NTNU

Norwegian University of
Science and Technology

Load Balancing of Pseudo-random Workloads on Heterogeneous Systems

Anders Wenhaug

Master of Science in Computer Science
Submission date: August 2017
Supervisor: Anne Cathrine Elster, IDI
Co-supervisor: Jan Christian Meyer, IDI

Norwegian University of Science and Technology
Department of Computer Science

Problem description

Pseudo-random workloads and branch divergence present load balancing challenges to heterogeneous systems using GPUs as accelerators. A method to partition an irregular workload between host system processors and accelerators should be developed, and compared to conventional load balancing methods from the literature.

Abstract

Heterogeneous computing systems using one or more graphics processing units (GPUs) as accelerators present unique load balancing challenges due to the architecture of the GPUs. Assigning a part of the workload proportional to the throughput of the GPU is unlikely to achieve the peak theoretical performance of the GPU, partly because of branch divergence. Additionally, for workloads depending on pseudo-random numbers, the branch divergence may appear unpredictable, making it hard to work around.

In this thesis we present an approach for reorganizing pseudo-random workloads before execution on the GPU, with the goal of reducing the branch divergence. In our experiments, the method achieves a speedup in kernel execution time of up to 1.45 on a real application. We also show that the method may be faster even if the overhead of it is accounted for. Additionally, a method for estimating the resulting reduction in execution time is developed, which can be used for determining whether or not to apply the reorganization.

A graph based method for task balancing is also presented, which is able to select the optimal task sequence in over 96% of the tested cases. This task graph doubles as a model for the throughput of the GPU, and the estimates are used by a load balancer

to partition the workload between the central processing unit (CPU) and GPU.

Sammendrag

Heterogene systemer som bruker grafikkprosessorer (GPUer) som akseleratorer medfører unike problemer med tanke på lastbalansering på grunn av arkitekturen til GPUer. Å tildele en mengde arbeid som er proposjonal med ytelsen til en GPU fører sjelden til at man oppnår optimal ytelse på GPUen, noe som delvis skyldes ytelsestap på grunn av tråddivergens. Arbeidslaster som er avhengige av pseudo-tilfeldige tall er spesielt vanskelige, fordi tråddivergensen kan være tilsynelatende tilfeldig, noe som gjør det vanskelig å redusere tråddivergensen.

I denne oppgaven presenterer vi en metode for å omorganisere pseudo-tilfeldige arbeidslaster før de kjører på GPUen, med mål om å redusere tråddivergens. Våre eksperimenter viser at metoden medfører en ytelsesøkning på opptil 1.45 for en ekte applikasjon. Vi viser også at metoden kan være raskere selv om tiden som brukes til å omorganisere lasten er tatt med. I tillegg til dette utvikler vi en metode for å estimere reduksjonen i kjøretid, som kan brukes for å bestemme om omorganiseringen skal gjøres eller ikke.

En graf-basert metode for balansering av oppgaver blir også presentert. Denne metoden velger den optimale rekkefølgen av oppgaver i 96% av tilfellene som ble testet. Denne oppgavegrafen fungerer også som en modell for ytelsen til GPUen, og

estimatene blir brukt av en lastbalanserer for å dele arbeidslasten mellom prosessoren (CPUen) og GPUen.

Acknowledgements

I would like to thank my supervisor **Anne Cathrine Elster** and my co-supervisor **Jan Christian Meyer** for their support and guidance throughout this thesis. Our discussions have been very interesting.

To my parents, **Kristin** and **Geir**, I am eternally grateful for your emotional and unconditional support.

I would also like to thank my girlfriend and family, **Silje**, **Marius**, **Vilde**, **Øystein**, **Marie**, **Ola**, **Kjersti** and **Milo**. I am very lucky to have you as my girlfriend, family and step dog.

A special thanks to my grandparents, **Oddrun**, **Steinar**, **Eli** and **Sverre**. I am very lucky to have you as my grandparents.

Last, but not in the slightest least, I would like to express my sincere appreciation to my friend **Bjarne Grimstad**. Your friendship, guidance and knowledge have had an immensely positive impact on my life. I am truly grateful.

Table of Contents

List of Tables

List of Figures

List of Source Code

List of Abbreviations and Nomenclature

Abbreviations

CPU	=	Central processing unit
GPU	=	Graphics processing unit
FLOPS	=	Floating point operations per second
KiB	=	Kibibyte (1024B)
MiB	=	Mibibyte (1024KiB)
GiB	=	Gibibyte (1024MiB)
PRNG	=	Pseudo-random number generator
WF	=	Warp-friendly
NWF	=	Non warp-friendly
DAG	=	Directed acyclic graph
WEE	=	Warp execution efficiency
PCI	=	Peripheral Component Interconnect

Nomenclature

l	=	Time series length
p	=	p-parameter of the geometric distribution
n	=	Number of resamplings
Warp-friendly	=	When stationary bootstrap is run using reorganized workload
Non warp-friendly	=	When stationary bootstrap is run without reorganization

Chapter 1

Introduction

Heterogeneous systems composed of multi-core central processing units (CPU) and accelerators such as graphics processing units (GPU) are becoming increasingly popular for data-parallel applications. GPUs were originally produced for accelerating demanding graphical tasks, but is now being used for general purpose computing because of the high performance it offers relative to the CPU. Programming toolkits such as CUDA[1] [NVI17c] have been developed to make it easier to develop and run programs on GPUs. However, achieving the best performance on a heterogeneous system requires the workload to be partitioned such that the devices get a workload relative to their performance. This remains a main challenge.

In this thesis, we will look at how data-parallel applications can be partitioned across the CPU and GPU. A system for determining the optimal partitioning of a specific application with irregular workloads (workloads where the amount of work per work unit varies) will be implemented. Additionally, a novel method for balancing

[1] CUDA used to be an abbreviation for *Compute Unified Device Architecture*, but was later renamed to just CUDA.

the workload internally in the GPU is also presented, implemented and studied.

Load balancing is the process of determining the portions of the total workload to run on each device. Typically, it is desired to minimize the total execution time of the program. To achieve this, each device must be assigned a workload such that the execution time of all devices are equal, and finish computing at the same time. Failing to find a good partition means that the devices that finish first must wait for the slowest to finish, effectively wasting the computational resources of the devices that must wait.

1.1 Motivation

In subsea oil production, there are typically a number of wells where oil and gas is extracted from. These wells are often connected to a single large pipe that the oil and gas are transported through to the oil platform or production ship at the *top-side*. The result of this is that only the cumulative production of all wells is known by measuring the rate of flow at the top-side, and the production of each individual well is unknown. For optimizing the valve settings with the objective of maximizing the oil and gas rates, it is very useful to know the production of each well.

One of the commonly used methods for determining the production of a single well is to fully close the valves on the well. Then, the delta in cumulative production is measured at the top-side. Because of noise and oscillations in the flow, the well must remain closed for a period of time until the true production of the well can be known with a certain confidence. Each second the well is closed leads to lower total production and therefore less income for the oil production company. It is therefore important to be able to determine the production of the well as fast as possible, so that the valve choke can be reopened and the full production potential regained.

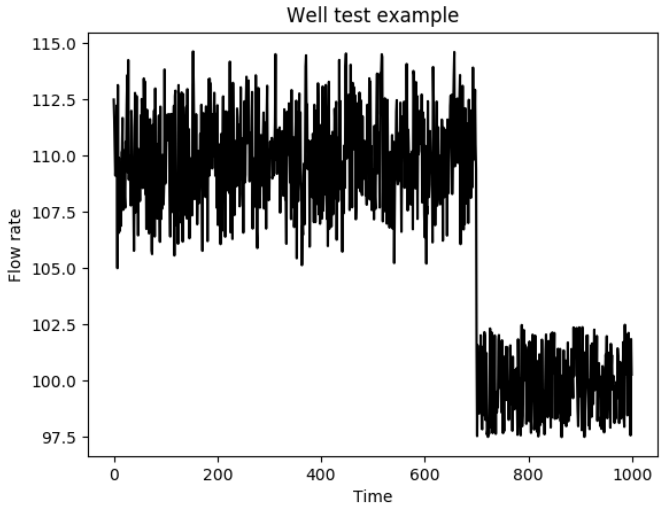


Figure 1.1: Example of flow rate before and after closing a well.

Figure 1.1 shows a fictional example of a drop in the total flow rate when one well is closed at t=700.

A method called stationary bootstrap can be applied to give an estimate of the mean production of the well. The time series of the rate of flow is resampled many times and the arithmetic mean of each resampling can be used to build a histogram of means, which in turn can be used to compute a confidence interval for the true mean of the time series. It is imperative for the method to be fast, so that the well can be reopened as soon as possible. In this project we will look at methods for achieving this.

1.2 Thesis scope

The scope of this thesis is to develop a method for partitioning irregular pseudo-random workloads on heterogeneous systems using GPUs as accelerators. Workload partitioning is traditionally interpreted as determining a partition of the workload that minimizes execution time when run on a heterogeneous system. In this thesis, we will include the partitioning of the workload internally in the GPU, that is, between the cores of the GPU, as part of the study.

The scope is thus divided into two main parts: Developing a method for partitioning the total workload between the CPU and GPU, and developing a method for partitioning the workload internally in the GPU. The methods will be compared to traditional methods found in the literature.

We focus our attention to applications that execute a single data-parallel kernel with *pseudo-random irregular workloads*. Pseudo-random irregular workloads are applications where the workload of each data point is partially determined by the output of a pseudo-random number generator (PRNG).

1.2.1 Assumptions

- A single GPU will be used for all test cases. Both the approach taken for partitioning the workload between the CPU and GPU, and the approach for the workload partitioning internally in the GPU should be valid for multi-GPU setups as well. Multi-GPU setups is therefore considered to be out of scope for this thesis.

