermore, Palmer [30] implemented LSPI on a GPU. He used the Brook GPU programming framework [3] to compute columns of the matrices in parallel. As mentioned before, with LSPI, speedups can only be expected with small problems. Doing so, with a small sample set of 2000 samples, he obtained a speedup of approximately 3.5 times with respect to a CPU implementation.

4.6.4 *Choice for Fitted Q Iteration*

Based on the literature research it was decided to implement Fitted Q Iteration on the NVIDIA C2075 GPU, since FQI's parallelization potential is high and no previous attempts have been found in literature. The next chapter shows the exact workings of the algorithm and where its parallelization potential lies more specifically.

4.7 SUMMARY

The main theory behind the reinforcement learning problem has now been described. With RL, a learning *agent* interacts with an environment. The environment gives the agent a *reward* (which can be positive, as well as negative or zero) for every action the agent takes. The agent tries to maximize the *return* (the maximum cumulative reward) over its course of actions. It bases its behavioral *policy* on the *value function*, which stores for each state the maximum possible return. Some RL methods try to optimize the value function (value based methods), whereas others try to find the optimal policy directly (policy search methods).

The way in which the value function, reward function or policy are stored, is determined by their *representation*. This can in the simplest case be a lookup table. However, with big state-action spaces, the agent is unlikely to reach every state-action pair during the learning process. In these cases function approximation representations are therefore used, like tile coding, regression trees or neural networks.

A *supervised learning method* determines how experience is mapped onto the representation. Gradient descent, least squares methods and tree-based regression are often used to this end.

In a literature research it was found that several RL methods are suitable to be programmed on a GPU. Of these, Fitted Q Iteration was chosen to be implemented because of its parallelization potential and the lack of previous attempts.

5 FITTED Q ITERATION

Fitted Q Iteration (FQI) [9] is the reinforcement learning method that was parallelized and programmed for the GPU. It aims to approximate the value function Q from a set of obtained samples, from which a policy is derived. It therefore belongs to the group of *batch reinforcement learning* methods. Fitted Q Iteration iteratively applies a supervised learning method to the set of samples, according to the following two steps:

1. Converting the sample set to a training set, based on the current approximation of Q
2. Using the training set and a supervised learning algorithm to obtain an improved approximation of Q

These steps are repeated until certain stopping conditions are met.

This chapter starts with an explanation of the FQI algorithm in the first section. The second section then describes tree-based FQI (a more specific implementation of FQI), followed by its parallelization possibilities in the third section, from which the tree-based procedures to parallelize are chosen. The last two sections explain these procedures.

5.1 ALGORITHM OUTLINE

In pseudocode, the Fitted Q Iteration algorithm looks as follows:

Algorithm 5.1 Fitted Q Iteration

```
sample_set ← Interact_With_Environment()
Q̂ ← 0
return ← -∞

while stopping conditions not met do
    training_set ← Generate_Training_Set(sample_set, Q̂)
    Q̂ ← Supervised_Learning(training_set)
    return ← Test_Policy(Q̂)
end while
```

An initial set of samples is generated from interaction with the environment. These samples have format (s, a, r, s'), in which the state s and action a can have multiple dimensions. Then, during each iteration of the algorithm, the samples are converted to *traintuples*, which together form the training set $\mathcal{TS}$.

Each traintuple consists of an input i and output o. The conversion (as described in Algorithm 5.2) is done based on the current approximation of Q. The input simply consists of the sample's state and action:

$$i = (s, a) \quad , \tag{5.1}$$

Algorithm 5.2 Generating a training set based on the current approximation of Q

```
1: function GENERATE_TRAINING_SET(sample_set, Q̂)
2:     training_set ← empty array
3:     for sample ∈ sample_set do
4:         traintuple t
5:         t.input ← (sample.s, sample.a)
6:         t.output ← sample.r + γ max_{a∈A} Q̂(sample.s', a)
7:         training_set.add(t)
8:     end for
9:     return training_set
10: end function
```

whereas the output is found by applying an implementation of the Bellman optimality operator (Equation 4.7) to the current approximation of Q in the sample's successor state:

$$o = r + \gamma \max_{a \in \mathcal{A}} \widehat{Q}(s', a) \quad . \tag{5.2}$$

This means: in the successor state s' of the sample the best action $a \in \mathcal{A}$ to take is determined, and its discounted value is added to the sample's reward.

When the training set has been generated, a supervised learning method is used to update $\widehat{Q}$. With the new approximation the return of the policy can be evaluated. The algorithm stops if stopping conditions are met. Examples of possible stopping conditions are:

- A fixed number of iterations has been performed
- The maximum change of Q-values is less than a certain threshold
- The optimal return has been found

5.2 TREE-BASED FITTED Q ITERATION

In principle, FQI can be used with any representation of Q and any supervised learning method. During experiments with the algorithm, Ernst et al. chose to use tree models and tree-based supervised learning as described in Section 4.4.2 and 4.5.3, respectively. They argue that tree-based methods achieve good accuracy and are robust, computationally efficient and scale well to problems with many dimensions [9, Section 1].

They describe and experiment with several tree-based supervised learning methods of every category shown in Table 4.1. The cut-parameters used in the binary tests are the input parameters of the traintuples (i.e. one of the state- or action dimensions of the samples). This way the trees partition the training set $\mathcal{TS}$ into regions of state-action space. Nodes are split until the number of traintuples in a node drops below a constant n_{min}, or other leaf-conditions are met, which are dependent on the tree-building method. In these cases the node is a leaf of the tree. Each leaf stores the average output of the traintuples it contains, which represents the value of that region of state-action space.

5.2.1 *Determining Values of State-Action Pairs*

The tree model represents the state-action value function Q. Thus, it can be used to determine the value of state-action pairs (*queries*). Suppose a learning agent is currently at state s. The value of taking action a in that state can be determined by passing the (s, a)-pair through the tree. The query starts at the root node of the tree and keeps following the tests at every node it encounters, until it reaches a leaf. The value of the query is the average output of the traintuples of that leaf. This process is shown in Algorithm 5.3 and 5.4.

Algorithm 5.3 Determining the value of a query (state-action pair)

```
1: function DETERMINE_STATE_ACTION_VALUE(sample, tree)
2:     query ← (sample.s, sample.a)
3:     value ← Pass_Through(tree, query)
4:     return value
5: end function
```

Algorithm 5.4 Passing a query through the tree to find its value

```
1: function PASS_THROUGH(tree, query)
2:     node ← tree.root_node
3:     while node is not leaf do
4:         if query.value_of_input_param(node.cut-param) < node.cut-point then
5:             node ← node.left_child
6:         else
7:             node ← node.right_child
8:         end if
9:     end while
10:
11:    value ← node.average_traintuple_output
12:    return value
13: end function
```

5.2.2 *Fundamental Procedures*

At the start of this chapter the two main procedures of FQI were given:

1. Generating a training set using the current approximation of Q
2. Using the training set and a supervised learning method to improve the approximation of Q

In the case of tree-based FQI, training set generation mainly consists of passing each sample $\mathcal{A}$ times through the current tree(s) in order to calculate $\max_{a \in \mathcal{A}} \widehat{Q}(s', a)$ in Equation 5.2. This is referred to as the *pass-through* in the remainder of this report.

Since tree models are used to represent Q in tree-based FQI, improving the approximation of Q means building the tree(s). The remainder of the report refers to this as *tree-building*. The pass-through and tree-building tasks are fundamental parts of any implementation of tree-based FQI.

5.3 PARALLEL TREE-BASED FITTED Q ITERATION

From the steps above it can be seen that, in order to parallelize tree-based FQI, tree-building and/or passing samples through the trees should be parallelized. However, these steps have different potential for achieving speedups: the pass-through is easier to parallelize than building trees. To achieve maximum speedups this should be taken into account, as it influences the choice of tree-building method to use. Let's have a look at the parallelization potential of both steps, from which the right tree-building method can be chosen.

5.3.1 *Parallelization Potential*

Passing samples through the trees

Every processing element can process its own sample. Caching can be exploited by letting the PEs access the same tree at the same time. Only one kernel call needs to be scheduled, since the samples are completely independent and thus do not depend on each other's outcome.

The time complexity of this process can be approximated by:

$$O = \text{scheduling\_overhead} + n_{trees} \frac{n_{samples} n_{actions} n_{layers}}{n_{PEs}} \quad . \tag{5.3}$$

As mentioned, there is only a single scheduling overhead for the entire process. Every sample needs to be passed through all the n_{trees} trees $n_{actions}$ times, since the best action to take in the sample's successor state needs to be determined. Inside each tree, the sample needs to pass at most n_{layers} nodes.

Great speedups can be expected as $n_{PEs} \to \infty$ and the scheduling overhead is much less than the kernel time.

Building trees

During the tree-building task, one or more trees are created. A tree starts with a root node, which is split until only leaves remain. During each node split, the cut-parameter and cut-point of the split is determined, and for every sample in the node it is determined whether it belongs to the left or right child node. This is a computationally intensive part of most tree-based FQI algorithms, as multiple samples need to be processed at every node.

There are various tree-building algorithms, which can be parallelized in multiple ways. Roughly, the following approaches can be distinguished:

1. **Node-by-node splitting, one node using all PEs (Figure 5.1)**
   In this case the entire GPU works on splitting one single node. Since many calculations are the same for every traintuple, the traintuples can be divided over the PEs. The GPU can this way process hundreds of traintuples at the same time.

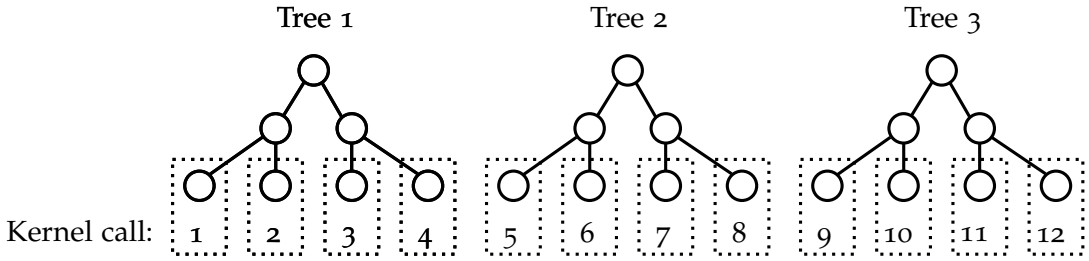
   However, scheduling one or more kernel calls for every node has an overhead (e.g. to read and write buffers or enqueue kernels). In practice it means that a node needs to have thousands of traintuples to justify the overhead, let alone making the split faster than a split on the CPU. In typically-sized problems,

only nodes in the first couple of layers have this amount of traintuples, meaning that the great majority of nodes are split slower than the CPU.

The time complexity of this approach can be approximated by:

$$O = n_{nodes}\left(\text{scheduling\_overhead} + \frac{n_{traintuples/node}}{n_{PEs}}\right) \quad . \tag{5.4}$$

Every node has a scheduling overhead. The traintuples of the node are distributed over the PEs. If $n_{PEs} \to \infty$, then an overhead of $n_{nodes} \cdot \text{scheduling\_overhead}$ remains.



Division of work on GPU during each kernel call:

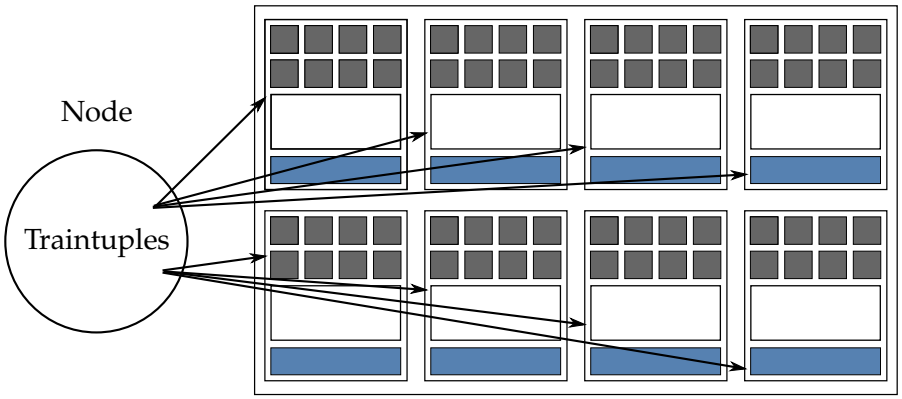


Figure 5.1: Tree-building approach 1: node-by-node splitting. The traintuples inside the node are distributed over the PEs of all CUs of the GPU. Only three trees are shown for illustration purposes.

2. **Layer-by-layer splitting of nodes of a single tree, one node per CU (Figure 5.2)**

   In this case multiple nodes are split at the same time, in a layer-by-layer fashion. Every node is split on one CU, and it divides its traintuples over the PEs of that CU. This way, tens of nodes can be split at the same time. This approach is used for example by Jansson et al. [17] in their attempt to parallelize tree-ensemble algorithms using CUDA.

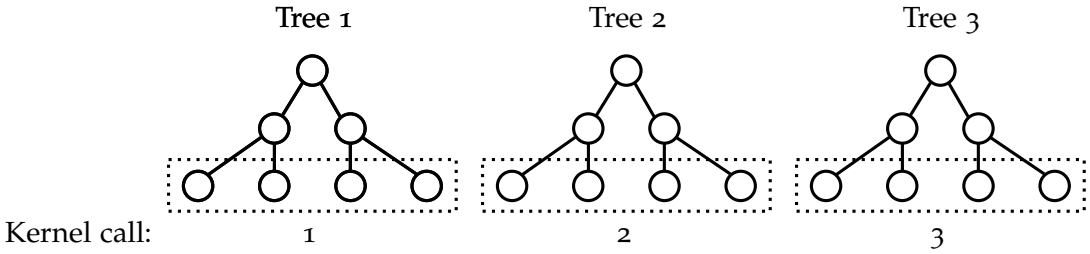
   This approach means: the more nodes, the better. Contrary to the previous approach, this approach therefore has greater parallelism in later layers of the algorithm, where there are more nodes. However, every layer does come with an overhead. This is because of the fact that, for a layer to be split, the results of the previous layer need to be known.

The time complexity of this approach can be approximated by:

$$O = n_{trees} n_{layers} \Big(\begin{array}{l} \text{scheduling\_overhead} \\ + \text{cleanup\_overhead} \\ + \dfrac{n_{traintuples}}{\min(n_{CUs}, n_{nodes}) n_{PEs/CU}} \end{array} \Big) \quad . \tag{5.5}$$

A total of n_{trees} trees need to be built, of which every layer has a scheduling and cleanup overhead. The cleanup removes leaves from the trees, to limit global memory usage. Furthermore, it can be assumed that in each layer an amount of samples in the order of $n_{samples}$ is going to be split, which in early layers are contained in only a couple of nodes, while in later layers they will be distributed over many nodes. Lastly, the $\min(n_{CUs}, n_{nodes}) n_{PEs/CU}$ denominator can be explained as follows. Since one CU splits one node, a maximum of n_{CUs} nodes can be split simultaneously. If n_{nodes} in a layer is less than n_{CUs}, then n_{nodes} CUs are used.

If $n_{PEs/CU} \to \infty$, then an overhead of $n_{trees} n_{layers} \cdot (\text{scheduling\_overhead} + \text{cleanup\_overhead})$ remains.