- The majority of the workload per data point occurs in a loop, and the number of iterations of the loop must be partially determined by the output of a PRNG.

1.2.2 Research question

> Determine the feasibility of reorganizing the workload to increase the performance of GPUs on irregular pseudo-random workloads, and if so, for which cases one can expect decreased execution time.

1.3 Structure of the thesis

The thesis is organized into nine chapters. The first chapter gives a brief introduction to heterogeneous systems and load balancing, motivates the work done in this thesis, and defines the scope of the thesis. Chapter 2 aims to provide the reader with the theoretical background that is necessary to follow the rest of the thesis, along with a review of existing load balancing approaches that are found in the literature. In Chapter 3, we describe the methods we have developed for workload partitioning. Chapter 4 gives a detailed description of the implementation of our methods, followed by Chapter 5 where the experiments and metrics we use are described. In Chapter 6, the results of the experiments are presented, and then discussed in Chapter 7. Finally, we present our conclusions in Chapter 8, and provide recommendations for further research in Chapter 9.

Chapter 2

Background

The purpose of this chapter is to provide an overview of the key concepts and technologies that is used in this thesis. To fully understand the rest of the thesis, it is necessary to understand the concepts described here.

We start with a description of pseudo-random number generators, followed by a brief overview of the GPU programming toolkit called CUDA. An introduction to branch divergence on GPUs is then given, and followed by a short description of regular and irregular workloads. Next, the application that is used for evaluating our methods, the stationary bootstrap, is introduced. We then provide an overview of splines, which are used for building regression models. Finally, an overview of existing workload partitioning approaches that we found interesting and relevant to this thesis is presented.

2.1 Pseudo-random number generators

Generation of random numbers in computers is often done using pseudo-random number generators (PRNGs). The "pseudo" in the name stems from the fact that the sequence of generated numbers is not really random, but designed so that it will seem random to any external observers [Mon94].

PRNGs consist of an internal state, a state transition function and an output function [Mon94]. When a random number is to be generated, the internal state of the generator is computed by the state transition function, with the *current state* as input. The output function is then used, with the *new state* as input. The output of this function is the random number.

$$x_{n+1} = (ax_n + b) \mathrm{mod} m, 0 \leq x_0 < m, n \geq 0 \tag{2.1}$$

A simple example of a random number generator is the linear recursive congruential method [Hef86]. Here, the internal state is the variable x, the state transition function is the congruence in Equation (2.1), and the output function is the identity function.

A very important property of PRNGs is that both the state transfer and output functions are deterministic. Therefore, given an initial state x_0, the entire sequence of numbers generated is predictable. This initial state is called the *seed*. For example, using Equation (2.1) with parameters $a = 73$, $b = 87$, $m = 93$, and setting the initial state (seeding the generator) $x_0 = 59$, these are the first ten pseudo-random numbers: 23, 92, 14, 86, 41, 11, 53, 50, 17, 26. There is no more input to the PRNG than the parameters and the seed, so the exact same sequence will be generated for those same initial conditions.

A consequence of this determinism is that, while providing sufficient random-

ness for many use cases, the use of PRNGs make experiments reproducible. As long as the initial seed is known, any experiment that depend on a PRNG for randomness will be reproducible, assuming the rest of the experiment is deterministic.

2.2 CUDA

CUDA is NVIDIA's general purpose programming model for GPUs. It was introduced in 2006 [NVI17c], and features a compiler, library and runtime to enable developers to create programs to be executed on NVIDIA GPUs. One of the programming languages used by CUDA is C++, with a few extensions. Programs written for CUDA are first compiled using NVIDIAs compiler, nvcc, which separates the GPU specific code from the rest and compiles it. The CPU specific code is compiled using a standard C++ compiler, and the results combined into the same binary.

CUDA programs typically mix normal serial CPU code with code that runs on the GPU. Special directives are used for marking functions that can be run on the GPU or CPU, or both. These functions are called kernels. Running a kernel is called launching it, and a special syntax element has been added to the normal function call syntax to enable some additional configuration parameters.

2.2.1 GPU topology

NVIDIA GPUs consist of a number of *streaming multiprocessors* (SM). In each SM, there is a number of CUDA cores. The amount of CUDA cores and SMs differ between GPU generations, and also between GPUs of the same generation. For instance, the Pascal generation NVIDIA GTX 1080 Ti has 128 CUDA cores in each of the 28 SMs, for a total of 3584 CUDA cores.

Each streaming multiprocessor on the GPU has one instruction unit per 32 CUDA

cores. The CUDA cores are simple and relatively slow compared to a CPU core. They feature no out-of-order or speculative execution, and there is no branch prediction.

2.2.2 Grid, thread block and warp

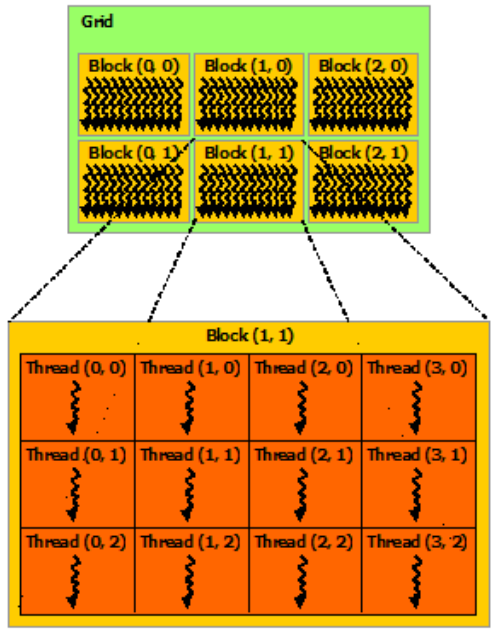


Figure 2.1: CUDA grid, thread block and threads [NVI17c].

When a CUDA kernel is launched, a specified number of threads are spawned. Each thread executes the same kernel with the same parameters, except for a few implicit parameters that can be used to calculate the threads unique id. The threads are organized into thread blocks (sometimes just called blocks), which can be 1, 2 or 3-dimensional. The blocks are organized into a grid of blocks, which can also be 1, 2 or 3-dimensional. Figure 2.1 shows an example of a 2-dimensional block of threads

in a 2-dimensional grid of blocks.

Each block is assigned to a single SM, and each SM can have multiple blocks assigned to it. When executing, groups of 32 threads from the same block are executed simultaneously. These groups of 32 threads are called warps. Because there is only one instruction unit per 32 CUDA cores, threads that belong to the same warp must always execute the same instruction at any time.

2.2.3 Memory hierarchy

There are several types of memories in CUDA GPUs. The fastest, and smallest, are located on the GPU chip, while the largest and slowest are located off the chip.

Registers

Registers are the fastest type of memory in CUDA. They are used for thread local storage, and are located in each SM. There are typically thousands of registers per SM.

Shared memory

Each SM has a fast memory type that is called shared memory. Shared memory is local to each thread block, and the amount of shared memory is configurable. Shared memory can typically store on the order of tens of kibibytes.

L1 cache

Similarly to shared memory, there is an L1 cache in each SM. The amount is configurable per kernel, so the amount of L1 cache can be increased at the expense of less shared memory, and vice versa.

L2 cache

L2 is the largest memory that is located on chip, at a few mibibytes of size. It is used for caching accesses to local or global memory.

Texture memory

Texture memory is located off chip in global memory, but has a special cache on the chip that optimizes for spatial locality accesses.

Local memory

Local memory is also located off chip in global memory, and is used for storing thread local data that does not fit in the registers of a thread.

Constant memory

Constant memory is used for read only data, and has a special cache associated with it in each SM. Data can only be written to constant memory from the CPU. It is located off chip in global memory.

Global memory

Global memory is the largest of the memory types in CUDA GPUs. It can be both read from and written to, but is very slow compared to on chip memories. The size of the global memory may be tens of gibibytes.

2.3 Branch divergence on GPUs

GPUs implement the single instruction, multiple threads (SIMT) architecture. In this architecture, there is only one instruction unit per n cores ($n = 32$ in newer

CUDA cards). Statements that may cause threads to execute different control flows is therefore an issue, as the threads no longer will share instruction stream. This is called branch divergence.

```
1 __global__ branch_divergence_if() {
2     int id = blockDim.x * blockIdx.y + blockIdx.x;
3     if (id % 2) {
4         // Do one thing
5     } else {
6         // Do something else
7     }
8 }
```

Listing 2.1: Branch divergence example 1.

In listing 2.1, a CUDA kernel which exhibits branch divergence is shown. Threads with odd ids will enter the true part of the if statement, while threads with an even id will enter the false part. Because of the restriction of only one instruction per warp at a time, the GPU may solve this by first executing the true part with only the relevant threads marked active, and then doing the opposite for the false part. The result of this is that the total execution time becomes the sum of the execution times in both branches, as opposed to being the maximum of the two had there been two (or more) instruction units.

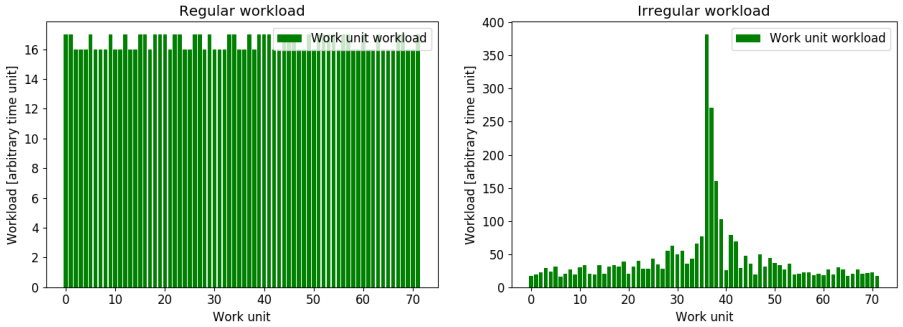
```
1 __global__ branch_divergence_for() {
2     int id = blockDim.x * blockIdx.y + blockIdx.x;
3     for (int i = 0; i < id; i++) {
4         // Do useful work
5     }
6 }
```

Listing 2.2: Branch divergence example 2.