Division of work on GPU during each kernel call:

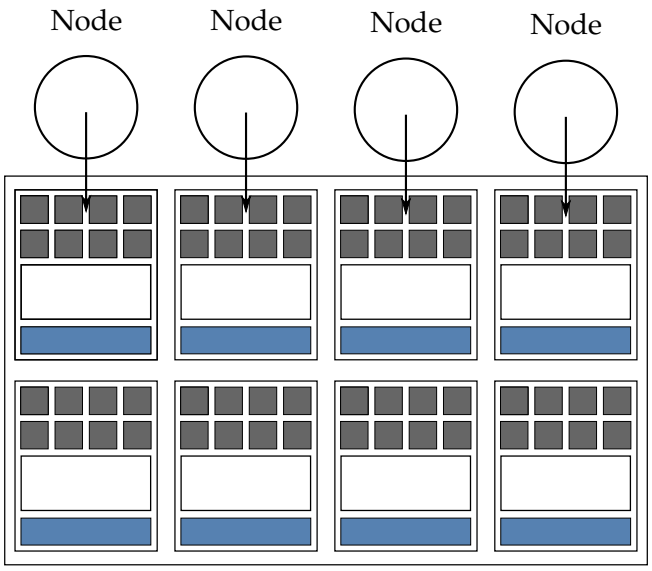


Figure 5.2: Tree-building approach 2: layer-by-layer splitting, one tree at a time. One node is split by one CU.

3. **Layer-by-layer splitting of nodes of multiple trees, one node per CU (Figure 5.3)**
   The previous option does not need any information about the actual trees, it

just takes nodes with sets of traintuples and performs the splits. It can therefore easily be extended to split multiple nodes of multiple trees at the same time, as Jansson et al. [17] also noted. The advantage is that more splits can be scheduled at every kernel call, reducing the overall overhead time. To be more precise, in the time complexity as shown in Equation 5.5, the n_{trees} factor moves from the front to the fraction within the parentheses, giving:

$$\begin{aligned} O = n_{layers}\big(& \text{scheduling\_overhead} \\ & + \text{cleanup\_overhead} \\ & + \frac{n_{trees} n_{traintuples}}{\min(n_{CUs}, n_{nodes}) n_{PEs/CU}} \big) \quad . \end{aligned} \tag{5.6}$$

Here, if $n_{PEs/CU} \to \infty$, then an overhead of $n_{layers} \cdot (\text{scheduling\_overhead} + \text{cleanup\_overhead})$ remains.

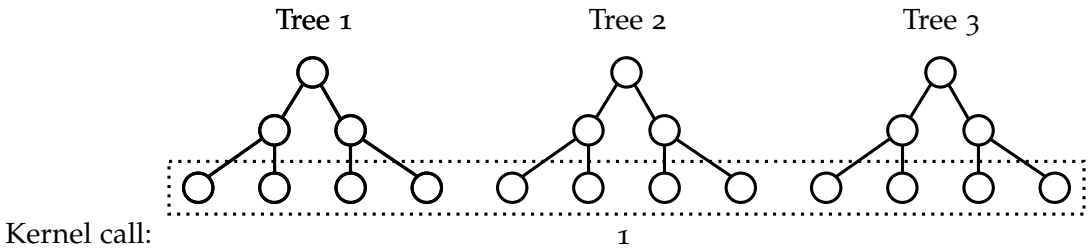
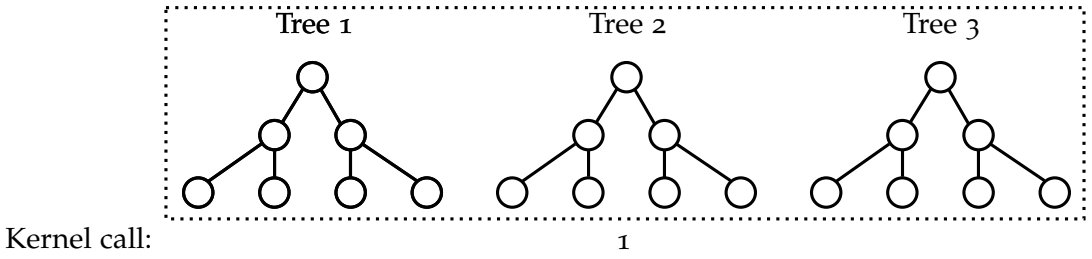


Figure 5.3: Tree-building approach 3: layer-by-layer splitting, multiple trees simultaneously. As with approach 2, one node is split by one CU, so the division of work on the GPU during each kernel call is the same as with approach 2 (Figure 5.2).

4. **Splitting multiple trees at the same time, one tree per PE (Figure 5.4)**
   This approach is used in the CudaRF algorithm written by Grahn et al. [11] to build random forests using CUDA. They note that "*their implementation works best for a large number of trees*", which is exactly where the problem lies for tree-based FQI. According to Ernst et al., tree-building methods that build multiple trees already had good learning performance from 20 to 50 trees, and did not improve much when using a greater number of trees. Using only 50 PEs of the hundreds available would not be an efficient way of programming tree-based FQI, and is therefore considered unsuitable as a parallelization strategy.



Division of work on GPU during each kernel call:

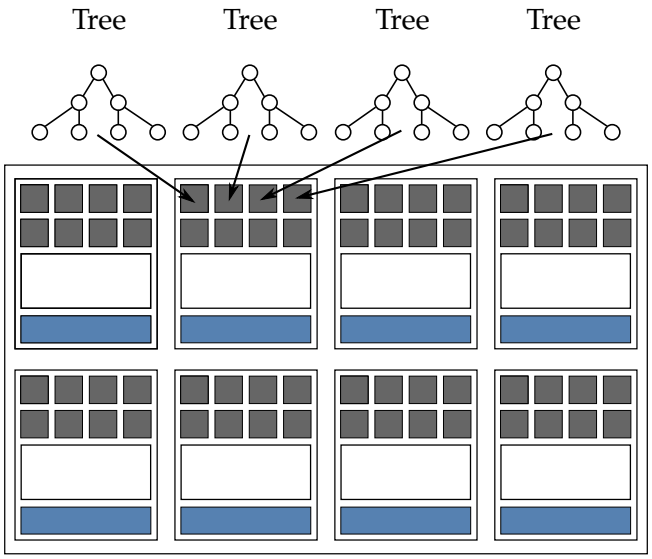


Figure 5.4: Tree-building approach 4: building multiple trees at the same time. One tree is built by one PE.

Since with approach 1 $n_{traintuples/node}$ is likely to be smaller than n_{PEs} for the great majority of nodes, approach 2 and 3 are the better alternatives. Splitting multiple trees at the same time reduces overall overheads by a factor n_{trees}, making 3 the best parallelization strategy.

Conclusion

All in all it can be concluded that passing samples through the trees has a greater parallelization potential than tree-building. With the pass-through there is only one single scheduling overhead, after which all the PEs of the device are used. With tree-building there is an overhead at every layer, which is greater than the overhead of the pass-through. Furthermore, in early layers, only a part of the PEs is used, decreasing the potential compared to the pass-through.

5.3.2 *Choice of Tree-Building Method*

Based on the above, it was chosen to parallelize the *KD-Trees* [9, Section 4.2.1] and *Extra-Trees* [9, Section 4.2.4] tree-building methods, whose algorithms are explained in the following two sections. This choice can be explained as follows.

Equation 5.2 shows that, when generating the training set, the traintuple output is calculated based on the current approximation of Q. Since the approximation

changes at every iteration of FQI, the output of traintuples are different every iteration as well. Therefore, if the tree structure is dependent on traintuple output, the trees need to be re-built every iteration.

Since tree generation is more difficult to speed up than the pass-through, for maximum speedups, it is best to use trees that do not change during iterations of FQI. That is, they should not depend on traintuple output (Category 1 and 2 in Table 4.1). This way the trees only need to be generated once; the structure of the tree (i.e. the test at each node) remains the same during the iterations. The pass-through speedup is in this case not limited by the lower speedup of the building of the trees. *KD-Trees* is a good tree method of Category 1.

However, as mentioned in Section 4.5.3, tree methods that *do* depend on traintuple output have more accurate learning results. Because of this, also the tree method *Extra-Trees* of Category 4 was chosen to parallelize, which had the most accurate results in the experiments performed by Ernst et al. in their paper.

5.4 KD-TREES

The node-splitting algorithm of KD-Trees is shown in Algorithm 5.5. Nodes are split recursively, using alternating cut-parameters: the first node uses parameter 1, its child nodes use parameter 2, after which their child nodes use parameter 3, etcetera. The cut-point is the median of the traintuple input values for the cut-parameter. This means that the child nodes each receive an equal amount of traintuples. A node is considered a leaf when the number of traintuples in the node is less than a constant n_{min}.

Algorithm 5.5 KD-Trees node-splitting algorithm

```
function KD_Trees_Split_Node(TS_node, param)
    cutpoint ← Find_Param_Median(param, TS_node)
    (TS_L, TS_R) ← Perform_Test(TS_node, cutpoint)
    return (cutpoint, TS_L, TS_R)
end function
```

Since the tree structure does not depend on the traintuple output, the structure does not change during the iterations of FQI and therefore only needs to be generated once. During the iterations, only the average traintuple output inside the leaves of the tree changes.

5.5 EXTRA-TREES

The node-splitting algorithm of Extra-Trees is shown in Algorithm 5.6. At every node, Extra-Trees performs a test on every input parameter, using a random cut-point for each test. A score for each test is calculated, and the test with the highest score is saved. The test score is calculated according to:

$$\text{Score} = \frac{var(o|\mathcal{TS}) - \frac{|\mathcal{TS}_L|}{|\mathcal{TS}|}var(o|\mathcal{TS}_L) - \frac{|\mathcal{TS}_R|}{|\mathcal{TS}|}var(o|\mathcal{TS}_R)}{var(o|\mathcal{TS})} \quad , \tag{5.7}$$

in which $var(o|\mathcal{TS})$ is the variance of the output of the traintuples in $\mathcal{TS}$, and $|\mathcal{TS}|$ the amount of traintuples in $\mathcal{TS}$. This equation shows that the test with the highest score is the one that has the smallest traintuple output variance in its child nodes. Since the test score and thus the split depends on the traintuple output, the structure of the trees change during every iteration of FQI. A node is considered a leaf (and thus is not split further) if one or more of the following conditions are met:

1. The number of traintuples in the node is less than a constant n_{min}
2. The inputs of the traintuples in the node are equal
3. The outputs of the traintuples in the node are equal

Algorithm 5.6 Extra-Trees node-splitting algorithm

```
1: function Extra_Trees_Split_Node(TS_node)
2:     max_score ← −∞
3:     best_param ← −∞

4:     for param ∈ input_params do
5:         (min, max) ← Find_Param_Minmax(param, TS_node)
6:         cutpoint[param] ← Random_In_Between(min, max)

7:         (TS_L[param], TS_R[param]) ← Perform_Test(TS_node, cutpoint[param])
8:         score[param] ← Calculate_Test_Score(TS_L[param], TS_R[param])

9:         if score[param] > max_score then
10:            max_score ← score[param]
11:            best_param ← param
12:        end if
13:    end for

14:    return (best_param, cutpoint[best_param],
           TS_L[best_param], TS_R[best_param])
15: end function
```

5.6 SUMMARY

Fitted Q Iteration is an RL method which iteratively applies a supervised learning method to a set of samples, to obtain an approximation for the state-action value function Q. It therefore consists of two steps:

1. Building a training set using the sample set and the current approximation of Q
2. Performing supervised learning to improve the approximation of Q

Tree-based FQI uses a tree model and tree-based supervised learning to represent Q and build the trees, respectively. The fundamental process for step 1 in tree-based FQI is a *pass-through* of all samples through the trees, whereas step 2 is now

a *tree-building* task. The pass-through has greater parallelization potential than tree-building. This means that, in order to maximize speedups, as few trees as possible should be built, in order to not let the lower tree-building speedup limit the pass-through speedup. Because of this, *KD-Trees* was chosen to parallelize, as it only uses one tree which only needs to be generated once. However, tree methods that change the trees during iterations of FQI (and thus have to re-build the trees every time) result in greater learning accuracy. Therefore, *Extra-Trees* was also chosen to parallelize.

6

EXPERIMENTAL SETUP

This chapter explains the setup of the experiments that were performed. In the first two sections, the sequential and parallel implementations of KD-Trees and Extra-Trees are explained. The third section then shows the system on which the experiments were run. The chapter is finished with a description of the Puddle World task that was used to benchmark the implementations.

6.1 KD-TREES IMPLEMENTATION

In this section the sequential and parallel implementations of KD-Trees that were made, are described. This is done for the two most important processes in tree-based FQI: training set generation (in which the pass-through is performed) and tree-building. With KD-Trees the tree structure does not change between iterations. However, the average traintuple outputs of the leaves do need to be updated. The sequential and parallel implementations of this process are described as well.

6.1.1 *Training Set Generation*

As explained in Chapter 5, the first step of FQI is to convert the sample set to a training set. Recall from Equation 5.1 and 5.2 how a traintuple is built from a sample (s, a, r, s'):

$$i = (s, a) \tag{6.1}$$

$$o = r + \gamma \max_{a \in \mathcal{A}} \widehat{Q}(s', a) \quad . \tag{6.2}$$

So, in tree-based FQI, this mostly consists of passing the samples through the trees ($\widehat{Q}$), in order to find $\max_{a \in \mathcal{A}} \widehat{Q}(s', a)$: the value of taking the best action in the sample's successor state. This section explains the sequential and parallel implementations of training set generation in KD-Trees, which were used during the benchmarks.

Sequential

The sequential training set generation algorithm that was used is shown in Algorithm 6.1. Every sample is converted to a traintuple sequentially. In order to determine the best action to take in the sample's successor state, every possible action needs to be evaluated. To this end, a query (i.e. a state-action pair) is created with state $sample.s'$. Every action is set in turn in the query, after which it is passed through the tree to calculate the value for this action. The `Pass_Through` function used is equal to the one shown in Algorithm 5.4 of the previous chapter. This way, a possible value for every action is found, of which the highest is used to calculate the traintuple's output. The traintuple is then stored in the training set with this output and inputs $(sample.s, sample.a)$.

Algorithm 6.1 KD-Trees: sequential training set generation.

```
1: for every sample do
2:     for a ∈ 𝒜 do
3:         query ← (sample.s', a)
4:         possible_values[a] ← Pass_Through(tree, query)
5:     end for
6:     max_value ← Max(possible_values)
7:     traintuple ← i: (sample.s, sample.a), o: sample.r + γ max_value
8:     Add_To_Training_Set(traintuple)
9: end for
```

Parallel

Training set generation is embarrassingly parallel and thus trivial to parallelize. As shown in Section 5.3.1, the samples can simply be streamed from global memory to be distributed over the processing elements of the device, and can be processed independently. A schematic overview of the implementation that was made is shown in Figure 6.1. This implementation is used with Extra-Trees as well, the only difference being the amount of trees used in the pass-through. The work done by a single work-item is shown. The algorithm consists of the following steps:

1. **Initialize a query from a sample**
   Every work-item processes one sample. So, at the beginning of the pass-through, the work-item loads its sample from global memory. The query is initialized based on this sample.

2. **Pass through the tree with all possible actions**
   This is the most substantial part of the algorithm. The same algorithm as shown in Algorithm 6.1 was used. Of course, instead of passing through a single sample at a time, the parallel implementation has many work-items passing samples through the trees simultaneously. Nodes of the trees are read from global memory with random-access. But, since many work-items use the nodes of the tree at the same time, the nodes are cached efficiently. This especially holds true for nodes near the root of the trees: they are used by most of the samples.

3. **Determine the best action**
   The best action to take in the sample's successor state corresponds to the highest value found in the previous step.

4. **Calculate traintuple output**
   The actual traintuple is created and written to the training set in global memory.

6.1.2 *Tree-Building*

With KD-Trees the tree only needs to be built once.

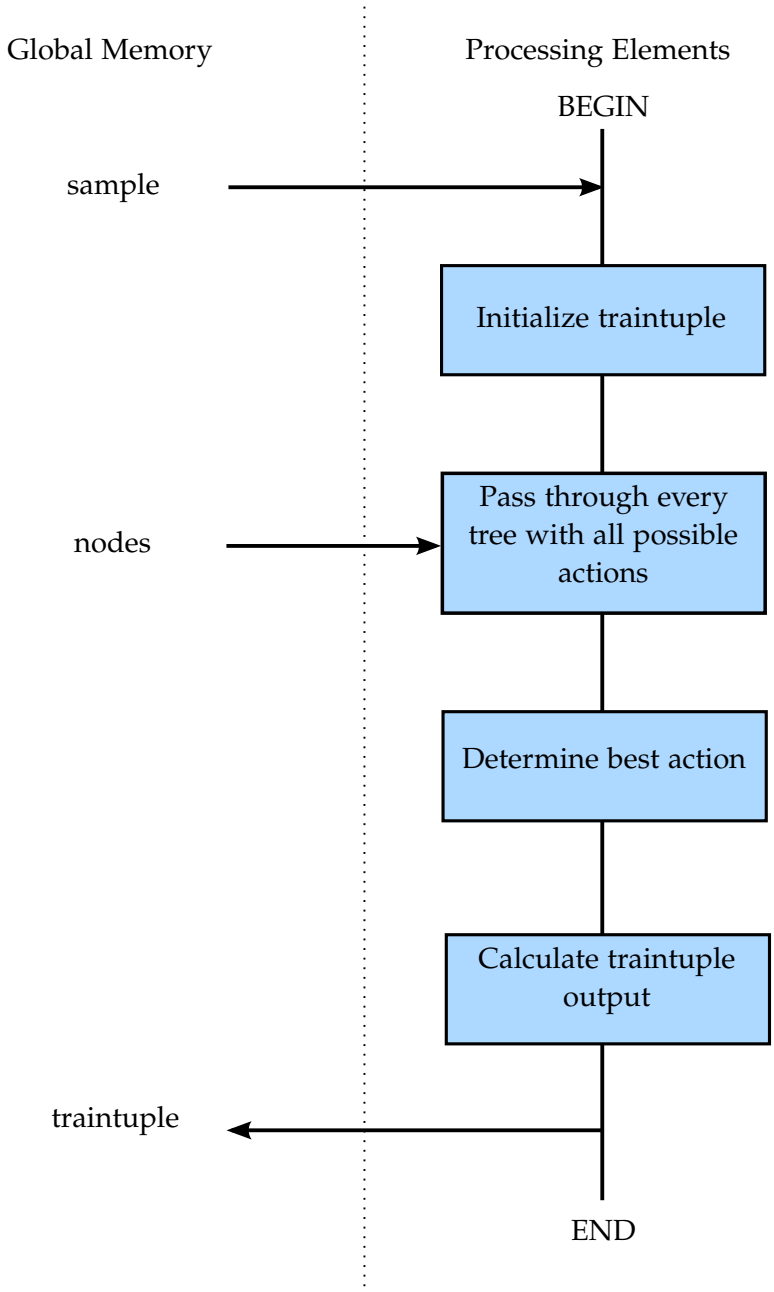


Figure 6.1: Schematic overview of the parallel implementation of training set generation as used with both KD-Trees and Extra-Trees. The work done by every work-item in a work-group is shown.

Sequential

The sequential implementation of KD-Trees' tree-building algorithm is shown in Algorithm 6.2. It keeps splitting nodes recursively until leaves are found.

Algorithm 6.2 KD-Trees: sequential tree-building.

```
1:  function SPLIT_NODE(node, param, tree)
2:      if node is a leaf then
3:          leaf_idx ← tree.append(node)
4:          for traintuple ∈ node.TS do
5:              traintuple.leaf_idx ← leaf_idx
6:          end for
7:          return
8:      end if
9:
10:     (cutpoint, TS_L, TS_R) ← KD_Trees_Split_Node(node.TS, param)
11:
12:     node.cutparam ← param
13:     node.cutpoint ← cutpoint
14:     tree.append(node)
15:
16:     node_left ← Create_Node(TS_L)
17:     node_right ← Create_Node(TS_R)
18:
19:     Split_Node(node_left, param + 1)
20:     Split_Node(node_right, param + 1)
21: end function
```

When a leaf is found, its traintuples are assigned the ID of the leaf, using which the tree can be updated during the iterations of FQI, as explained in the next section.

Parallel

Since there is only one tree, which has to be generated only once, the KD-Trees tree-building algorithm has not been parallelized. The amount of time this step takes is negligible compared to the time an actual learning task takes.

6.1.3 *Tree Update*

With KD-Trees the structure of the tree is constant, so the tree does not need to be re-built every iteration. However, the average traintuple output of the leaves do need to be updated.

Sequential

The sequential implementation is shown in Algorithm 6.3. First, the average traintuple output of every leaf is reset to zero. After that, every traintuple increments the output of the leaf it belongs to. Lastly, the average output of every leaf is calculated by dividing the sum by the amount of traintuples inside that leaf.

Algorithm 6.3 KD-Trees: tree update.

```
1:  function UPDATE_TREE(TS, leaves)
2:      for every leaf do
3:          leaf.o_avg ← 0
4:      end for
5:
6:      for traintuple ∈ TS do
7:          leaf[traintuple.leaf_idx] += traintuple.o_avg
8:      end for
9:
10:     for every leaf do
11:         leaf.o_avg /= leaf.num_traintuples
12:     end for
13:
14:     return leaves
15: end function
```

Parallel

The parallel implementation of the tree update in KD-Trees was made as follows. Before the training set generation process displayed in Figure 6.1 is started, a small kernel is launched that resets the traintuple output of every leaf (similar to the first loop in Algorithm 6.3). Every work-item resets the output of one leaf.

Then, an addition is made to the training set generation process. After calculating the new traintuple output (the last block in the figure), the output of the corresponding leaf is incremented immediately (the second loop in Algorithm 6.3). Doing this here has the advantage that the traintuple is already loaded from global memory. If the increment was done afterwards in a separate kernel, it had to be loaded again, adding latency.

The last loop in Algorithm 6.3 is implemented by a small kernel after training set generation. In this kernel every work-item performs the averaging operation on a leaf.

With the above parallel implementations of training set generation and updating the tree, the entire FQI iterations of KD-Trees are performed on the GPU.

6.2 EXTRA-TREES IMPLEMENTATION

Just like with KD-Trees, sequential and parallel implementations of Extra-Trees were made. This section discusses these implementations.

6.2.1 *Training Set Generation*

The training set generation implementations are very similar to the ones made for KD-Trees.

Sequential

Sequential training set generation is the same as KD-Trees' implementation (Algorithm 6.1), with one difference: every sample needs to pass through multiple trees instead of just one, after which the values obtained from the different trees are averaged to find the `possible_value` for that action. This means within lines 2 to 5 of the algorithm, a loop over the trees is made, changing these lines to:

```
1: for a ∈ A do
2:     for every tree do
3:         query ← (sample.s', a)
4:         tree_values[tree] ← Pass_Through(tree, query)
5:     end for
6:     possible_values[a] ← Average_Of(tree_values)
7: end for
```

Parallel

The parallel algorithm (as was shown in Figure 6.1) also adds a loop over the trees to the KD-Trees algorithm. However, contrary to the sequential implementation, in the parallel implementation the tree-loop is the outer loop, whereas the action-loop is the inner one.[1] Since the nodes of the trees have to be loaded from global memory, having this order means less memory transactions have to be made, as many work-items inside one work-group are processing the same tree at the same time. Data can thus be cached efficiently, improving the efficiency of the algorithm.

6.2.2 *Tree-Building*

Sequential

The sequential implementation of Extra-Trees uses a recursive node splitting algorithm as shown in Algorithm 6.4. It uses the `Extra_Trees_Split_Node` function given in Algorithm 5.6 of Chapter 5. The main program now only needs to create a root node for every tree, using the entire training set, and call `Split_Node` once per tree. This is shown in Algorithm 6.5.

Before splitting a node, it is checked whether or not it is a leaf. Recall that with Extra-Trees a node is considered a leaf if one of the following is true:

1. The number of traintuples in the node is less than a constant n_{min}
2. The inputs of the traintuples in the node are equal
3. The outputs of the traintuples in the node are equal

If the node is a leaf it is not split and simply appended to the tree. If it is not a leaf, the node is split, after which it is appended and its child nodes are split.

This implementation shows that trees are generated sequentially, and the nodes within each tree are split sequentially as well.

1 The averaging operation is in this case performed outside the loops.

Algorithm 6.4 Sequential tree-building in Extra-Trees: recursive node-splitting.

```
1:  function SPLIT_NODE(node, tree)
2:      if node is a leaf then
3:          tree.append(node)
4:          return
5:      end if
6:
7:      (best_param, cutpoint, TS_L, TS_R) ← Extra_Trees_Split_Node(node.TS)
8:
9:      node.cutparam ← best_param
10:     node.cutpoint ← cutpoint
11:     tree.append(node)
12:
13:     node_left ← Create_Node(TS_L)
14:     node_right ← Create_Node(TS_R)
15:
16:     Split_Node(node_left)
17:     Split_Node(node_right)
18: end function
```

Algorithm 6.5 Sequential tree-building in Extra-Trees: main program.

```
1: for i ∈ num_trees do
2:     root_node ← Create_Node(training_set)
3:     trees[i] ← []
4:     Split_Node(root_node, trees[i])
5: end for
```

Parallel

Section 5.3 explained that splitting multiple nodes of multiple trees (in a layer-by-layer fashion) at the same time is a good approach to parallelizing Extra-Trees. Figure 6.2 shows a schematic overview of the procedure that was made to parallelize Extra-Trees. The work done by every work-item in a work-group is shown. The procedure consists of the following steps:

1. **Calculate cut-points**
   Since one work-group splits one node, every work-item in the work-group starts by loading the same node from global memory. Recall that with Extra-Trees, a split is performed on every input parameter of the traintuples inside the node. Using the minmax values of the input parameters and random numbers from global memory, the cut-points for the tests are calculated. They are the same for every work-item in the work-group.

   Note that since OpenCL does not come with built-in functionality to generate random numbers, the easiest way to use random numbers on the device is to generate them on the host and read them from memory on the device.

2. **Simulate tests on all input parameters**
   The node's traintuples are distributed evenly over the work-items of the work-group as depicted in Figure 6.3. With bigger problems the traintuples of a

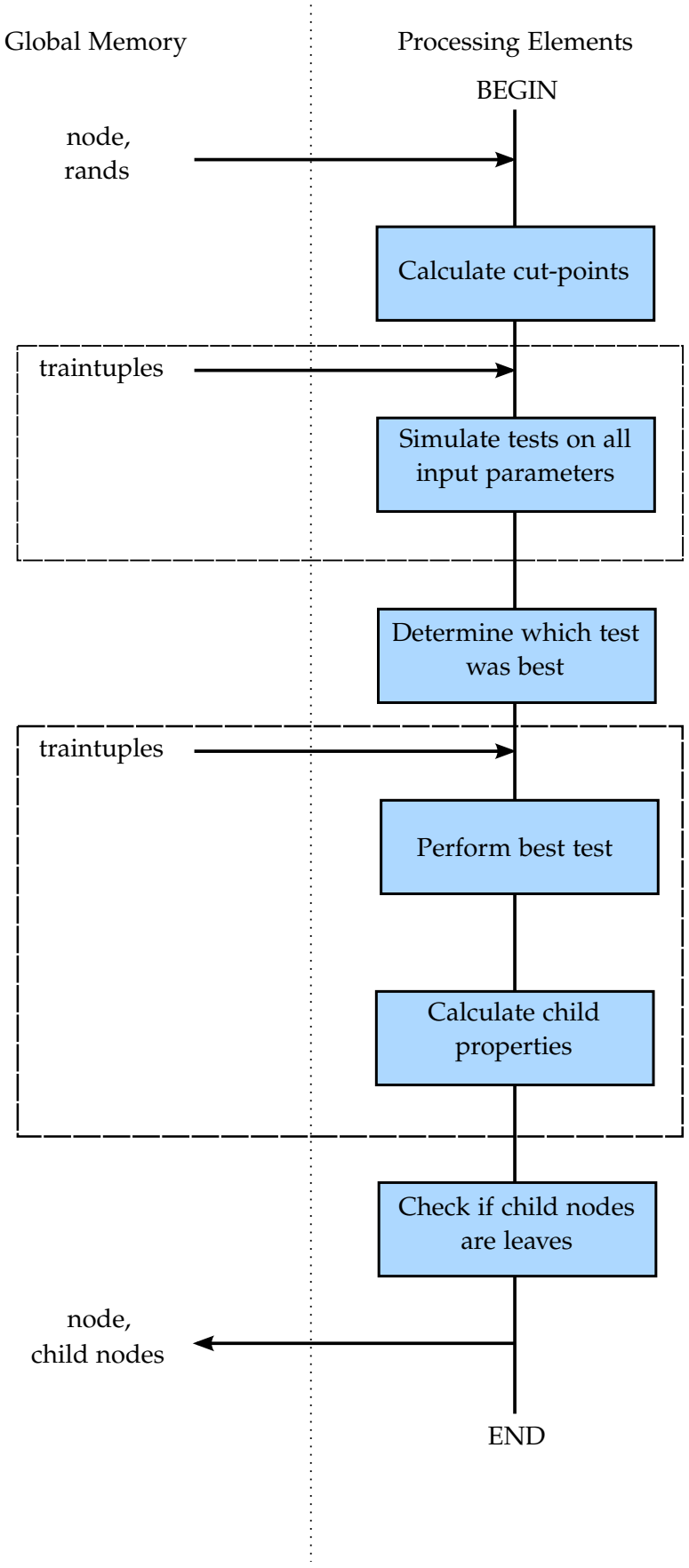


Figure 6.2: Schematic overview of the OpenCL implementation of Extra-Trees. The work done by every work-item in a work-group is shown.

work-item do not fit in private memory, which is why every work-item loads its own traintuples from *global* memory. However, to limit the amount and size of global memory transfers (especially in Step 4), the *indexes* of the traintuples used by the work-item are stored in private memory – a 4 byte integer is only half the size of an 8 byte traintuple.

The global memory layout of the training set and traintuple indexes is as follows. The training set is not reordered during the algorithm. Every node (i.e. work-group) knows at which index its chunk of traintuple indexes starts. For example, consider the following situation in which two nodes will be split:

$$\begin{aligned} \text{traintuple\_indexes} &= [1,2,3,4,5,6,7] \\ \text{nodes[0].traintuple\_indexes\_start\_at} &= 0 \\ \text{nodes[0].num\_traintuples} &= 3 \\ \text{nodes[1].traintuple\_indexes\_start\_at} &= 3 \\ \text{nodes[1].num\_traintuples} &= 4 \end{aligned}$$

This shows that the first node can find the indexes of its 3 traintuples from index 0 of the `traintuple_indexes` array. After loading the indexes the work-group knows that traintuples 1, 2 and 3 are the traintuples of the node it is splitting. Similarly, the indexes of the second node's 4 traintuples start at index 3 of the `traintuple_indexes` array.

This approach has another advantage in that only one training set needs to be present in global memory of the GPU. Instead of every tree having its own training set in global memory, there is only one training set, while every tree has its own set of traintuple indexes in the `traintuple_indexes` array.