Listing 2.2 shows another type of branch divergence, which is when the number of iterations in a loop is not equal for all threads. In this example the loop condition is what causes this, but it could just as well had been something else, like the amount the induction variable i is incremented each iteration. In this particular example, the id's of the threads of a warp will be strictly increasing by one from the lowest id to the highest. The number of iterations of the loop will match the id of a thread, so for each iteration, one less thread will be executing, and waiting instead. The end results is that approximately half of the "thread-time" will have been spent waiting for the other threads to finish (assuming $0 \leq id \leq 31$ for all id's).

2.4 Irregular workloads

Parallel workloads can be divided into regular and irregular workloads [She+13]. Regular workloads are applications where all the work units are approximately of the same size, *i.e.* they take approximately the same amount of time to finish on the same hardware. Irregular workloads does not have this property, and the amount of work per work unit may vary by a lot.



(a) Regular workload

(b) Irregular workload

Figure 2.2: Comparison of regular and irregular workloads

Figure 2.2a shows an example of an application with a regular workload. Such applications may perform well without modifications when run on the GPU. An example of a regular workload is (dense) matrix multiplication.

Figure 2.2b shows an example of an application with an irregular workload. If run on a GPU without modifications, one can expect to see a large fraction of inactive threads to active threads, due to some threads being assigned a very time consuming workload compared to the other threads in the same warp. An example of an irregular workload is ray tracing [She+13], where each ray represents a work unit. The work unit size per ray may differ because of a variable time step, which may be dependent on simulation conditions.

2.5 Bootstrap

Statistical bootstrap [Efr79] is a method of estimating properties of the sampling distribution of a random sample. Such properties may be the mean, variance and median or higher order moments. In this work, the arithmetic mean is of particular

interest, so the method will be described with a focus on it, yet the same method applies to the other properties as well.

Given a random sample $X = (x_0, x_1, ...x_m)$, the mean can be estimated using the bootstrap method as follows: A new sample X_1 is generated by sampling with replacement from X $m+1$ times, such that X_1 is of the same size as X. The statistic of interest is then computed from the new random sample X_1. This process is repeated n times, for sufficiently large n. The results from those computations are n means, which can be grouped into bins to form a histogram. This histogram is the resulting estimate of the mean, and can be used to compute such values as the confidence interval of the mean.

2.5.1 The stationary bootstrap

The stationary bootstrap is a special case of the bootstrap, introduced by D. Politis and J. Romano [NP94]. It attempts to address the issue that regular bootstrap fails to capture the dependency between adjacent samples in the data, if there is such a dependency. The stationary bootstrap is applicable to stationary, weakly dependent time series [ECO]. The stationary property means that the sampling distribution is static, and the weakly dependent property means that the correlation between two samples tends to 0 sufficiently quickly as the distance between them (in time) grows.

The stationary bootstrap differs from the regular bootstrap in how it resamples the data. To capture the dependence between samples, it does not draw samples one-by-one, but selects all samples that fall within some window. The start of this window is drawn from a uniform distribution, and the size of the window is drawn from a geometric distribution (with parameter p). If the start of the window, plus the size of it, should exceed the size of the time series, the windows wraps around and continues from the beginning of the time series. This process is continued until

the new time series is the same size as the original. If a window is selected such that the size of the new time series will be larger than the original time series, the size of the window is shrunk until this is no longer the case.

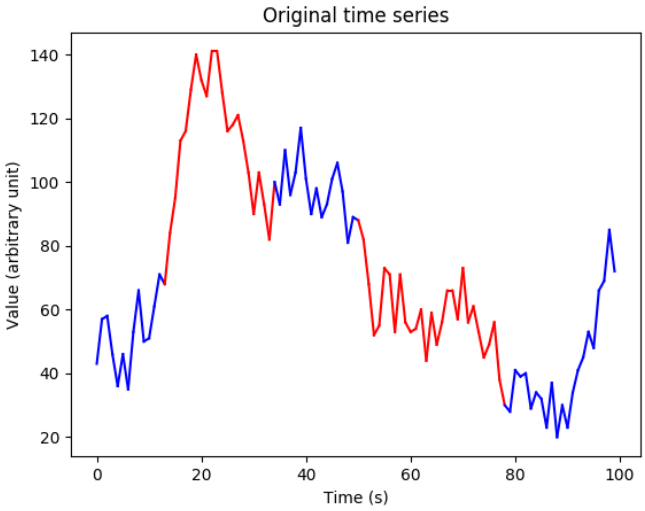


Figure 2.3: Original time series.

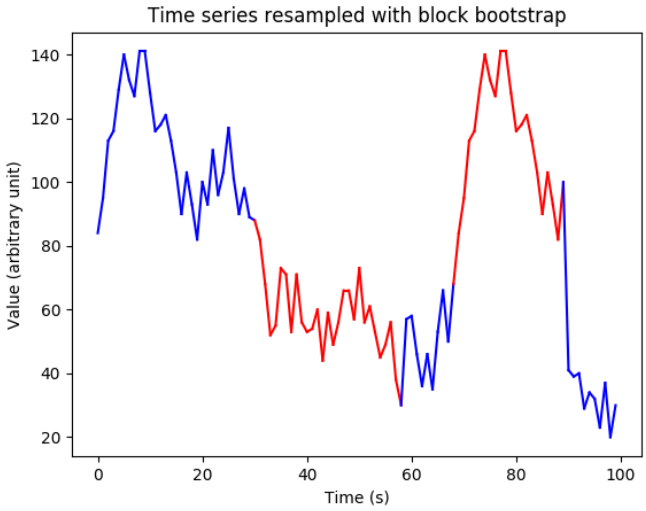


Figure 2.4: One of the possible resampled time series.

Figure 2.3 shows an example of a time series. The red parts are two examples of windows that were selected for "transfer" to the new, resampled time series. In Figure 2.4, the same windows of samples have also been marked red, to make them easier to identify. Note that the entire resampled time series is made up of such windows, but only two of those are marked in red. The first red part from the original time series can be found near the end in the resampled time series, where it has also been marked red. The other red part, which can be found in the original time series at $50 < \text{Time} < 80$, appears in the resampled time series at $30 < \text{Time} < 60$.

2.6 Splines

Splines are mathematical functions defined by piecewise continuous polynomials. They have a high degree of smoothness, meaning that, for all x at the interval bounds of the piecewise functions f_i, the sum

$$\sum_{i=0}^{n-1} (f_i'(x) - f_{i+1}'(x))^2 \tag{2.2}$$

is minimized.

Splines can be defined for multivariate functions $f : \mathbb{R}^m \to \mathbb{R}^n$, but in this project only the univariate spline with a single output is needed ($f : \mathbb{R} \to \mathbb{R}$), so this explanation will focus on the univariate spline. In particular the basis spline (B-spline), which defines splines using a recurrence relation, will be presented here.

A B-spline is defined in terms of a vector of basis functions, with a coefficient for each basis function which adjusts the contribution from each basis function. Mathematically a spline f can be defined as follows [GS16]:

$$f(x) = \sum_{i=0}^{n} c_i b_i(x), \tag{2.3}$$

where b is the vector of basis functions, and c are the coefficients. Two of the parameters to the basis functions have been skipped here; The interval it is defined on (t), and its degree p. All basis functions, except those of degree $p = 0$, are defined in terms of two other basis functions of degree one less than itself. The basis functions b can thus be defined by a recurrence relation (p parameter included, t being implicit):

$$b_{i,p} = \frac{x - t_i}{t_{i+p} - t_i} b_{i,p-1}(x) + \frac{t_{i+1+p} - x}{t_{i+1+p} - t_{i+1}} b_{i+1,p-1}(x) \tag{2.4}$$

with the special case for basis functions of degree $p = 0$:

$$b_{i,0} = \begin{cases} 1, & \text{if } t_i \leq x < t_{i+1}. \\ 0, & \text{otherwise.} \end{cases} \tag{2.5}$$

The vector t is often called the knot vector of the spline. Each basis function of degree $p = 0$ is 1 inside one such interval, and 0 everywhere else. This *local support* property of the $p = 0$ degree basis function means that the basis function that is defined in terms of the zero degree basis function also has local support, but on one additional interval. So, for a $p = 3$ degree basis function, it is defined in terms of $p + 1 = 4$ zero degree basis functions, and by extension $p + 1 = 4$ intervals. Spline implementations on computers can exploit this property to only evaluate the basis functions that contribute to the value of the spline at each point, for example by using de Boor's algorithm [Boo72]. Because the basis functions are defined as a convex combination, the evaluation of a spline is numerically stable.

2.6.1 The penalized spline (P-spline)

There are several ways of fitting a spline to a data set, one of which is to minimize Equation (2.6) to get an interpolating spline.

$$\sum_{i=0}^{n} (y_i - f(x_i))^2 \tag{2.6}$$

The coefficients and knot vector of interpolating splines are chosen so as to minimize Equation (2.6), while retaining the smoothness property of the spline. For some datasets, minimizing Equation (2.6) may lead to a phenomenon known as overfitting, where the spline ends up fitting noise instead of capturing the informa-

tion in the data. Interpolating splines may therefore have suboptimal performance on regression problems.

Penalized splines (P-splines) may reduce the problem of overfitting. When building a P-spline, a term is added to Equation (2.6) (resulting in Equation (2.7)) to penalize large second derivatives of the spline.

$$\sum_{i=0}^{n}(y_i - f(x_i))^2 + \lambda \int_{x_i}^{x_{i+1}} f''(x)^2 \mathrm{d}x \tag{2.7}$$

The parameter $\lambda \in [0, \infty)$ controls the amount to penalize large $f''(x)$. A $\lambda = 0$ yields an interpolating spline, while $\lambda = \infty$ will yield behaviour equal to linear least squares regression.

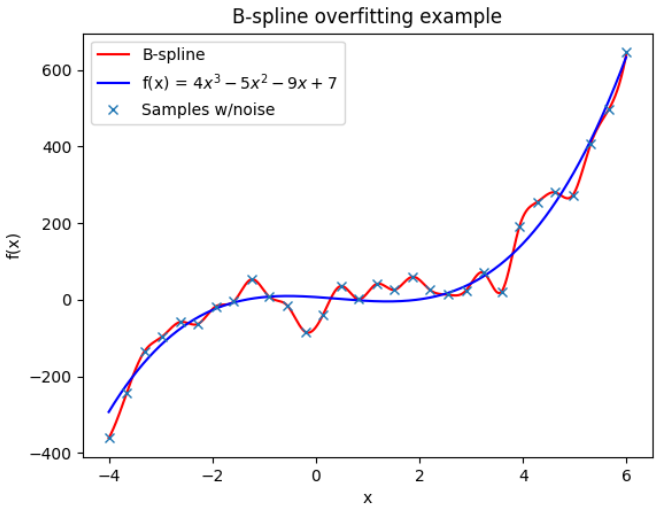


Figure 2.5: Example of a B-spline overfitting a data set.

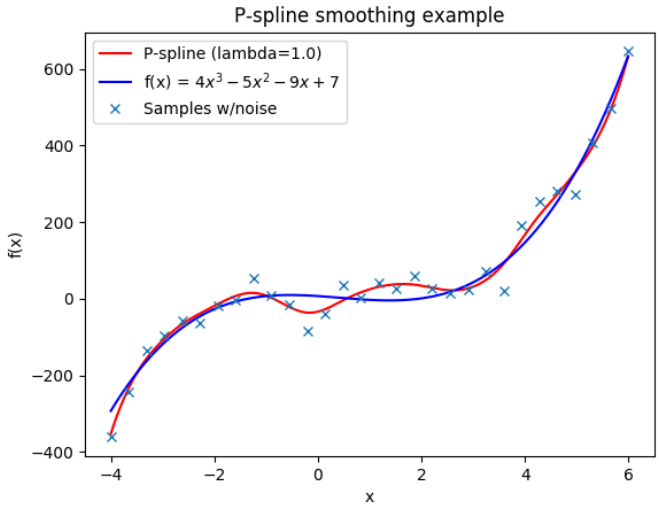


Figure 2.6: Fitting the same data set with a penalized spline.

Figures 2.5 and 2.6 demonstrate how a B-spline may overfit a data set, and how a penalized spline may be used to capture the underlying information in the data better. In this case, the underlying function is a third degree function with Gaussian noise ($\mu = 0, \sigma = 70$) applied to the 30 samples that were used for building the splines. $\lambda = 1.0$ were used for building the P-spline. At $x = 0$ we see the worst case of overfitting, the actual value of the function is $f(0) = 7$, but due to the noisy sample, the B-spline predicts a value of around -100. The P-spline has a far better prediction, at around -40. Fitting the sample at $x = 0$ would require a large second derivative at the local minimum, which would add a large penalty to the spline, and is therefore avoided.

2.7 Related work

In the literature, there is a division of load balancing methods into static and dynamic load balancing. Static load balancing is when the balance is determined once - typically based on performance models - and is not changed during computation. Dynamic load balancing is when the balance may change during execution, for example as a consequence of observing the relative performance of the devices. The types of load balancers are further divided into balancing data-parallel and task-parallel applications. We focus our study on data-parallel load balancers as they fit our scope, but include an interesting approach to task-parallel load balancing that may be used for data-parallel applications with minor changes, and another approach to task-parallel load balancing using a task graph.

2.7.1 Static load balancing approaches

In static load balancing approaches, a performance model of the devices is often used to predict their relative performance [She+13; She+16; FE16], and then selecting a partition based on the estimated throughput of each device.

Workload reshaping

[She+13] introduced an approach to static load balancing of *irregular* workloads that is applicable to applications with kernels consisting of one or more loops. The workload distribution is sampled, where the output from the samples is the number of iterations of these loops. They then apply a technique they call *workload reshaping*, in which they build a model of the workload distribution using the samples, and sort the workloads by the number of iterations of the loops. The distribution is then analyzed to determine the shape of it, *e.g.* find flat areas and areas with peaks. Using

execution times from previous runs, a partition point that minimizes the difference between the CPU and GPU execution times is determined. Flat areas are scheduled to run on the GPU, and areas with peaks are scheduled to run on the CPU. The reasoning for this scheduling is that flat areas is thought to have less branch divergence.

Machine learning based models

[FE16] proposed an approach to static load balancing of *regular* workloads based on machine learning. In their method, a device is selected to be the main device. Then, for each additional device, a model is built using machine learning that takes hardware counters from the main device as input, and the throughput of another device as output. The models are trained in an offline phase by running the program with many different parameters. At run time, a small input is run on the main device, and the hardware counters used as input for all the models. The output is then used as an estimate for the execution time of each additional device, and the workload is distributed so that each device gets a workload that is proportional to their estimated throughput.

Linear regression models

[She+16] proposes two approaches for modelling the throughput of the devices. The first uses offline profiling, where the throughput of each device, and the transfer time from CPU memory to GPU memory, are recorded for different input sizes. The throughputs are then used to build linear regression models for each device and the data transfer. They divide the problem range into several sets, where each set fits into a different physical memory type. This means that they build one linear regression model for problem sizes that fit in L1 cache memory, one regression model for problems that fit in L2 cache memory, etc. For online profiling, they run a very small

input on each device and record the throughputs and data transfer time. Similarly to [She+13], they apply workload reshaping to facilitate division of the workload into flat and peak areas. In this method, however, the programmer is required to manually model the shape of the workload.

2.7.2 Dynamic load balancing approaches

Increasing chunk size

[Boy+13] developed a dynamic load balancer that works by initially assigning a small workload, called chunks, to each device. As the devices finish computing the initial chunks, a new, larger chunk is assigned to them. When all devices have finished executing a set number of chunks, γ, the throughputs of the previous runs are used to partition the remaining work into n chunks, where n is the number of devices.

Persistent kernel

[Che+10] offers an interesting approach to balance task-parallel applications. They launch one persistent kernel with B thread blocks for each GPU, where B can be as large as the maximum number of concurrently executing thread blocks on each device. One or more task queues are created, to which the CPU thread submits tasks. One thread in each thread block polls the queue(s) for a task to execute. When a task is received by the thread, the thread block executes the task. This process continues until a special HALT task is submitted to the queue(s), causing the kernel(s) to halt execution.

We argue that this approach can be used for balancing data-parallel applications by splitting the input into many small tasks, and submitting these to the task queue for processing by the GPU(s).

Task graph

StarPU is a runtime system introduced by [Aug11]. Tasks are submitted by the application to the StarPU scheduler, which selects the next task to run and on which device, based on a scheduling policy. Each task has a *codelet*, which contains an implementation of the functionality of the task for one or more devices. When a task is selected for execution, the scheduler uses the estimated execution time of running the codelet on each device to determine the device the codelet should be run on.

The tasks in StarPU may have dependencies on other tasks. StarPU automatically builds a task graph from these dependencies, and handles transfer of the input and output data from and to each device. Special functions called *filters* can be used to partition the data into *blocks*. A task is generated for each block of data, and the data is combined to the final result when all the tasks have finished executing.

Chapter 3

Methodology

The purpose of this chapter is to provide a description of the methods used for partitioning the workload both between the CPU and GPU, and internally in the GPU.

Throughout this chapter and the following chapters, the variables l, p and n will be used extensively. Where not specified otherwise, they are meant to denote the length of the time series, the p-parameter of the geometric distribution used in the stationary bootstrap, and the number of resamplings to perform, respectively. n_{cpu} and n_{gpu} are used to denote the number of resamplings assigned to the CPU and GPU, respectively.

The structure of the chapter is as follows: First, the method used for modelling the execution time of different processes is described. Then, a presentation of the methodology for reducing branch divergence in the GPU kernels is provided, along with a short review of the irregularity of the workload in stationary bootstrap. A method for automatically determining the optimal execution path of the GPU process is then presented, and finally our method for workload partitioning across the

CPU and GPU is described.

3.1 Performance modelling

The execution time of different processes are modelled by a one dimensional P-spline of degree 3 with $\lambda = 0.1$, or, in the case of the model of the CPU execution time, $\lambda = 0.001$. Every time the process is executed, the execution time is recorded and used to update the spline with the new data. The execution time of most processes depend on 3 variables: l, p and n. The version of SPLINTER [Gri+15] we use require the samples to form a regular grid. This means that building a spline with multiple input dimensions require a number of samples that is exponential in the number of dimensions, and can therefore quickly become very large. To avoid having to sample the input space in many points, the dimensionality has been reduced to 1 by constructing a new spline for each pair of l and p.

A P-spline of degree 3 requires at least 4 unique samples to be built. For inputs where the number of unique samples is less than this number, the estimate is reported as 0. The same is also true for n that lie outside of the estimator support, which limits the use of the estimator to interpolation, and avoids extrapolation.

At program exit the splines are saved to disk. When estimates for some pair of l and p is required, the relevant splines are loaded and used.