The dashed box in Figure 6.2 depicts until where the traintuples are used in the PEs. With the traintuples a test on every input parameter is performed in the following way:

```
1: for every traintuple in this work-item do
2:     for param ∈ input_params do          ▷ Simulate every test on this traintuple
3:         if tt.input[param] < cutpoints[param] then
4:             Update num_L[param], o_var_L[param] and o_avg_L[param]
5:         else
6:             Update num_R[param], o_var_R[param] and o_avg_R[param]
7:         end if
8:     end for
9: end for
```

As can be seen, the work-item keeps track of its own split results (i.e. the number of traintuples that went left and right, as well as their variance and average output). Once every work-item has collected its own results, they need to be combined to the results of the entire node. This is done using a *reduction*, which is a common method in parallel programming for combining data of separate work-items for the entire work-group. Figure 6.4 schematically shows how the reduction works. Since the amount of elements is halved at every step, the amount of work-items in a work-group should be a power of two. Results

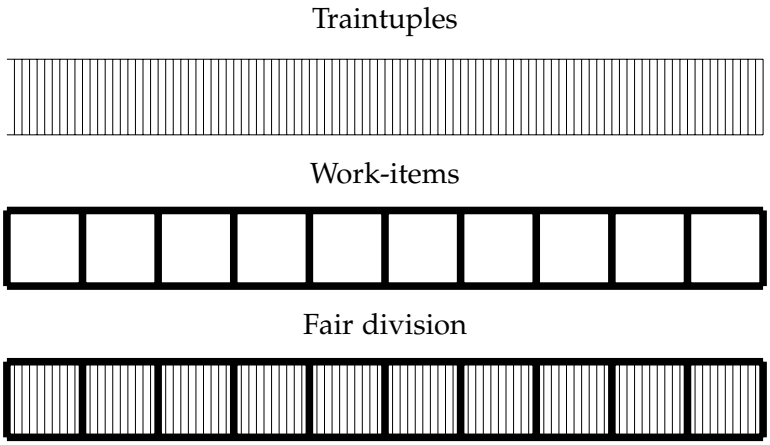


Figure 6.3: Fair division of traintuples over work-items.

of work-item pairs are combined until there is only one result left: the result for the work-group (i.e. the node).

After each reduction, the score of the test is calculated (see Equation 5.7).

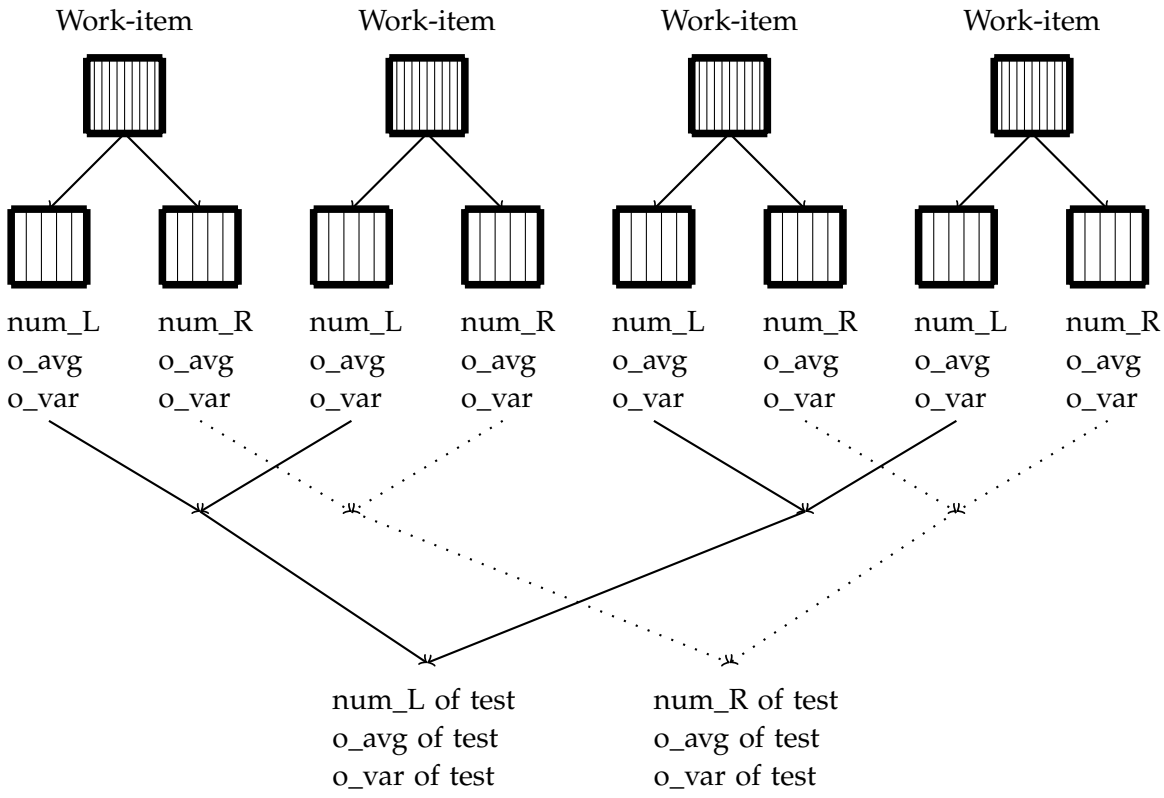


Figure 6.4: Reduction of test results of multiple work-items gives the results for the entire work-group (and thus for the test).

3. **Determine which test was best**
   The test scores of the simulated splits are compared, from which the best parameter to split on is determined.

4. **Perform best test**
   In this step the actual split is performed. During the simulation only the test's

statistics were calculated; the traintuples were not actually split. Here, the traintuple *indexes* are reordered in global memory to reflect the split. Reordering 4 byte traintuple indexes in global memory is more efficient than reordering the actual 8 byte traintuples themselves.

Every work-item again loads its own traintuples from global memory. It might seem inefficient to do this a second time, but it unfortunately is the only option with the hardware used. For typical learning problems keeping the traintuples in private memory is impossible due to the limited space available.

To continue the example shown in Step 2, suppose that during the first node's split traintuples 1 and 3 went left, whereas 2 went right. In the second node, traintuples 5 and 7 went left, while 4 and 6 went right. During the reordering step the `traintuple_indexes` global memory array then changes to:

$$\text{traintuple\_indexes} = [1, 3, 2, 5, 7, 4, 6] \quad ,$$

and the output nodes will be:

$$\begin{aligned}
\text{nodes_out[0].traintuple_indexes_start_at} &= 0 \\
\text{nodes_out[0].num_traintuples} &= 2 \\
\text{nodes_out[1].traintuple_indexes_start_at} &= 2 \\
\text{nodes_out[1].num_traintuples} &= 1 \\
\text{nodes_out[2].traintuple_indexes_start_at} &= 3 \\
\text{nodes_out[2].num_traintuples} &= 2 \\
\text{nodes_out[3].traintuple_indexes_start_at} &= 5 \\
\text{nodes_out[3].num_traintuples} &= 2
\end{aligned}$$

5. **Calculate child properties**
   During the actual split, extra properties of the child nodes are calculated. This can be done efficiently as the traintuples were already loaded from global memory in the previous step. Every work-item checks for its own left and right traintuples:
   a) The minmax of every input parameter, for use when this child node is split in the next layer
   b) If the input or output values are equal

   The per-work-item results are again combined using a reduction. The results for the entire left and right child nodes are then known.

6. **Check if child nodes are leaves**
   If a child node is a leaf, the average traintuple output (as calculated during the simulated tests of Step 2) is stored with it.

7. **Write back results to global memory**
   Finally, the results are written back to memory. The node that was just split is stored along with the cut-parameter and cut-point that were used, and the resulting child nodes are written back.

Once a layer is split, the results are read back from the GPU to the CPU. The CPU then performs a cleanup on the results: leaves that were found are removed from

the node array and the traintuple index array is updated accordingly. The cleaned data is pushed back to the GPU and the kernel is called again, repeating the above process.

6.3 BENCHMARK SYSTEM

The above algorithms were all benchmarked on the system as described in this section. The GPU used to run the parallel code on is an NVIDIA Tesla C2075 [25] workstation GPU, which uses the Fermi GF100 [24] GPU architecture. The specifications are the following:

- Compute units: 14
- Processing elements: 32 per CU (448 in total) running at 1.15 GHz
- Global memory: 6 GB (bandwidth 144 GB/sec)
- Constant memory: 64 kB (not used with the above implementations)
- Local memory: 48 kB per CU

All C++ code was compiled on Linux using g++, combined with NVIDIA's version 5.5.0 CUDA toolkit. At the time of writing the latest OpenCL version supported by NVIDIA is OpenCL 1.1, using which all code was tested.

The CPU used to run the sequential implementations is an Intel Xeon CPU E5-2665 [16] with 20 MB of cache and 8 cores running at 2.40 GHz, of which only one core was used.

For timing of the code and calculating speedups, wall clock time was used. This method of timing was chosen because it is interesting to know how much real-world time would be saved by running the learning problem on a GPU. The wall clock time is determined using `gettimeofday()` from `sys/time.h`.

6.4 THE PUDDLE WORLD TASK

To benchmark the algorithms, it has been chosen to apply them to a variation of the *Puddle World* task as first described by Sutton [34]. The state space is discrete[2], consisting of a two-dimensional square grid in which several 'puddles' have been generated randomly (see Figure 6.5). The goal of the task is for the agent to find an optimal route from four starting states (the top right, bottom left, bottom right and center states) to the goal state (the top left corner). Learning stops when the current policy gives an optimal route to the goal state from every starting state. The reward function is defined as follows:

1. Each step gives the agent a reward of $r = -1$.
2. Entering a puddle region gives the agent a reward of $r = -10$.
3. Reaching the goal state gives the agent a reward of $r = 10$.

2 Note that FQI is by no means limited to discrete state spaces as it uses a function approximator for Q.

The agent is able to move in all four cardinal directions, moving one state at a time. Transitions are assumed to be deterministic, i.e. taking an action takes the agent to the adjacent state in the corresponding direction with a probability of 1. Since the task is discrete and deterministic, the optimal return can be determined beforehand. This return is therefore used as a stopping criterion for FQI.

This task has been chosen for the following two reasons:

1. The task can easily be made more complex by increasing the size of the world
2. Debugging the task is easy because it is deterministic and the optimal return is known beforehand

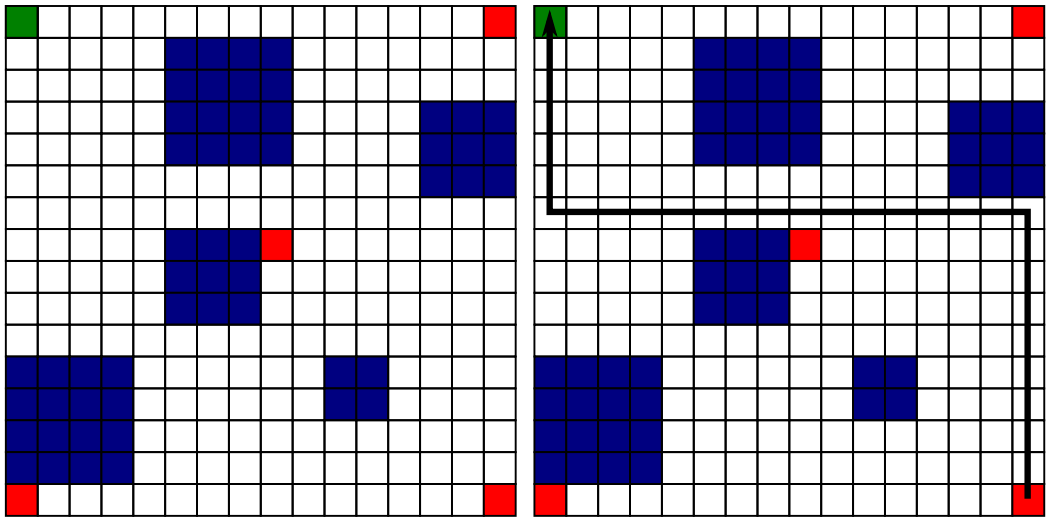


Figure 6.5: Example of a 16x16 Puddle World. The puddles are generated randomly. The learning agent needs to find an optimal route from four starting states (the top right, bottom left and bottom right corners, and the center of the world) to the top left corner. A solution for the bottom right corner is shown on the right hand side.

7 RESULTS

This chapter shows the results of the experiments that were performed. In the first section proof is given that the sequential and parallel implementations of tree-building in Extra-Trees produced equal results. This is important to be able to compare results. For KD-Trees such proof was not necessary, since the tree is generated on the host in both the sequential and parallel implementations. The second section shows the speedups that were achieved in learning the Puddle World task using KD-Trees. It is demonstrated how the speedup changes when varying the amount of samples. Then, in the third section the speedups for Extra-Trees are shown for varying amounts of samples and trees. The last section contains an elaboration on the speedups *within* the Extra-Trees tree-generation algorithm. Speedups are shown in a per-layer manner, showing which layers of the trees generated fastest and which relatively had the greatest amount of overhead.

7.1 VERIFICATION OF EXTRA-TREES

An experiment was performed to check the validity of the parallel Extra-Trees tree-building algorithm. A known function was approximated and evaluated using both the sequential (CPU) and parallel (GPU) implementations. This section shows that the parallel and sequential methods indeed had equal results.

7.1.1 *Approximating a Function*

For this experiment the following sinusoid was considered:

$$y(x) = \sin(x) + 0.5\sin(2x) + 3\sin(0.5x + 0.2) \quad . \tag{7.1}$$

One thousand random x-values were picked from the domain [0, 50]. The correct y-value for each of these points was calculated, giving a sample set with 1000 points. These (x,y)-points were fed as traintuples into the Extra-Trees tree-building algorithm. So, Extra-Trees now saw 1000 random (x,y) samples and built 20 trees approximating a function describing these samples. This was done using both the sequential and parallel Extra-Trees algorithms.

7.1.2 *Evaluating the Function Approximation*

The function approximation was evaluated by picking 1000 new random x-values from the same domain. For each of these values, the following figures were calculated:

1. The true y-value, found by evaluating Equation 7.1
2. The approximated y-value, found by passing the x value through the trees
3. The *approximation error*, which is the difference between the two y-values

With these values, the average approximation error for the 1000 data points could be calculated for the sequential and parallel algorithms.

7.1.3 *Results*

Since trees built by Extra-Trees use random numbers, the experiment was performed ten times. The result of one of the trials is shown in Figure 7.1. The entire x-domain of [0, 50] is shown, on which the graphs seem to overlap completely. Only when looking at a detailed view the differences between the true y-values and the approximations are visible, as shown in Figure 7.2. Still, both the CPU and GPU approximations closely follow the true values.

The results for all ten experiments combined are shown in Table 7.1. In order to verify that the average error was indeed the same in both cases, a two-sample t-Test [23, Section 1.3.5.3] was performed. The following hypotheses were used:

$$\begin{aligned} H_0 &: \mu_{sequential} = \mu_{parallel} \\ H_a &: \mu_{sequential} \neq \mu_{parallel} \quad . \end{aligned} \tag{7.2}$$

The null hypothesis H_0 is rejected when the absolute value of the test statistic T is greater than the critical value of the t-distribution: $|T| >$ critical value. The test statistic is calculated as follows:

$$T = \frac{\mu_s - \mu_p}{\frac{\sigma_s^2}{N_s} + \frac{\sigma_p^2}{N_p}} = 0.0150 \quad . \tag{7.3}$$

To look up the critical value, the number of degrees of freedom needs to be calculated:

$$\nu = \frac{\left(\frac{\sigma_s^2}{N_s} + \frac{\sigma_p^2}{N_p}\right)^2}{\frac{\left(\frac{\sigma_s^2}{N_s}\right)^2}{N_s - 1} + \frac{\left(\frac{\sigma_p^2}{N_p}\right)^2}{N_p - 1}} = 17.3 \quad . \tag{7.4}$$

Using a significance level of 5% and 17 degrees of freedom, the critical value 1.740 was found. Since $|T| \leqslant 1.740$, the null hypothesis was accepted. Thus, the sequential and parallel implementations of Extra-Trees tree-generation had equal average errors on 1000 data points. This proved the fact that both implementations granted the same results.