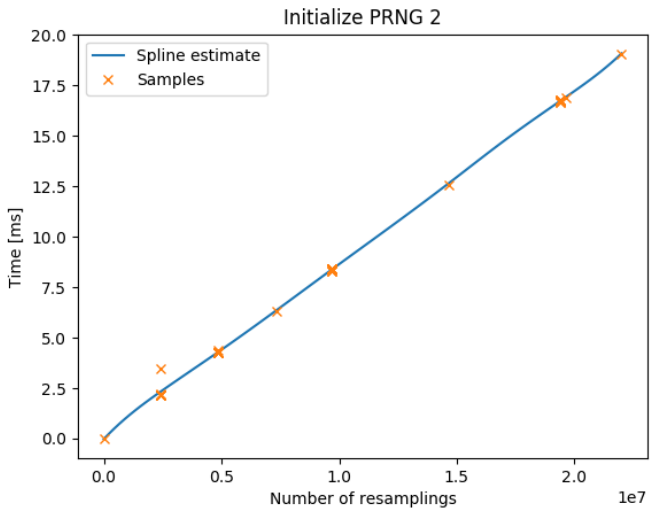


Figure 3.1: The estimator for the operation *Initialize PRNG 2* after 40 executions.

Figure 3.1 shows the recorded data points and the spline estimate for a range of values of n. We can see that the penalty of the P-spline ensures that the spline does not overfit by interpolating the sample at $n \approx 2,500,000$.

3.2 Partitioning of workload in the GPU

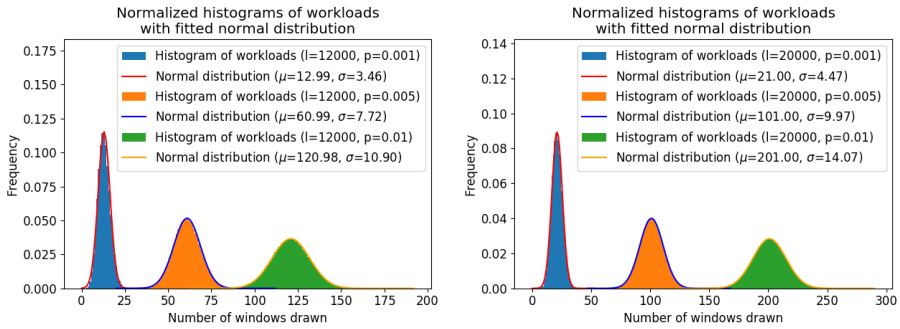
Our approach to partition the workload on the GPU aims to reduce the branch divergence of the kernels, and thereby achieve a reduction in the execution time of the kernel by spending less time in divergent branches.

We start by providing a short analysis of the irregularity of the workload in stationary bootstrap, which is followed by a description of the method we use to reduce branch divergence.

3.2.1 Irregular workload in stationary bootstrap

Distribution of workload

The stationary bootstrap uses a geometric distribution for selecting the sizes of the windows. The geometric distribution has a parameter, $p \in (0,1]$. The expected value of the distribution is $\mu_{\text{geometric distribution}} = \frac{1}{p}$. For a time series of size l, the expected number of windows to be drawn is therefore $\mu_{\#\text{windows}} = \frac{l}{\mu_{\text{geometric distribution}}} = lp$. In this thesis, we use the term *number of windows* or *number of iterations* extensively to denote the number of windows that must be selected to fully resample a time series, which is equivalent to the number of iterations of the loop that performs the resampling. This metric is our measure of the workload of each resampling.



(a) Workload distribution for $l = 12000$.

(b) Workload distribution for $l = 20000$.

Figure 3.2: Workload distribution for $l \in [12000, 20000]$ and $p \in \{0.001, 0.005, 0.01\}$.

Figure 3.2a shows the frequency of each iteration count / number of windows for $l = 12000$ and three different values of p. Figure 3.2b shows the distribution of the same ps for $l = 20000$. We can see that as p is increased, the expected number of windows increases, but so does the variation. The workload also increases as l is

increased. The workloads appear to be approximately normally distributed for all sets of parameters, with a worse fit for when the expected value is low.

Coefficient of variation

The coefficient of variation is a metric for showing the relative variation of a population with respect to the expected value. It is defined as

$$c_v = \frac{\sigma}{\mu} \tag{3.1}$$

The metric proves to be a very useful metric for determining the speedup that is achievable by reorganizing the workload. In the figures below we show how the coefficient of variation varies for the stationary bootstrap as l or p is changed.

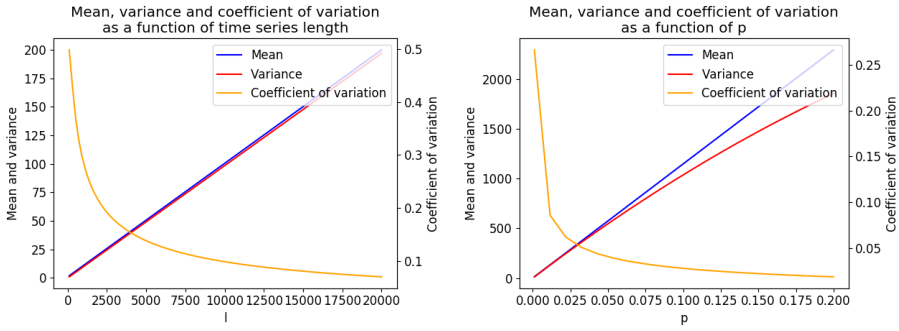
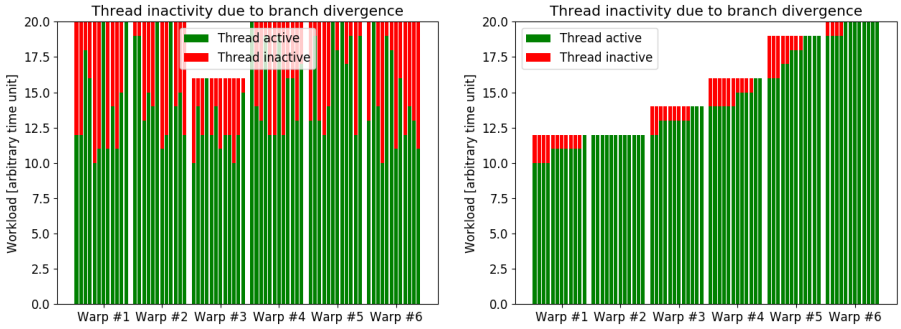


Figure 3.3: Coefficient of variation, mean and variance as a function of the time series length and p, separately.

We can see that the mean and variance of the workload distribution grows as l or p is increased, while the coefficient of variation decreases.

3.2.2 Reorganization of workload

In our implementation of the stationary bootstrap on the GPU, each thread performs one resampling of the time series. There is a separate PRNG for each thread, and each PRNG is seeded with a randomly drawn seed. Taking inspiration from the workload reshaping of [She+16], we apply a sorting of the PRNG seeds based on the resulting main loop iteration count. Figure 3.4a shows a fictional example of the workloads per work unit, divided into warps of size 12 (for illustrative purposes). The green bars are where a thread is active, and the red bars are where a thread is marked inactive, thus wasting resources. Because of branch divergence, the longest green bar determines the execution time of each warp. The bigger the variation in work unit size, the more thread-time is wasted by threads waiting for the thread with the largest work unit to finish. Figure 3.4b shows the exact same workload as figure 3.4a, but where the work units are sorted based on their size. The amount of wasted thread-time has been reduced from 338 to 62, or from 24% of the total execution time to 6% of the total execution time. There is even a warp where no thread is marked inactive at all.



(a) Before reorganization.

(b) After reorganization.

Figure 3.4: Thread activity before and after reorganizing the workload by the amount of work per work unit.

PRNG seed as predictor for iteration count

In some problems, such as the stationary bootstrap described later, the seed of the PRNG is a perfect predictor for the number of iterations of the main loop in the kernel. The PRNG that implements the geometric distribution is a deterministic function of the seed, which makes the sequence of window sizes deterministic, and therefore the number of iterations needed by the loop to reach the stopping condition. When saying that a seed uniquely determines the size of a work unit, it is implied that l and p are not varied.

3.2.3 Generation of seed to iteration count mapping

To be able to sort the seeds based on their respective workloads, a mapping from all possible PRNG seeds to the iteration counts they result in is needed. Mathematically, there is an infinite number of different seeds to the PRNG, which renders this task impossible. Due to technical constraints, however, there is a finite set of seeds.

On some systems, the seed is represented by a 32-bit integer, which can represent 2^{32} different values. A mapping from 2^{32} different seeds to iteration counts may be represented by an array where the index is the seed, and the value is the resulting iteration count if the PRNG is seeded with that seed. Assuming the values need to be represented by a 32-bit integer, this array would take $2^{32} \times 4\text{B} = 16$ GiB of memory. To be able to load this array into memory for fast access, we have halved the amount of seeds we generate, by not computing the mapping for seeds where the most significant bit is 1. Our mapping is therefore a mapping from 2^{31} seeds to iteration counts, and occupies 8GiB on disk / memory.

Section 3.2.2 notes that the iteration counts depends on the PRNG seed in addition to l and p. Therefore, we precompute and store the mappings each time a new unique pair of l and p is encountered.

3.3 Task graph

The GPU bootstrap process is represented by a set of *operations*. An operation is a function that operates on some input, and has either side effects or outputs, or both. Each operation has some preconditions that must be true before it can execute, and postconditions that must be true after its execution. The postconditions of some operations are the preconditions of other operations, so that each operation has one or more operations that must be executed before it itself can be executed. An operation is statically assigned to either the GPU or the CPU at compile time. The execution time of each operation is modelled using the method presented in Section 3.1.

A task graph of the program is then made by constructing a directed acyclic graph (DAG) of the operations. The nodes in the DAG represents an operation, and an edge from node A to node B represents that the operation represented by node

A must be executed before the operation represented by node B. If a node has more than one ancestor, one and only one of the operations represented by the ancestor nodes must be executed before it. The complete task graph of the program can be seen in figure A.1 in the appendix.

The task graph has a single top node (source) and a single leaf node (sink), where both represent null operations. Executing all operations along a path from the source to the sink results in a valid execution of the program. The possible paths of the task graph can be seen in figures A.2, A.3, A.4 and A.5 in the appendix.

The underlying function of an operation is not necessarily unique. As in the case of the *Initialize PRNG 1* and *Initialize PRNG 2* operations, they are distinct operations but the underlying function is the same. This makes it possible to distinguish between the two in the task graph, which is important as the two have different preconditions. As an example, one of the preconditions of the *Initialize PRNG 1* is that the seeds are reorganized so that the iteration counts are monotonically increasing, which is one of the postconditions of the sorting operations. The *Initialize PRNG 2* does not have this precondition.

Resources may be allocated in an operation, and not released by the same operation. An operation later in the task graph will handle the release of the resource. Memory leaks and segmentation faults are avoided by carefully inspecting the pre- and postconditions of each operation. For example, the precondition of an operation may be that memory has been allocated for a variable, which is a postcondition of an earlier operation.

3.3.1 Optimal execution path

At run time, the task graph is used with the execution time models of each operation to determine the optimal path through the program. Using the estimated time of

each operation, we find the path that is estimated to take the least amount of time to execute.

The task graph is used directly for finding the shortest path. The weight of each edge is defined to be the estimate of the operation at the end of the edge. A shortest path algorithm is then used for finding the *optimal execution path*, and returns it as a linked list of the operations. Because the operations have estimators that depend on the set of inputs to the program (see 3.1), the optimal path through the program depends on these parameters as well. If there exists some point in the parameter space where the optimal path changes, then the program will automatically change execution path to that path, assuming perfect estimators.

It is useful to be able to force execution through one path, for example if we want to compare one execution path to another. To achieve this, the estimator of an operation can be forced to report a large negative number. Because of this, the Bellman-Ford algorithm [Shi54] is used for finding the shortest path, because it works correctly with negative edge weights. Introducing negative edges can lead to negative cycles, which is not handled correctly by the algorithm. This is not an issue, as there are no cycles in the task graph. This knowledge of no (negative) cycles has been exploited to optimize the Bellman-Ford algorithm, by removing the check for negative cycles.

A new and untrained task graph will have operations where no estimators have been built yet. When the program has selected the same path 4 times, the estimators of the operations in that path will be built. The path will then have a non-zero estimate, while the other paths still estimates 0 execution time, except for the operations they have in common with the "trained" path. This will force the program to select one of the paths that has not yet been tested sufficiently to build a model. The same is true if parameters outside the support of the underlying splines are selected.

3.4 CPU and GPU workload partitioning

We treat load balancing between the CPU and GPU as an optimization problem, where the objective is to minimize the total execution time of the program. The execution time is defined to be the maximum of the execution time of the CPU and the GPU. The objective function to be minimized is therefore

$$\max(t_{\text{cpu}}(nf_{\text{cpu}}), t_{\text{gpu}}(nf_{\text{gpu}})) \tag{3.2}$$

where
$t_{\text{cpu}}(x)$ is the estimated execution time of running x resamplings on the CPU,
$t_{\text{gpu}}(x)$ is the estimated execution time of running x resamplings on the GPU,
f_{cpu} is the fraction of the total amount of resamplings to be run on the CPU,
f_{gpu} is the fraction of the total amount of resamplings to be run on the GPU,
n is the total number of resamplings.

The optimization is subject to the constraint

$$f_{\text{cpu}} + f_{\text{gpu}} = 1 \tag{3.3}$$

There are two decision variables in this optimization problem: f_{cpu} and f_{gpu}. Rewriting the constraint in Equation (3.3), we can eliminate one of the decision variables:

$$f_{\text{cpu}} = 1 - f_{\text{gpu}} \tag{3.4}$$

The objective function then becomes

$$\max(t_{\text{cpu}}(nf_{\text{cpu}}), t_{\text{gpu}}(n(1-f_{\text{cpu}}))) \tag{3.5}$$

with the original constraint in Equation (3.3) replaced by

$$0 \leq f_{\text{cpu}} \leq 1 \tag{3.6}$$

For clarity, the optimization problem is presented in its entirety below.

$$\begin{aligned} \min_{f_{\text{cpu}}} \quad & t_{\text{cpu}}(nf_{\text{cpu}}), t_{\text{gpu}}(n(1-f_{\text{cpu}})) \\ \text{s.t.} \quad & 0 \leq f_{\text{cpu}} \leq 1 \end{aligned} \tag{3.7}$$

The objective function is minimized using gradient descent [Rud16]. The gradient descent is started at $f^0_{\text{cpu}} = 0.5$, and in each iteration x, $t_{\text{cpu}}(nf^x_{\text{cpu}})$ and $t_{\text{gpu}}(n(1-f^x_{\text{cpu}}))$ are computed. The derivative of the max function is the derivative of the larger of its arguments, so if $t_{\text{cpu}}(nf^x_{\text{cpu}})$ is larger than $t_{\text{gpu}}(n(1-f^x_{\text{cpu}}))$, f^{x+1}_{cpu} is updated according to Equation (3.8). If $t_{\text{gpu}}(n(1-f^x_{\text{cpu}}))$ is the larger of the two values, f^{x+1}_{cpu} is updated according to Equation (3.9).

$$f^{x+1}_{\text{cpu}} = f^x_{\text{cpu}} - \gamma \frac{\partial t_{\text{cpu}}(nf^x_{\text{cpu}})}{\partial n}, \tag{3.8}$$

$$f^{x+1}_{\text{cpu}} = f^x_{\text{cpu}} - \gamma \frac{\partial t_{\text{gpu}}(n(1-f^x_{\text{cpu}}))}{\partial n}, \tag{3.9}$$

In our experiments, we have found that $\gamma = 0.7$ balances the trade-off between convergence rate and accuracy well. The optimization continues while the number of iteration is less than 100 and Inequality (3.10) is true. In the optimal point, the

left-hand side of Inequality (3.10) is equal to 1.

$$\frac{\min(t_{\text{cpu}}(nf^x_{\text{cpu}}), t_{\text{gpu}}(n(1-f^x_{\text{cpu}})))}{\max(t_{\text{cpu}}(nf^x_{\text{cpu}}), t_{\text{gpu}}(n(1-f^x_{\text{cpu}})))} < 0.99 \tag{3.10}$$

The gradient descent algorithm is started at a single point, rather than multiple, because the execution time models have been experimentally verified to be monotonically increasing, and therefore have no local minima - except the global minimum - that the algorithm can get stuck in. The discontinuity of the derivative of the objective function in the optimal point is not a problem, because the optimization terminates at that point.

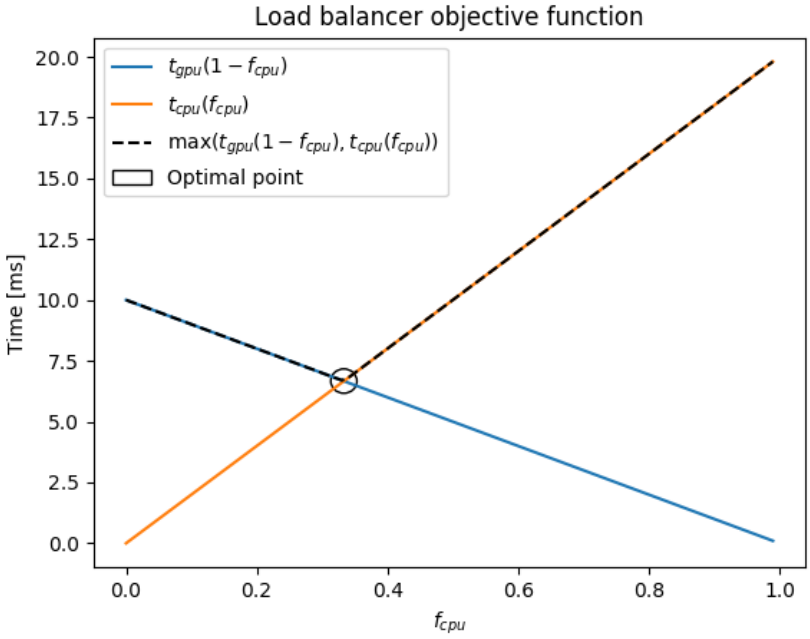


Figure 3.5: Constructed example of load balancer objective function.

Figure 3.5 shows a synthetic example of the behaviour of the t_{cpu} and t_{gpu} functions. The max function of the two functions would be equal to t_{gpu} from $f_{cpu} = 0$ to $f_{cpu} \approx 0.33$, and equal to t_{cpu} from $f_{cpu} \approx 0.33$ to $f_{cpu} = 1$, in this case. The derivative of the max of these functions is therefore equal to the derivative of the larger of the two.

Algorithm 3.1: Load balancer optimization.

```
1  input: int n
2  output: float f_cpu, float f_gpu
3  begin
4      f_cpu ← 0.5
5      i ← 0
6      γ ← 0.7
7      while i < 100 and min(t_cpu(n f_cpu), t_gpu(n(1−f_cpu))) / max(t_cpu(n f_cpu), t_gpu(n(1−f_cpu))) < 0.99 do
8          if t_cpu(n f_cpu) > t_gpu(n(1 − f_cpu)) then
9              f_cpu ← f_cpu − γ ∂t_cpu(n f_cpu)/∂n
10         else
11             f_cpu ← f_cpu − γ ∂t_gpu(n(1−f_cpu))/∂n
12         f_gpu ←1−f_cpu
13         i←i+1
14     return f_cpu, f_gpu
15 end
```

Algorithm 3.1 shows the pseudocode of the load balancer optimization process.