Table 7.1: Results of approximating a known function (Equation 7.1) ten times using the sequential and parallel implementations of Extra-Trees.

	Sequential	**Parallel**
Average number of nodes per tree	1449	1461
Average error per data point (μ)	8.57×10^{-3}	8.45×10^{-3}
Maximum error	0.24	0.19
Error variance (σ^2)	4.13×10^{-4}	2.75×10^{-4}

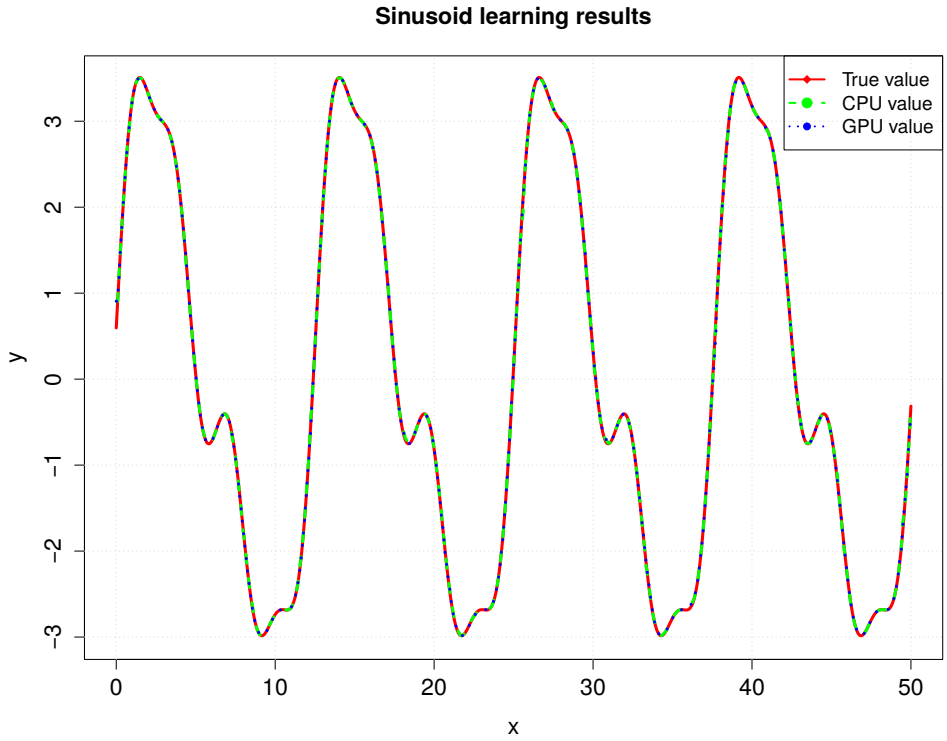


Figure 7.1: Learning result for $y(x) = \sin(x) + 0.5\sin(2x) + 3\sin(0.5x + 0.2)$ on the entire domain. The graphs seem to overlap completely.

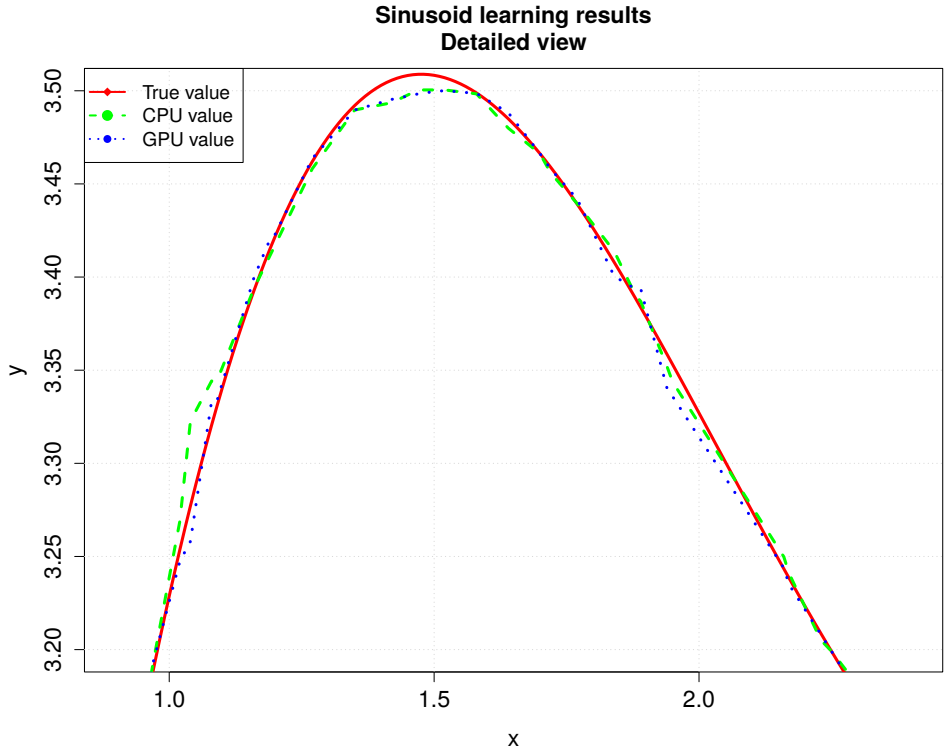


Figure 7.2: Detailed view of the sinusoid learning results. Small differences between the true y-value and the approximations are visible.

7.2 PUDDLE WORLD SPEEDUP USING KD-TREES

With KD-Trees the speedup for learning the Puddle World task was determined for an increasing amount of samples. The worlds as shown in Table 7.2 were used.[1] As explained in Chapter 5, the KD-Trees algorithm mainly consists of generating the training set and updating the tree (i.e. updating the average traintuple output of each leaf). On the CPU, the former consistently took up about 99% of the total algorithm time, as the amount of leaves was approximately 1% of the amount of samples. Because of this it was initially decided to parallelize training set generation only, while keeping the tree update on the host. The resulting speedups are shown in Figure 7.3. It can be seen that the speedup kept increasing when samples were added, reaching about 35 times at 33 million samples. Since the tree update was done on the host, indeed no speedup was visible in the tree update itself. Training set generation on the other hand achieved speedups exceeding 105 times.

The fact that the overall speedup is much lower than the training set generation speedup, shows that the tree update slowed down the overall program. It was therefore to decided to parallelize the tree update as well, so that both training set generation and the tree update were done on the GPU. The speedup that was achieved in this case is shown in Figure 7.4. The speedup of tree generation still lay around 100 times, but this time the tree update achieved the same speedup, or even more. This pushed the speedup of the overall algorithm to over 105 times as well.

Table 7.2: Puddle Worlds that were used in the KD-Trees experiments and the corresponding tree characteristics.

Puddle World size	**Number of samples**	**Nodes in tree**	**Leaves in tree**
16 x 16	26.1×10^3	0.72×10^3	0.36×10^3
32 x 32	78.8×10^3	2.64×10^3	1.32×10^3
64 x 64	0.42×10^6	11.1×10^3	5.54×10^3
128 x 128	2.23×10^6	43.3×10^3	21.7×10^3
256 x 256	9.52×10^6	0.18×10^6	87.7×10^3
512 x 512	33.3×10^6	0.70×10^6	0.35×10^6

7.2.1 *Discussion*

The fact that the tree update reduced the speedup to about 35 times in Figure 7.3 can be explained as follows. When the training set has been generated on the GPU, it needs to be read back to the CPU in order to update the leaves. After that, the updated leaves need to be written back to the GPU in order to generate the next training set. These memory transfers are slow and hence reduced the speedup. When the tree is updated on the GPU as well, these memory transfers do not need to be performed since the training set and the leaves can then remain on the GPU. So, in addition to the speedup of the tree update itself, the slow memory communication is avoided as well. These two things combined lead to a speedup exceeding 105 times.

1 Note that neither FQI nor the Puddle World task is limited to the world sizes used.

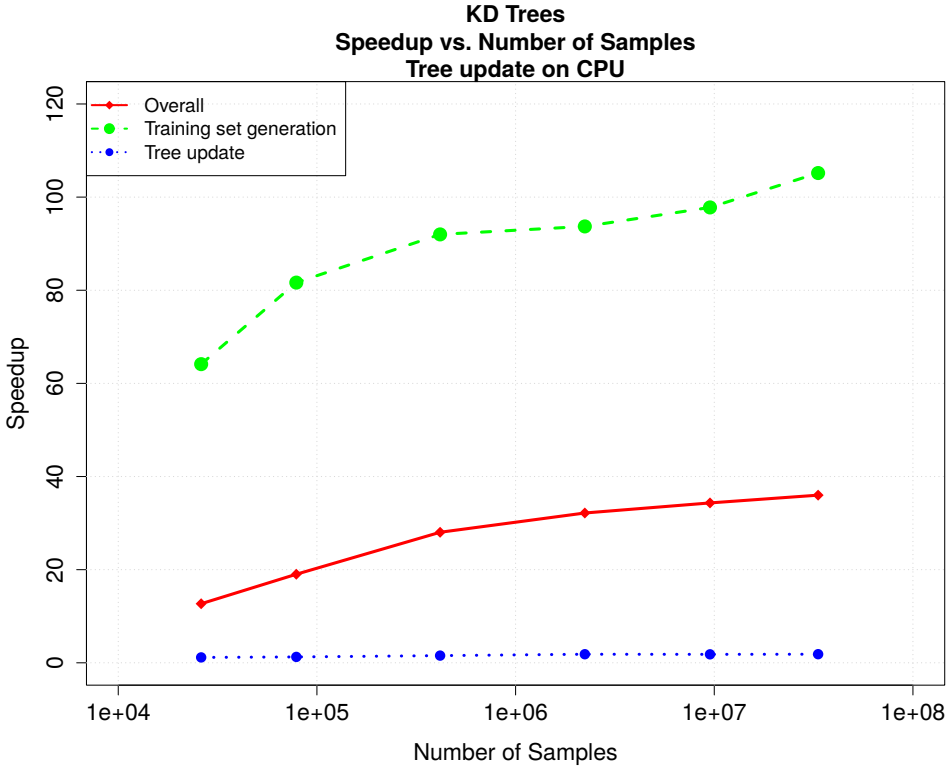


Figure 7.3: KD-Trees speedup with leaf updates on the host.

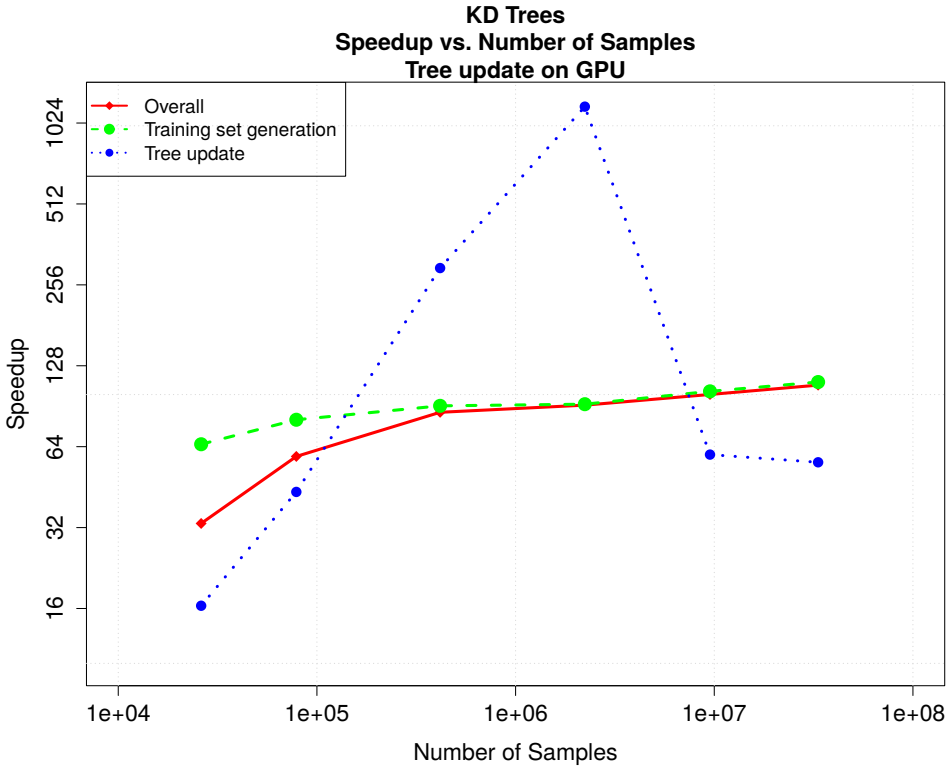


Figure 7.4: KD-Trees speedup with leaf updates on the device.

7.3 PUDDLE WORLD SPEEDUP USING EXTRA-TREES

During the Extra-Trees experiments, the Puddle Worlds shown in Table 7.3 were used. Recall from Chapter 5 that Extra-Trees generates new trees at every iteration of the FQI algorithm, besides generating a training set. Figure 7.5 shows the division of time between these two parts on the CPU, for a typical Extra-Trees configuration with 20 trees. It shows that tree generation was indeed a computationally intensive task on the CPU, taking up over 60% of the total algorithm time. Increasing the amount of samples did not change the division of time to a great extent, since it affects both training set generation (more samples need to be converted) and tree generation (more samples need to be split).

Only when varying the amount of trees did the time division change, as shown in Figure 7.6. With an increasing number of trees, training set generation became more and more computationally expensive. The reason for this is assumed to be the fact that caching of nodes during pass-throughs of successive samples becomes less efficient when more trees are added. With fewer trees there are fewer nodes, which means caching of the nodes is more efficient as they are used during the pass-through of many samples in a row. When more trees are added, more nodes are used and caching becomes less efficient.

Table 7.3: Puddle Worlds that were used in the Extra-Trees experiments.

Puddle World size	**Number of samples**	**Number of trees**
16 x 16	12.8×10^3	20
32 x 32	44.1×10^3	20
64 x 64	0.16×10^6	20
128 x 128	0.53×10^6	$1, 2, 5, 10, 20, 50, 100$

Since both training set generation and tree-building took a substantial amount of time on the CPU, the parallel implementation performed both these parts on the GPU. The speedups in learning the Puddle World task with Extra-Trees were calculated for two cases: one where the amount of samples was varied, and another which varied the amount of trees.

The speedup achieved when varying the amount of samples is shown in Figure 7.7. At low amounts of samples the speedup was low, increasing as more samples were added. Eventually, at nearly one million samples, the overall speedup was just over 8 times. It follows the trend shown by the tree generation speedup.

When the amount of trees was increased, as shown in Figure 7.8, the speedups of both parts of the algorithm increased further, taking the overall speedup with them to nearly 20 times. No clear speedup ceiling was reached; even at 100 trees the trends were still upwards.