Chapter 4

Implementation

In this chapter we present a detailed description of the program that was implemented to evaluate our methods. Of the sections in this chapter, only Section 4.4 and 4.6.1 are required reading to be able to follow the rest of the thesis. The other sections describe implementation details which are included for the sake of reproducibility.

The chapter starts with an overview of the program, followed by a description of the program flow. Next, an overview of some command line parameters, followed by a note on the statistics that are recorded during execution. The different stationary bootstrap implementations are then presented, along with a general optimization that is applied to all implementations. Finally, a short presentation of the CPU and GPU modules. The description of the GPU module includes a short description of each of the implemented operations of the task graph.

4.1 Tools and hardware

Program code is written in C++11 and CUDA C, and is compiled using nvcc v8.0.61 with GCC version 5.4.0 as the back-end compiler used by nvcc. SPLINTER version 3-0 [Gri+15] is used for constructing, evaluating, saving and loading splines, and nlohmann::json [Loh17] is used for outputting statistics and results to the JSON format, which is easy for humans to parse as well as for machines. Python 3.5.1 is used for analyzing stats and, using matplotlib version 2.0.0, generating figures. All experiments have been run on Ubuntu 16.04 with Linux kernel 4.10.

The hardware configurations used are listed below:

CPU	AMD Ryzen 1800x
GPU	NVIDIA GTX 1080 Ti 11GiB
Motherboard	ASUS Crosshair VI Hero
Memory	16 GiB
Storage	Samsung 750 EVO 232 GiB SSD

Table 4.1: Hardware configuration #1

CPU	AMD Ryzen 1800x
GPU	NVIDIA GTX 480 1.5GiB
Motherboard	ASUS Crosshair VI Hero
Memory	16 GiB
Storage	Samsung 850 EVO 232 GiB SSD

Table 4.2: Hardware configuration #2

CPU	AMD Ryzen 1800x
GPU	NVIDIA GTX 480 1.5GiB
Motherboard	ASUS Crosshair VI Hero
Memory	32 GiB
Storage	Samsung 850 EVO 232 GiB SSD

Table 4.3: Hardware configuration #3

During testing, all machines had two GPUs installed, even though only one was used at a time. With a single GPU installed, the motherboard sets the PCI Express operation mode to x16 on the port that is connected to the GPU [INC17], but with two GPUs, both ports are in x8 operating mode, effectively halving the bandwidth available to the GPU, compared to using a single card.

Hardware configuration 3 is the same machine as hardware configuration 2, but with extra memory installed for testing the effect of increased memory in one experiment.

4.2 Measuring execution time

Execution time is measured using the std::chrono::steady_clock library [Ref17a] of C++11. The clock is guaranteed to be monotonically increasing, which means that even if the system clock is adjusted, the clock will always represent the correct amount of physically elapsed time since some point in time.

For timing portions of the program running on the CPU, the timer is started before execution and stopped when finished. For threaded CPU execution, the timer is started before any threads are launched, and stopped when all threads are finished.

For timing GPU kernels, a call to *cudaDeviceSynchronize* is made before starting the timer. This is done to ensure all operations on the GPU has finished, so that the

timer does not account for other operations in addition to the one we want to time. Similarly, a second call to *cudaDeviceSynchronize* is made before stopping the timer, to force the CUDA driver to wait for the kernel execution to finish. Kernel launches are asynchronous by default, and stopping the timer after launch would not yield the desired result.

4.3 The bootstrap program

A program has been implemented for performing workload partitioning, task graph optimization and running the stationary bootstrap on both the CPU and GPU at the same time. The program contains all the functionality required for generating the mapping from seeds to iterations, loading the file, and reorganizing the workload before bootstrap is run.

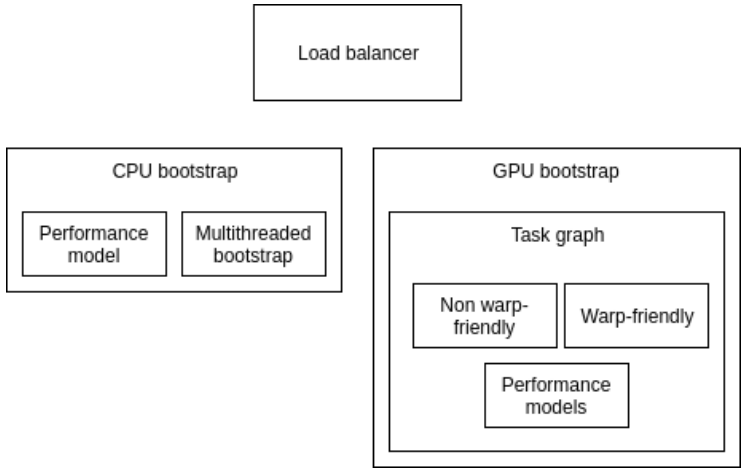


Figure 4.1: High level structure of the bootstrap program.

The overall structure of the program can be seen in figure 4.1. The purpose of the

task graph is to determine the optimal execution path of the GPU bootstrap process, and is therefore located in the GPU bootstrap module.

4.3.1 Program flow

At program launch, the command line arguments are parsed. Then, if the seeds to iterations counts mapping of the specified pair of l and p does not exist, it is generated and saved to disk. A time series of length l is generated using a PRNG, which is used as a synthetic input for the program.

The input time series to the program is always drawn from a uniform integer distribution. The size of it is specified as a command line argument, but the values themselves are always randomly generated. This is because none of the methods tested in this program depends on the values of the data, but rather the size of the time series and the other parameters. The result of bootstrapping is discarded, because it is of no interest to this thesis.

The load balancer is run and partitions the work between the CPU bootstrap module and the GPU bootstrap module. The CPU bootstrap module spawns additional threads for distributing the workload over additional CPU cores, with an equal amount of work assigned to each core. The GPU bootstrap analyses the task graph and the performance models to find the optimal sequence of operations to perform, and executes that sequence. At last, the results of both modules are combined to form the final result.

4.3.2 Command line arguments

There are numerous options that can be specified on the command line, of which the most important are listed here:

Name	Description
-l	Time series size
-p	p-parameter to the geometric distribution
-n	Number of resamplings
–gpu-fraction	Override load balancer, force gpu fraction
–force-path	Force execution through an operation
–device-id	Use the GPU with this id (as reported by CUDA)
–output-file	Path to output the statistics to (in JSON format)
–generate-iter-histogram	Generate a histogram of the iteration counts

Table 4.4: Some of the command line arguments of the bootstrap program.

The –gpu-fraction parameter can be used to override the load balancer and set the amount of resamplings for the GPU, and by extension the CPU. The –force-path parameter can be used to force execution through a specific operation of the task graph.

4.3.3 Recorded statistics

The program records a number of different statistics from its execution. The devices used (CPU, GPU) is recorded, along with the time each of the device used for executing the stationary bootstrap. Statistics from the load balancer, such as the estimated CPU and GPU cost, and the final n_{cpu} and n_{gpu}, are recorded. The time used for optimization, and the estimated cost of each device at each step is also recorded.

For the CPU, the number of resamplings per thread is recorded. For the GPU, the selected optimal execution path and the time used for finding it is part of the output, and the actual execution time of each operation.

The statistics are formatted using the nlohmann::json library [Loh17] and output to a specified output file, or discarded if no such file has been specified.

4.3.4 Histogram of iteration counts

If the –generate-iter-histogram command line parameter is specified, the program generates a histogram of iteration counts using the seeds to iteration counts file. The bucket size is one, so the frequency of each iteration count is preserved with no loss of information. The histogram is output to the value specified with the –generate-iter-histogram parameter in JSON format using the nlohmann::json library.

4.4 Stationary bootstrap implementations

The bootstrap program implements the stationary bootstrap for both the CPU and the GPU. For the CPU, C++11 is used, and for the GPU, CUDA C++ is used.

4.4.1 General optimizations

Each iteration of the loop in the bootstrap method needs to calculate the sum of the values in the window that was randomly selected. The naïve method of doing this is looping over all the numbers in the window, adding each number to the sum. The goal is to create a new time series of equal length to the original, which means that we need to add l total numbers, making the method $O(nl)$, l being the length of the original time series, and n is the number of resamplings.

By precomputing a cumulative sum of the original time series, a large speedup is achieved, while also transforming the problem from a regular problem to an irregular one. Defining

$$c(n) = \sum_{i=0}^{n-1} t(i) \tag{4.1}$$

and

$$c(0) = 0 \tag{4.2}$$

and precomputing c for $i \in [0, n]$. Then, computing the sum of a window leaves us with this operation:

$$c(\text{end}+1) - c(\text{start}) = \sum_{i=0}^{\text{end}} t(i) - \sum_{i=0}^{\text{start}-1} t(i) \tag{4.3}$$

which is the same as the original operation, because

$$\sum_{i=\text{start}}^{\text{end}} t(i) = \sum_{i=0}^{\text{end}} t(i) - \sum_{i=0}^{\text{start}-1} t(i) \tag{4.4}$$

Running bootstrap without this optimization results in a regular problem. If each thread is assigned with resampling the time series once, then n threads are spawned, where each thread must calculate the sum of l numbers after resampling, and divide the sum by l. Running bootstrap with the cumulative sum optimization and assigning each thread with performing one resampling, means that each thread must compute i subtractions and additions, and the final division by l. The number i is dependent the bootstrap window sizes, which are drawn from a geometric distribution. To avoid each thread ending up with resampling the time series in the exact same way due to the deterministic nature of PRNGs, each thread uses a separate PRNG which is seeded with a unique number. Because each thread uses a separate PRNG, the number of windows drawn, i, is therefore not the same for all threads, causing an irregular workload.