7.3.1 *Discussion*

The speedup graphs look like what was expected: the speedup increased with an increasing amount of samples and trees. However, problems arose when there were more than one million samples. In that case the indexes of the traintuples used

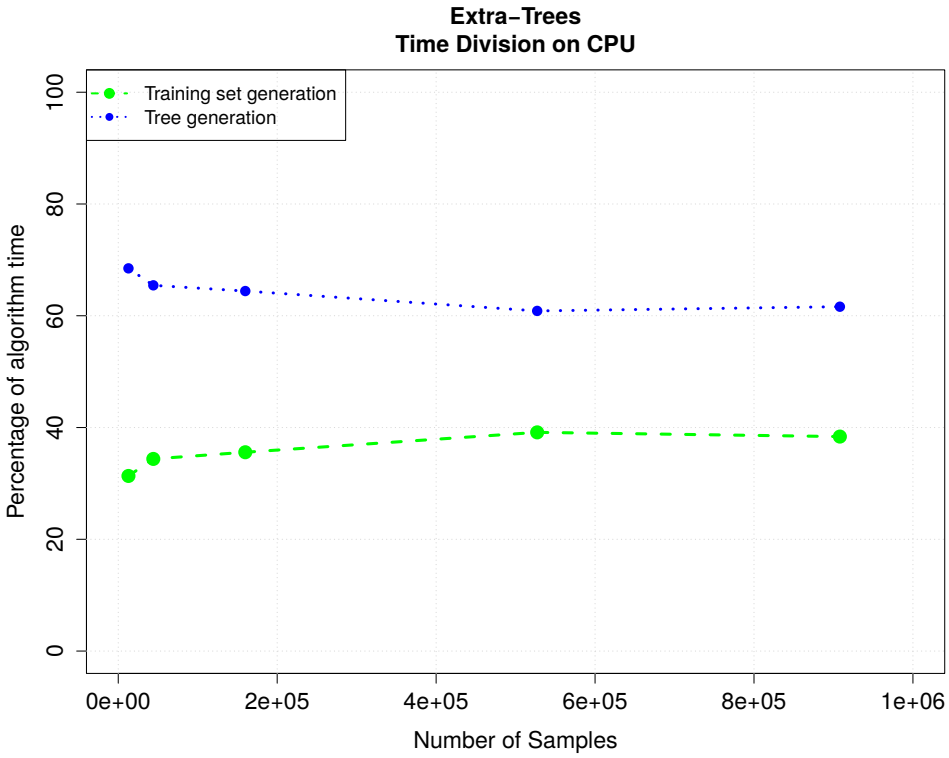


Figure 7.5: Division of time of Extra-Trees on the CPU while varying the amount of samples. A total of 20 trees was used.

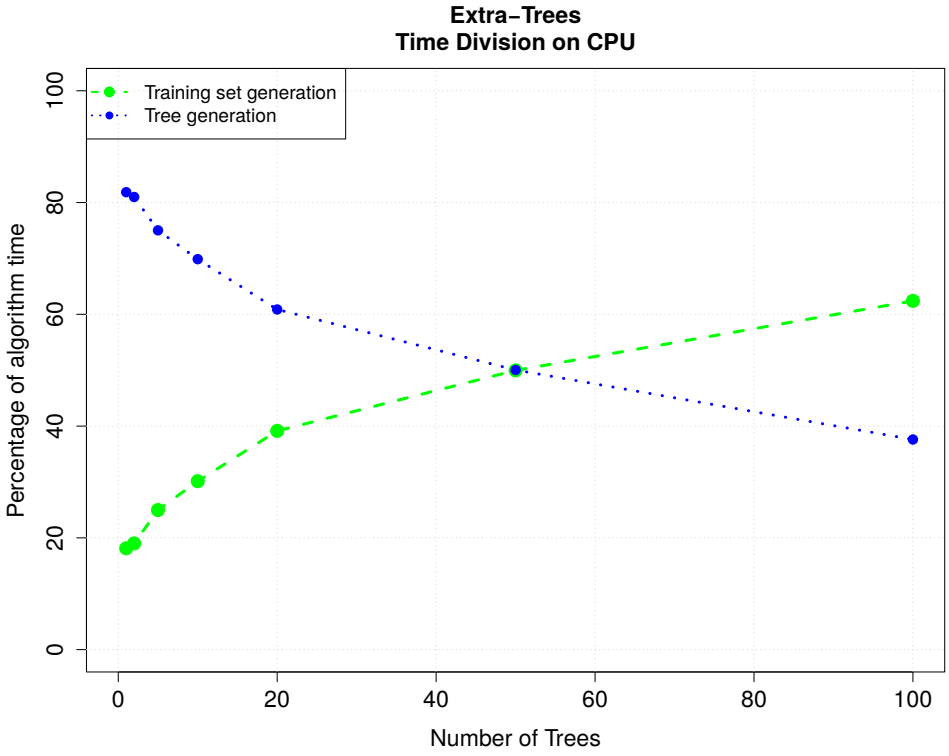


Figure 7.6: Division of time of Extra-Trees on the CPU while varying the amount of trees. A Puddle World of 128x128 was used, with 0.5×10^6 samples.

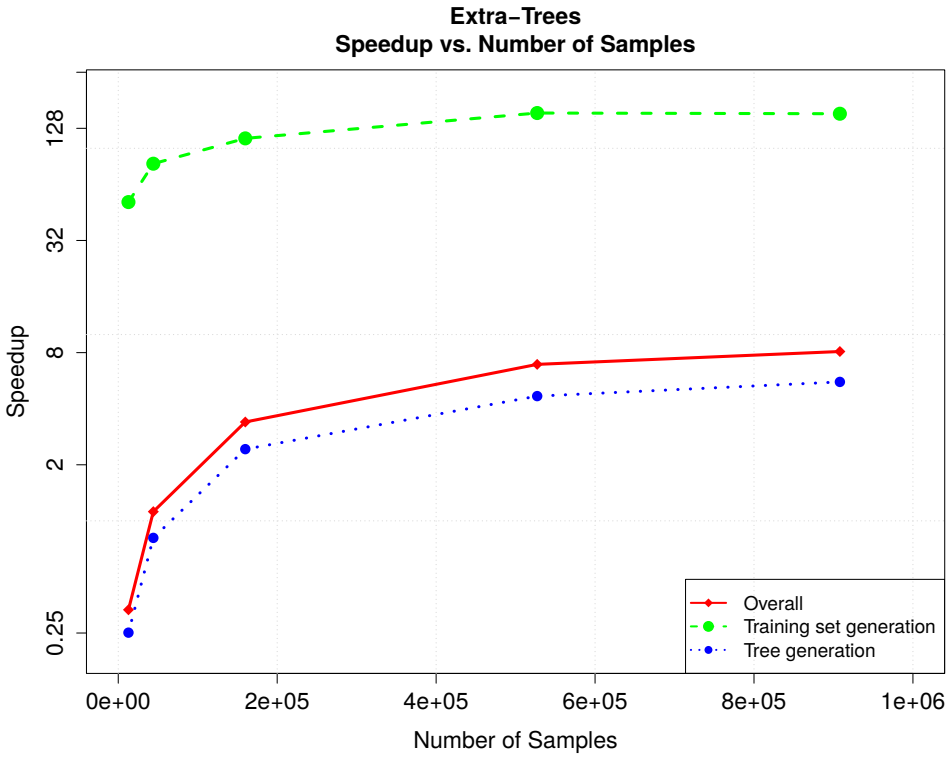


Figure 7.7: Extra-Trees speedup with an increasing amount of samples. A total of 20 trees was used.

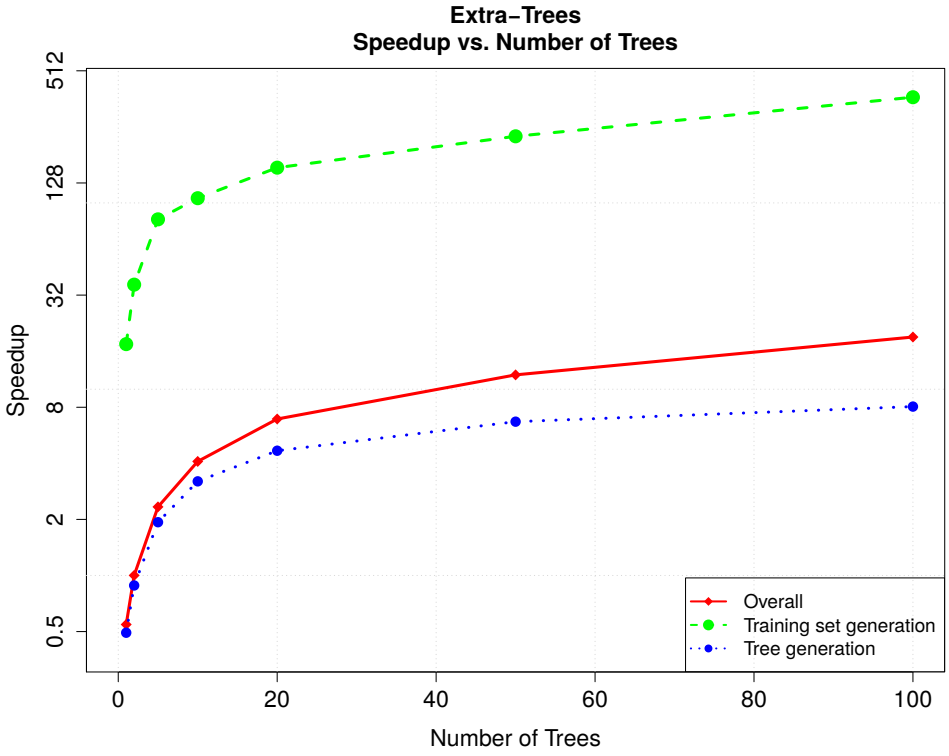


Figure 7.8: Extra-Trees speedup with an increasing amount of trees. A Puddle World of 128x128 was used, with 0.5×10^6 samples.

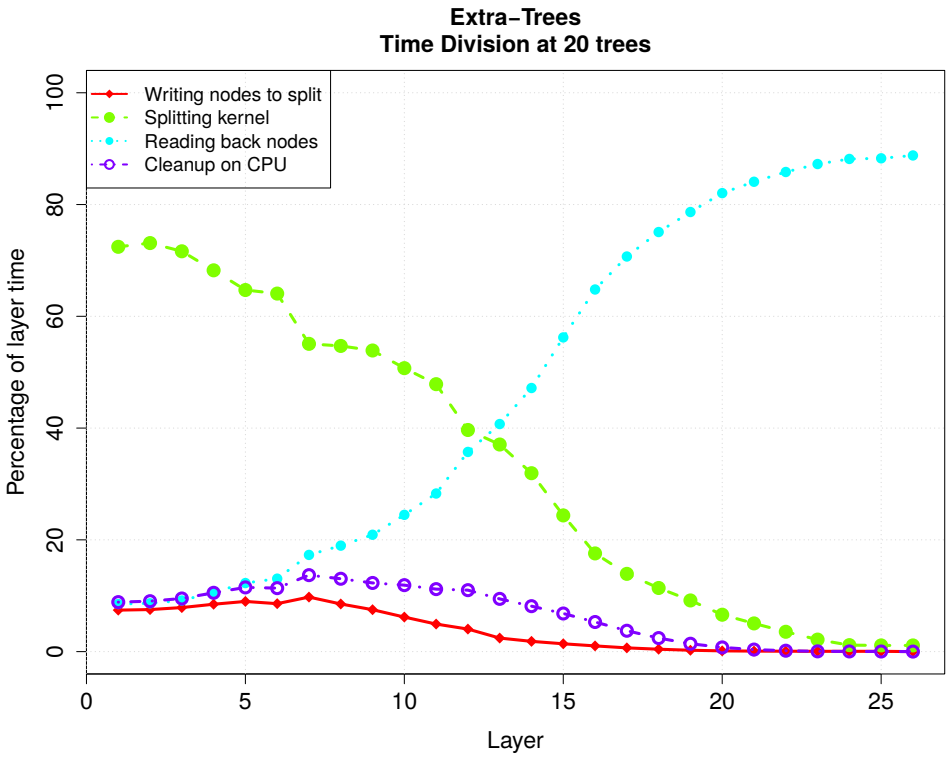


Figure 7.9: Distribution of time per layer when generating 20 trees with Extra-Trees.

by each work-item (described in Chapter 6.2.2) required more than the maximum available private memory. For the Puddle World problem this meant that the world could not be bigger than 128x128 with the current implementation to successfully find an optimal solution. The solution to this problem is to not store the traintuple indexes in private memory but to load them from global memory instead, which will allow bigger worlds to be learned. However, this increase in global memory communication will result in a decrease of speedup.

KD-Trees on the other hand did not have this limitation, as the tree is generated using the CPU.

7.4 EXTRA-TREES TREE-BUILDING

Chapter 5 described that a good approach to parallelizing FQI is to split multiple nodes of multiple trees at the same time, one layer at a time. Figure 6.2 showed how a work-item splits one node of a layer. The previous section demonstrated the speedups of the entire Extra-Trees FQI algorithm, and how the speedups of training set generation and tree-building interplayed. This section provides a more in-depth analysis of the performance of Extra-Trees' tree-building task, showing the speedups that were achieved *within* the building of the trees. First, the time division of the parallel algorithm is shown when generating 20 trees. The actual kernel time is compared to the overhead in a per-layer fashion. After that it is shown how the relative amount of kernel time changed when the amount of trees was varied. Lastly, the speedup per layer is given for different amounts of trees.

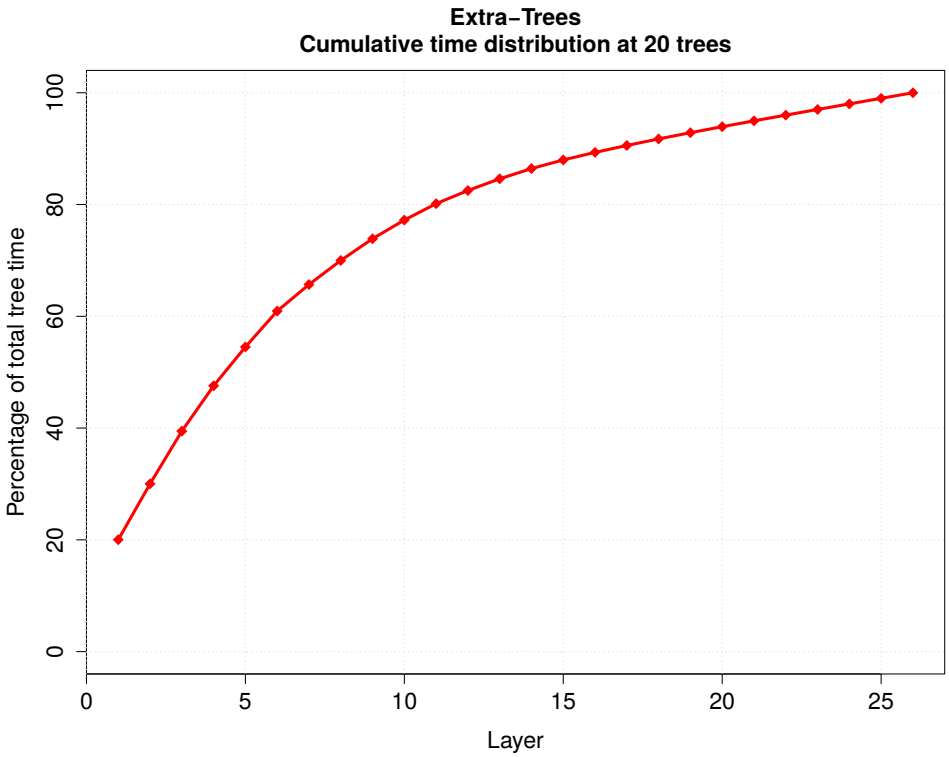


Figure 7.10: Cumulative layer time when generating 20 trees with Extra-Trees.

Figure 7.9 shows how time was divided when 20 trees were generated using the parallel algorithm. In the first 11 layers the algorithm was most efficient: the total overhead was less than the time the splitting kernel itself took. Further optimization of the kernel would have the greatest effect in these layers. From layer 12 on, the overhead of splitting the layer was greater than the splitting kernel itself. Here, reading back the nodes that resulted from the split took most time. Fortunately, the first 11 layers took up approximately 80% of the total tree generation time, as demonstrated by Figure 7.10. This figure also puts the very high node-reading share of the latest layers into perspective. The node-reading time is nearly constant over the layers. Since each of the later layers only took up about one percent of the total tree time, the small layer time makes the node-reading *percentage* very large, while the reading time itself did not change.

The trends of the lines meet the expectations. At earlier layers hardly any leaves have been found, meaning that almost all the traintuples are still used in the splits. The computational intensity of these layers is therefore high, making the splitting kernel take a big share of the time. In later layers more leaves have been found, eliminating many traintuples from further splits. The computational intensity is reduced, which increases the relative overhead of each split.

Figure 7.11 shows that the share of splitting kernel time decreased when more trees were added. This is not caused by the kernel becoming less efficient, but rather by the overhead increasing. With more trees, more nodes are being split, meaning that more data needs to be transferred to and from the GPU and removing leaves (as part of the cleanup on the CPU) takes more time.

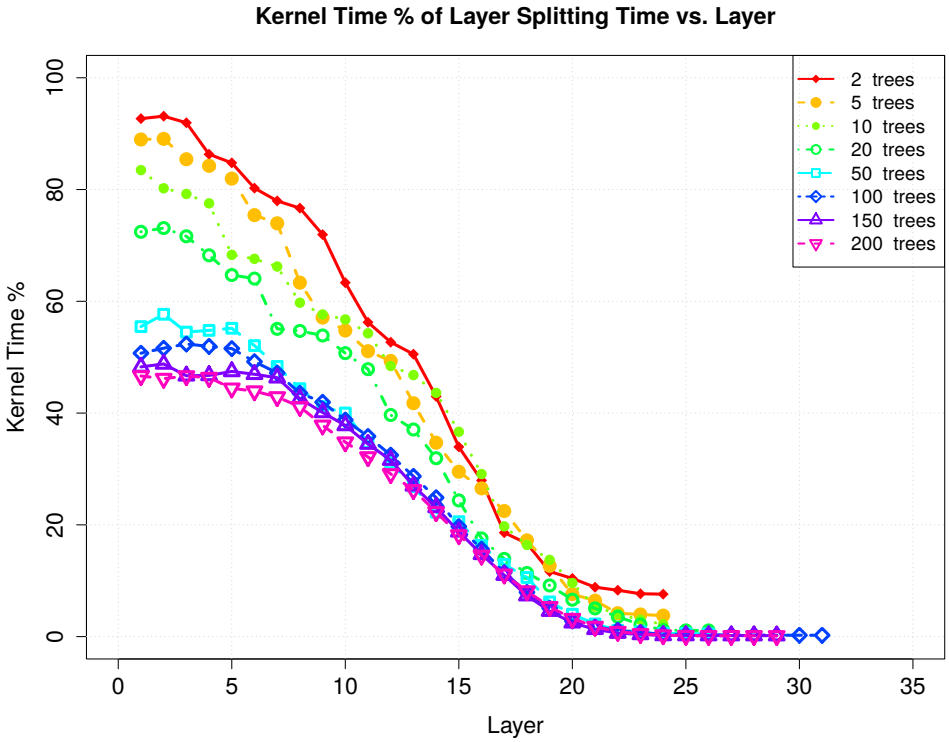


Figure 7.11: Percentage of node-splitting kernel time with respect to layer generation time.

Even though the share of kernel time decreased with an increasing amount of trees, the greatest tree-building speedups were still found at larger amounts of trees. This is shown in Figure 7.12. Having more trees means more nodes need to be split, increasing the degree of parallelism. A maximum speedup of around 11 times was reached for 150 and 200 trees at layers 8 to 10. These layers have the best balance between the amount of nodes and traintuples. In earlier layers the smaller amount of nodes lead to a smaller degree of parallelism, whereas later layers have less traintuples to split, reducing the computational intensity.

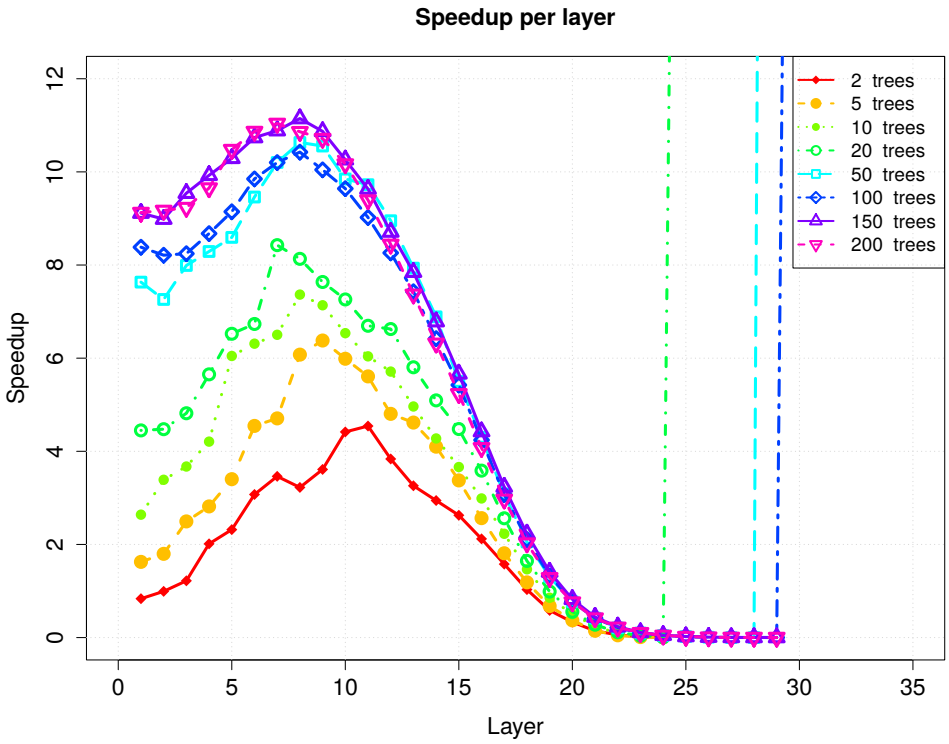


Figure 7.12: Speedup per layer of tree generation in Extra-Trees.

8 DISCUSSION

The results shown in the previous chapter can now be discussed. The first section handles the speedups and resource usage of both KD-Trees and Extra-Trees, and compares them. The second section contains a short interlude showing how state-action transitions could be cached with KD-Trees in order to reduce its computational intensity. The third section then dives into the practical meaning for reinforcement learning of KD-Trees and Extra-Trees on GPUs. It answers the question when to use Extra-Trees or KD-Trees when using a GPU, and highlights a practical example in which the algorithms could be used.

8.1 SPEEDUPS AND RESOURCE USAGE

As expected, the speedup of KD-Trees was much higher than that of Extra-Trees (105 times versus 20), since tree-building is more difficult to parallelize than training set generation. The need for synchronization in Extra-Trees and the amount of host operations required, limited its overall speedup. The graphs of KD-Trees actually showed the effect of moving code to the GPU in order to limit memory transfers and synchronization (in Figure 7.3 and 7.4): the speedup went from 35 to 105 times.

The trend of the speedup of Extra-Trees looks promising (Figure 7.7 and 7.8), as the speedup kept increasing and no clear ceiling was visible. However, according to the paper of Ernst et. al on FQI [9], 20 trees already gives good learning performance. Further increasing the amount of trees would not increase the performance as much as going from 10 to 20 trees would. This is why in most practical learning problems, with this implementation, the speedup would most likely be less than ten times. This is a good speedup, but much lower than the one found with KD-Trees.

Another difference is that Extra-Trees needed quite a big amount of samples in order to have a speedup – several tens of thousands in Figure 7.7 in case of 20 trees. Only then is the computational intensity high enough to overcome the CPU overhead. In later layers the overhead could not be overcome, because of the small amount of traintuples still needing to be split. Executing these splits on the CPU might be a better solution. KD-Trees on the other hand does not have this problem. Even at low amounts of samples a good speedup is found. At several tens of thousands of samples the speedup already is around 50 times in Figure 7.4.

Besides having a greater speedup, KD-Trees is also much less resource intensive. With KD-Trees millions of samples could be used without a problem, whereas Extra-Trees hit memory limitations at around one million samples. There was insufficient private memory for a work-item to store the indexes of the traintuples it needed to process. Streaming the indexes from global memory is a solution to this problem but will decrease speedups.

Extra-Trees needed much more global memory space, too. KD-Trees only uses one tree which does not have to be re-built. The only data residing on the GPU were the sample set, the nodes and leaves of a single tree and space for a training set. With Extra-Trees however, all nodes of all trees needed to be stored for the pass-through.

Furthermore, generating trees layer-by-layer meant that input *and* output buffers needed to be allocated (e.g. `nodes_in` and `nodes_out`; `minmax_in` and `minmax_out`). Changing data in-place was not considered an option, since race conditions would surely occur: one work-item might expect an old value from the buffer while another work-item already wrote a new value to it. Ample space needs to be pre-allocated on the GPU, as it is difficult to predict how many nodes the trees will have. Because of this, with very big problems it might be possible that Extra-Trees hits the global memory limit, while KD-Trees would have sufficient space to store its data.

All in all, with respect to Extra-Trees, KD-Trees:

1. could solve much larger problems because of less private memory usage,
2. already achieved speedups at much smaller problems,
3. achieved greater speedups overall, and
4. used less global memory.

8.2 TRANSITION CACHING IN KD-TREES

There is also a different, smart way of implementing KD-Trees which greatly decreases the computational intensity without sacrificing results. Since the tree structure is constant throughout the iterations of FQI, all possible state-action transitions could be *cached* before the iterations start. For example, every possible state-action combination could be passed through the tree and the resulting leaf index could be stored, resulting in a map with format:

$$\mathtt{map}[s][a] = \mathtt{leaf\_idx} \quad .$$

It would be even more efficient to only evaluate every action in every sample's successor state. The result could be added as an array to the sample:

$$\mathtt{sample.next\_action\_leaves} = [\mathtt{leaf\_idx\_1}, \mathtt{leaf\_idx\_2}, ..., \mathtt{leaf\_idx\_}\mathcal{A}] \quad ,$$

in which `leaf_idx_1` would be the index of the leaf corresponding to taking action 1 in the sample's successor state.

With these approaches, after caching the transitions, not a single pass-through needs to be done anymore during FQI's iterations. Since the pass-through is the most computationally intensive part of the algorithm, this makes the CPU algorithm much faster and speedups are greatly reduced. A small experiment demonstrated that the speedup would be reduced to under five times.

However, this only works for discrete state-action spaces. If the state-action space is continuous, there is not a finite set of combinations to evaluate. But for discrete problems this might be an efficient way of learning a task with KD-Trees.

8.3 COMPARING KD-TREES AND EXTRA-TREES IN REINFORCEMENT LEARNING

Still, in Chapter 5 it was explained that Extra-Trees should be able to achieve better learning accuracy than KD-Trees in fewer iterations. This actually raises the interesting question which algorithm is the best choice to use on a GPU. In the same

amount of time, which parallel algorithm would achieve better results? The results obtained during a small experiment are shown in Table 8.1. Two Puddle Worlds with specific sample sets were learned by KD-Trees as well as Extra-Trees. While Extra-Trees needed fewer iterations to find the optimal solution to the problems, KD-Trees clearly managed to solve them much faster.

More detailed results of solving the 128x128 Puddle World with varying sample set sizes are shown in Figure 8.1 and 8.2. FQI iterations were stopped when the optimal solution was found, or when 500 Extra-Trees iterations or 1000 KD-Trees iterations were performed[1]. At low amounts of samples, Extra-Trees already found a return close to the optimal one, while KD-Trees was still far from optimal. With 0.3 million samples Extra-Trees could find the optimal solution. KD-Trees on the other hand needed more than a million samples[2].

However, when KD-Trees had enough samples to find the optimal solution, it did so hundreds of times faster than Extra-Trees as shown in Figure 8.2. KD-Trees found the solution in about a second, performing around 230 iterations, while Extra-Trees generally needed almost nine minutes to perform around 200 iterations.

These results show that Extra-Trees is the more flexible algorithm, as expected, requiring less data to find the optimal solution. But once KD-Trees has enough samples to find the optimal solution, it does so orders of magnitude faster than Extra-Trees.

All in all it can be concluded that if samples can be obtained quickly and easily in a learning problem, KD-Trees will find an optimal solution orders of magnitude faster than a sequential or Extra-Trees (sequential or parallel) implementation would.

An example in which GPU KD-Trees could be used well, is the following. Mnih et al. [22] use deep reinforcement learning to learn control policies for Atari games, solely based on pixel information of the screen of the games. Vast amounts of data are used to learn these problems, which could possibly be processed well using KD-Trees on a GPU. The same holds for other big data learning problems.

Table 8.1: Solving the same Puddle World with the same samples with both KD-Trees and Extra-Trees (20 trees).

World	**Samples**	**Algorithm**	**Trees**	**GPU Iter.**	t_{GPU} **(sec)**	t_{CPU} **(sec)**	**Speedup**
64×64	47938	KD-Trees	1	102	0.03	1.35	44.1
64×64	47938	Extra-Trees	20	65	73.5	210.3	2.9
128×128	492500	KD-Trees	1	225	0.47	43.39	92.3
128×128	492500	Extra-Trees	20	183	714	6745	9.5

1 This difference in iterations is there since KD-Trees generally requires more iterations than Extra-Trees to solve a learning problem.

2 Just over 0.5 million samples it looks like KD-Trees found the optimal solution. However, the return was still slightly below optimal.

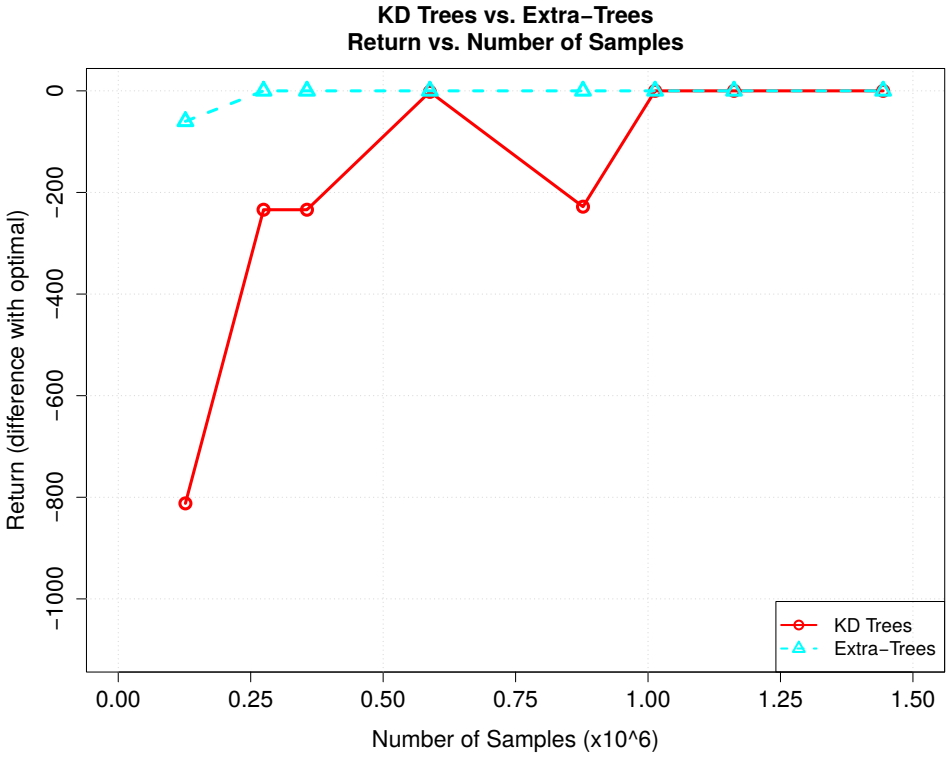


Figure 8.1: Solving the same 128x128 Puddle World with the same samples using KD-Trees and Extra-Trees (20 trees). Extra-Trees found the optimal solution with many fewer samples than KD-Trees.

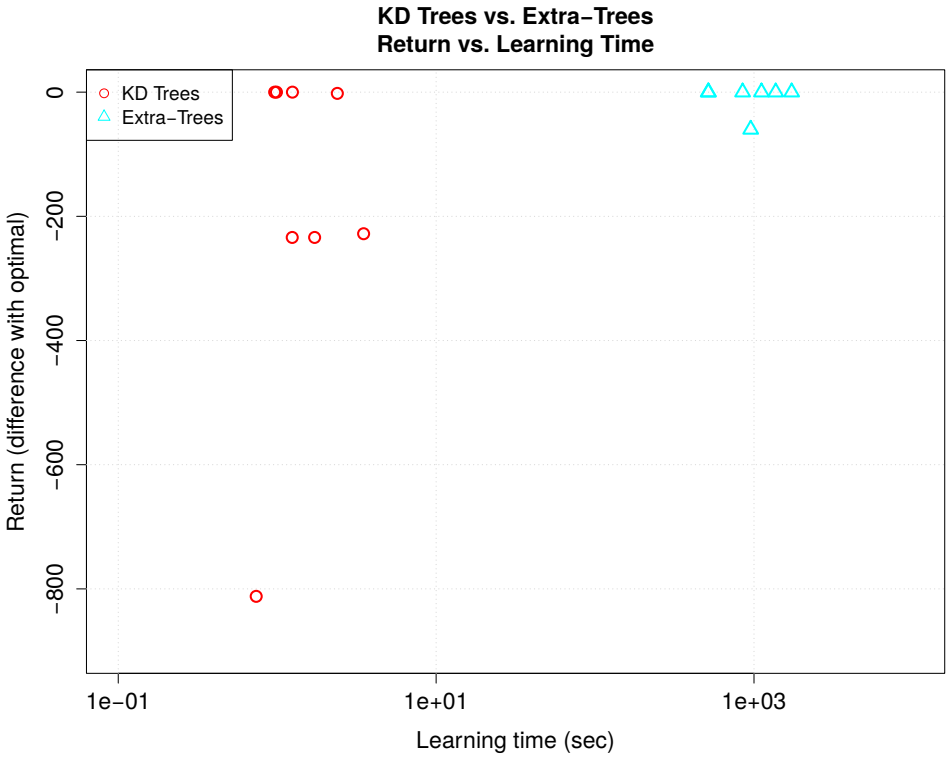


Figure 8.2: Learning times corresponding to Figure 8.1. With enough samples to find the solution, KD-Trees did so much quicker than Extra-Trees.

9 CONCLUSIONS

Research was performed to assess to what extent reinforcement learning can achieve speedups, when it is parallelized and executed on a graphics processing unit. Fitted Q Iteration (FQI) was found to be a suitable reinforcement learning method to speed up. Within FQI, tree-based FQI methods KD-Trees and Extra-Trees were chosen to parallelize – KD-Trees for its excellent parallelization potential and Extra-Trees for its accurate learning performance.

Parallel implementations were designed, and tested using the Puddle World task. FQI can indeed achieve significant speedups by programming it for a GPU. During the experiments KD-Trees achieved speedups of around 105 times, whereas Extra-Trees' speedup was limited to around 20 times. Since trees need to be re-built during every iteration with Extra-Trees, memory usage of Extra-Trees was much greater than that of KD-Trees. Host cleanups were needed in order to keep this within bounds. Still, private memory limited the size of problems the parallel implementation of Extra-Trees could solve. Generating trees on the GPU also requires synchronization: splitting results of parent nodes need to be known before its children can be split.

The synchronization and cleanups lead to an overhead that could only be overcome when the problem to be learned was large. KD-Trees did not have this problem and therefore also achieved much greater speedups at small problems. All in all, with respect to Extra-Trees, KD-Trees:

1. could solve much larger problems because of less private memory usage,
2. already achieved speedups at much smaller problems,
3. achieved greater speedups overall, and
4. used less global memory.

When trying to learn a specific Puddle World with the same samples, KD-Trees was able to do so hundreds of times faster than Extra-Trees. However, Extra-Trees could solve the problem with fewer samples and in fewer iterations.

In reinforcement learning this means that especially problems with many samples or problems in which samples are obtained quickly and easily, could greatly benefit from learning using KD-Trees on a GPU. A system could learn tasks or adapt to changes in its environment orders of magnitudes faster than with a sequential implementation.

9.1 FUTURE WORK

The results presented in this report prove that especially KD-Trees is able to achieve incredible speedups in learning a task. The possibilities that these speedups grant in terms of new applications or improving existing applications of FQI or RL in general, could be researched. The research shown only focused on FQI. Further research might investigate the parallelization potential of other or newer reinforcement learning algorithms.

The parallel implementation of Extra-Trees could be improved further. For example, private memory usage in the kernels could be limited in order to make it possible to learn larger tasks. This would likely lead to more global memory usage and thus a reduction of speedups, but it would make the implementation more flexible.

Another way of improving Extra-Trees would be to only perform a cleanup after splitting several layers, instead of after each layer. This way the amount of memory transfers between CPU and GPU and the overhead would be reduced, increasing efficiency.

Lastly, with Extra-Trees, different parallelization strategies could be combined. The current implementation used only one strategy (splitting layer-by-layer), but using different strategies for specific nodes or layers might make better use of the GPU. For example, in the first couple of layers, there are not enough nodes to let the entire GPU do work. Splitting these layers in a node-by-node fashion is likely to increase efficiency. Also, when the overhead of a split becomes greater than its kernel time, it might be better to perform the remaining splits on the CPU. This way the overhead is avoided.

Using a different GPU might also increase speedups. On the software side, the NVIDIA C2075 is still stuck on OpenCL 1.1 (as are all NVIDIA GPUs). OpenCL 1.2 implementations for AMD GPUs have been available since January 2012, and even OpenCL 2.0 implementations have been available since late 2014. NVIDIA's dated OpenCL implementation might therefore lack many improvements and new features. Running the parallel implementations using newer versions of OpenCL might therefore increase performance.

On the hardware side, many improvements have been made as well. The C2075 (released in 2011) features 14 compute units with 32 processing elements each (448 in total), running at 1.15 GHz. An AMD R9 290x (released in 2014) has 44 compute units with 64 processing elements each (2816 in total), running at 1.00 GHz. This increase in compute units and processing elements could make the implementations execute much faster, despite the lower clock frequency. The advantage of the C2075 is that it has 6 GB of global memory, whereas the R9 290x has 4 GB, both of the GDDR5 type. The R9 290x in turn has a much higher memory bandwidth at 320 GB/sec, compared to the 144 GB/sec of the C2075. So, if the parallel implementation can stick to the limit of 4 GB global memory, the R9 290x seems to be a better choice.

Debugging and profiling on AMD GPUs is much easier, too, with AMD's CodeXL software. A similar tool for NVIDIA GPUs is not available. Using tools like CodeXL makes development easier and faster. Bottlenecks in performance can also be identified more easily, possibly leading to better optimized code.

BIBLIOGRAPHY

[1] Shameen Akhter and Jason Roberts. *Multi-core programming*. Vol. 33. Intel press Hillsboro, 2006 (cit. on p. 9).

[2] Ian Buck. In: *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Ed. by Matt Pharr, Randima Fernando, and Tim Sweeney. 2005. Chap. Taking the Plunge into GPU Computing (cit. on p. 12).

[3] Ian Buck et al. "Brook for GPUs: stream computing on graphics hardware". In: *ACM Transactions on Graphics (TOG)*. Vol. 23. 3. ACM. 2004, pp. 777–786 (cit. on p. 33).

[4] Lucian Busoniu et al. *Reinforcement learning and dynamic programming using function approximators*. Vol. 39. CRC Press, 2010 (cit. on p. 31).

[5] Wouter Caarls et al. "Parallel Reinforcement Learning of Single Tasks Without a Presupplied Transition Model". Submitted to: *Cybernetics, Transactions on Neural Networks and Learning Systems*. 2012 (cit. on p. 33).

[6] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113 (cit. on p. 33).

[7] Marc Peter Deisenroth and Carl Edward Rasmussen. "PILCO: A Model-Based and Data-Efficient Approach to Policy Search". In: *Proceedings of the 28th International Conference on Machine Learning*. 2011 (cit. on p. 31).

[8] Wenliang Du and Zhijun Zhan. *Building Decision Tree Classifier on Private Data*. Syracuse University, Department of Electrical Engineering and Computer Science, 2002. URL: http://surface.syr.edu/eecs/8 (cit. on p. 30).

[9] Damian Ernst, Pierre Geurts, and Louis Wehenkel. *Tree-Based Batch Mode Reinforcement Learning*. Tech. rep. University of Liège, Department of Electrical Engineering and Computer Science, Apr. 2005 (cit. on pp. 7, 26, 30, 32, 35, 36, 42, 71).

[10] David Geer. "Chip makers turn to multicore processors". In: *Computer* 38.5 (2005), pp. 11–13 (cit. on p. 9).

[11] Håkan Grahn et al. "CudaRF: A CUDA-based implementation of random forests". In: *Computer Systems and Applications (AICCSA), 2011 9th IEEE/ACS International Conference on*. IEEE. 2011, pp. 95–101 (cit. on p. 41).

[12] Khronos OpenCL Working Group. *The OpenCL specification, version 1.2, revision 19*. 2012 (cit. on pp. 16, 19, 20).

[13] Jim Handy. *The cache memory book*. Morgan Kaufmann Pub, 1998 (cit. on p. 11).

[14] Mark Harris. In: *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Ed. by Matt Pharr, Randima Fernando, and Tim Sweeney. 2005. Chap. Mapping Computational Concepts to GPUs (cit. on pp. 13, 15).

[15] *Horde implemented in opencl*. URL: http://code.google.com/p/opencl-horde/ (visited on 06/26/2013) (cit. on p. 33).

[16] Intel. *News Fact Sheet, Intel Xeon Processor E5-2600 Product Family*. Tech. rep. Mar. 2012 (cit. on p. 57).

[17] Karl Jansson, Håkan Sundell, and Henrik Boström. "Parallel tree-ensemble algorithms for GPUs using CUDA". In: *Sixth Swedish Workshop on Multicore Computing*. 2013 (cit. on pp. 39, 41).

[18] R Matthew Kretchmar. "Reinforcement learning algorithms for homogenous multi-agent systems". In: *Workshop on Agent and Swarm Programming*. 2003 (cit. on p. 33).

[19] Michail G. Lagoudakis and Ronald Parr. *Least-Squares Policy Iteration*. Tech. rep. Duke University, Department of Computer Science, Dec. 2003 (cit. on pp. 25, 30, 31).

[20] Yuxi Li and Dale Schuurmans. "MapReduce for parallel reinforcement learning". In: *Recent Advances in Reinforcement Learning*. Springer, 2012, pp. 309–320 (cit. on p. 33).

[21] R.P. Lippmann. "An introduction to computing with neural nets". In: *ASSP Magazine, IEEE* 4.2 (1987), pp. 4–22 (cit. on p. 27).

[22] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *arXiv preprint arXiv:1312.5602* (2013) (cit. on p. 73).

[23] NIST/SEMATECH. *e-Handbook of Statistical Methods*. http://www.itl.nist.gov/div898/handbook/. 2015 (cit. on p. 60).

[24] NVIDIA. *NVIDIA GF100, World's Fastest GPU Delivering Great Gaming Performance with True Geometric Realism*. Tech. rep. 2010 (cit. on pp. 10, 57).

[25] NVIDIA. *NVIDIA Tesla C2075 Companion Processor*. Tech. rep. 2011 (cit. on p. 57).

[26] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. Tech. rep. 2009 (cit. on pp. 10–12).

[27] NVIDIA. *OpenCL Best Practices Guide*. Tech. rep. Feb. 2011 (cit. on pp. 13, 14).

[28] Kyoung-Su Oh and Keechul Jung. "GPU implementation of neural networks". In: *Pattern Recognition* 37.6 (2004), pp. 1311–1314 (cit. on p. 33).

[29] John Owens. In: *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Ed. by Matt Pharr, Randima Fernando, and Tim Sweeney. 2005. Chap. Streaming Architectures and Technology Trends (cit. on pp. 8, 9).

[30] Victor Palmer. "Multi-Agent Least-Squares Policy Iteration". In: *FRONTIERS IN ARTIFICIAL INTELLIGENCE AND APPLICATIONS* 141 (2006), p. 733 (cit. on p. 33).

[31] Arun Kumar Parakh, M Balakrishnan, and Kolin Paul. "Performance estimation of gpus with cache". In: *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE. 2012, pp. 2384–2393 (cit. on p. 11).

[32] J.H. Postma. *Speeding Up Reinforcement Learning With Graphics Processing Units*. Tech. rep. Delft University of Technology, Jan. 2014 (cit. on p. 31).

[33] Martin Riedmiller. *Neural Fitted Q Iteration - First Experiences with a Data Efficient Neural Reinforcement Learning Method*. Tech. rep. University of Osnabrück, Neuroinformatics Group, 2005 (cit. on p. 32).

[34] Richard S Sutton. "Generalization in reinforcement learning: Successful examples using sparse coarse coding". In: *Advances in neural information processing systems* (1996), pp. 1038–1044 (cit. on pp. 7, 57).

[35] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, 1998 (cit. on pp. 22, 24–27, 29, 31, 32).

[36] Richard S. Sutton et al. "Horde: A Scalable Real-time Architecture for Learning Knowledge from Unsupervised Sensorimotor Interaction". In: *Proc. of 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2011)*. Ed. by Tumer et al. May 2011, pp. 761–768 (cit. on p. 32).

[37] Richard S. Sutton et al. "Policy Gradient Methods for Reinforcement Learning with Function Approximation". In: *Advances in Neural Information Processing Systems 12*. 2000, pp. 1057–1063 (cit. on p. 31).

[38] Guangming Tan and Guang R. Gao. *A parallel dynamic programming algorithm on a multi-core architecture*. Tech. rep. University of Delaware, Department of Electrical, Computer Engineering, Computer Architecture, and Parallel Systems Laboratory, Feb. 2007 (cit. on p. 33).

[39] Ming Tan. "Multi-Agent Reinforcement Learning: Independent vs. Cooperative Agents". In: *Proceedings of the Tenth International Conference on Machine Learning*. Morgan Kaufmann, 1993, pp. 330–337 (cit. on p. 33).

[40] David Wingate and Kevin D. Seppi. "P3VI: A Partitioned, Prioritized, Parallel Value Iterator". In: *Proceedings of the 21st International Conference on Machine Learning*. 2004 (cit. on p. 32).

[41] Wm A Wulf and Sally A McKee. "Hitting the memory wall: implications of the obvious". In: *ACM SIGARCH computer architecture news* 23.1 (1995), pp. 20–24 (cit. on p. 8).