This method of creating a cumulative sum of the input time series has a potential problem, if the values of the time series is represented using floating point numbers. Because the precision of floating point numbers decreases as the number it repre-

sents grows larger, the result of adding two numbers of very different magnitude may end up being equal to the larger of the two operands:
$a + b = a$ (for $a >> b$)
This problem may be triggered by two conditions, or the combination of both:

- Very large time series
- Time series with values of very different magnitude

Consider an original time series of size l, where all the values are of the same magnitude. Starting at the first number, and adding each subsequent number to that, will eventually (for large enough l) end up having the temporary sum being so large that adding the next number to it will have no effect, as the answer will be rounded down to what the temporary was before the addition.

To avoid this problem, the precomputation uses a temporary variable with higher precision than the points of the data series. *I.e.*, assuming the time series data is represented by IEEE 754 [IEE08] binary32 format, then the temporary will be represented using IEEE 754 binary64 format.

4.4.2 GPU kernels

Geometric distribution

cuRAND [NVI17a] is used for generating random numbers on the GPU. The stationary bootstrap depends on numbers drawn from both uniform and geometric distributions, but cuRAND only provides an implementation for the uniform distribution. Therefore, we have implemented a function that takes a uniformly distributed random number as input, and transforms it into a geometrically distributed number [Knu97]. Assuming U is a uniformly distributed number on $[0,1]$, then N will be

geometrically distributed with parameter p by Equation (4.5).

$$N = \lceil \frac{\ln(U)}{\ln(1-p)} \rceil \tag{4.5}$$

Using GPU shared memory

Listing B.1 shows the implementation of the bootstrap kernel for time series that fit entirely in GPU shared memory. The parameters of the kernel are, in order:

- cumulativeInput: Pointer to an array of the cumulative sum of the input time series.
- cumulativeInputSize: Size of the cumulative sum array, and therefore 1 larger than the original time series.
- out: Pointer to the start of the output result array.
- nBootstrapIterations: The number of resamplings to perform.
- states: Pointer to the array of PRNG states.
- p: The probability of success parameter to the geometric distribution.

Each thread performs exactly one resampling of the time series, and calculates exactly one arithmetic mean. The out-array is therefore of sufficient size to hold *nBootstrapIterations* results. Because of the need to map from PRNG seed to number of iterations of the main loop at line 27 in B.1, each thread has its own PRNG state, which is stored in the states array. This array is therefore large enough to store *nBootstrapIterations* PRNG states.

The unique *id* of each thread is first calculated. Then the cumulative sum of the input time series is loaded into a variable that resides in shared memory. The size of

this variable is specified at kernel launch time. The threads in the thread block are then synchronized to ensure that all threads have finished loading their part of the cumulative sum into the shared memory variable.

The loop at line 27 performs the actual resampling of the time series. Each iteration selects a window, and sums the values inside the window as shown in subsection 4.4.1 and adds it to the temporary sum. The loop runs until the sum of the window sizes is equal to the size of the original time series, so that the resampled time series is identical in size to the original.

At the very end, the sum of the resampled time series is divided by the size of the time series to obtain the arithmetic mean of the resampled time series, which is then stored in the output array.

Without GPU shared memory

Each point of the cumulative sum of the time series are stored using 4 bytes of memory. The maximum amount of shared memory per block is 48 KiB [NVI17c] for all hardware configurations tested. This amount of shared memory means the maximum size of the time series is $48 \cdot 1024/4 = 12288$ (minus one because the cumulative sum of the time series is one element larger than the time series). To support larger time series than this, a version of the kernel using global memory for storing the cumulative sum of the time series has been implemented, called the global memory kernel. See listing B.2 for the implementation.

The global memory kernel differs from the shared memory kernel in two ways. First, there is no code for loading the time series into shared memory, and second, the cumulativeInput array is read directly when computing the sum of the values in a window, instead of reading from shared memory. Everything else, including the parameters of the kernel, and the logic for selecting the start and size of the windows

are identical.

CUDA enabled GPUs have some amount of memory that can be configured as either shared memory or L1 cache memory. Because shared memory is not used in this kernel, the GPU is configured to use this memory as L1 cache memory by setting the cache config for the kernel to *cudaFuncCachePreferL1*.

Mapping seeds to iteration counts

A special kernel for generating the mapping from PRNG seed to iteration count has been implemented, as seen in listing B.3. The kernel is very similar to the global memory kernel, but instead of calculating the sum of the elements of the windows it selects, it counts the number of iterations required by the main loop to finish resampling. The output of the kernel is this iteration count, instead of the average of the resampled time series.

4.4.3 CPU implementation

The CPU implementation of the stationary bootstrap is written in C++11. The implementation can be seen in listing B.5.

The random number generator used is the standard template library implementation of a Mersenne Twister (MT) [98] PRNG. A specific instantiation of the MT PRNG, called std::mt19937 [cpl17], is used for generating random numbers. Random numbers generated by the MT are transformed into uniformly and geometrically distributed numbers by std::uniform_int_distribution [Ref17d] and std::geometric_distribution [Ref17c] respectively.

The CPU implementation is very similar to the GPU implementations, because of CUDA C's similarity to C++. The main difference is that each CPU thread performs

multiple resamplings of the time series, whereas each GPU thread performs exactly one resampling.

4.4.4 Sources of branch divergence in the GPU kernels

There are three sources of branch divergence in the shared memory bootstrap kernel, and two in the global memory bootstrap kernel.

The first, which is not present in the global memory bootstrap kernel, is the conditional statement in the loop where the loading of the time series into shared memory happens. The worst case of branch divergence is that 31 threads will be inactive for one iteration of this very short loop, which is negligible to the divergence introduced by the main loop.

The conditional that ensures the correct number of resamplings is performed also results in a maximum of 31 idle threads, but for a longer amount of time than the previously discussed case. The amount is, however, still negligible to the amount of idle thread-time in the main loop.

The main loop is the main source of branch divergence. Each thread must wait (is set inactive) until the thread with the largest workload (largest number of iterations) has finished. All threads which have a workload less than the largest workload in the warp is therefore idling, and wasting computational resources. The amount of idling depends on the variance of the sizes of the workloads. If all workloads are identical in size, no threads will spend time waiting, and no time will be wasted. On the other hand, if all workloads have a size of 1, except one of size w_{max}, then 31 of the threads will spend w_{max}-1 iterations waiting for the thread with the largest workload to finish, wasting $\frac{31}{32} \approx 96.9\%$ of the computational resources of that warp while the threads are idling.

4.5 CPU bootstrap module

The CPU bootstrap module implements stationary bootstrap on the CPU. It consists of the procedure for performing stationary bootstrap, and a performance model used by the load balancer.

The CPU implementation is parallelized by launching m threads, where each thread is assigned $\frac{n_{\text{cpu}}}{m}$ resamplings. m is static and always selected to be 2 less than the number of threads the CPU can execute concurrently. Threads are launched and managed by using the C++11 feature std::async [Ref17b]. This allows for one thread to be dedicated to launching kernels and communicating with the GPU, and one thread to spare to avoid threads being preempted by the operating system for other tasks.

The performance model of the CPU bootstrap module is a P-spline with $\lambda = 0.001$. Every time the module is run, the time used is recorded and used to update the spline. The spline is saved to disk at program exit, and loaded each time it is needed.

4.6 GPU bootstrap module

4.6.1 Warp-friendly and non warp-friendly bootstrap

To facilitate discussion on the topic, we distinguish between *warp-friendly* (WF) bootstrap and *non warp-friendly* (NWF) bootstrap. We say that non warp-friendly bootstrap is when one of the bootstrap kernels in Section 4.4.2 is executed without performing any reorganization of the threads beforehand. Warp-friendly bootstrap is when the seeds to the PRNGs are reorganized as described in Section 3.2.2. We expect less branch divergence in each warp than if the workload had not been reorganized, and therefore name it warp-friendly.

Even though we distinguish between the two, and talk about them as if they were different kernels, they are the same kernels, but with additional preprocessing of seeds in the case of warp-friendly bootstrap. Note that the overhead of reorganizing the workload before execution by the warp-friendly kernel is timed and executed separately from the warp-friendly kernel.

4.6.2 Implemented operations of the task graph

ID	Operation name	Device
1	Initialize	Both
2	Generate seeds CPU	CPU
3	Generate seeds GPU	GPU
4	Get iteration counts 1	CPU
5	Get iteration counts 2	CPU
6	Initialize PRNG 1	GPU
7	Initialize PRNG 2	GPU
8	Sort seeds on the CPU	CPU
9	Sort seeds on the GPU	GPU
10	Copy seeds from GPU to CPU	Both
11	Copy seeds from the CPU to GPU	Both
12	Copy iteration counts from CPU to GPU	Both
13	Copy seeds and iteration counts from CPU to GPU	Both
14	Warp-friendly bootstrap	GPU
15	Non warp-friendly bootstrap	GPU
16	Copy result from GPU to CPU	Both
17	Finalize	Both

Table 4.5: Table of all operations.

Table 4.5 lists all the implemented operations, along with the device the operation executes on. Some operations have the device listed as *both*, which is either because the operation involves memory transfer between the CPU and GPU, or because the operation executes partly on each device. The operation named *Initialize*, for example, allocates memory on both the CPU and the GPU, and is therefore listed as both.

Below follows a description of all the implemented operations and the underlying functions.

Initialize

On the CPU, *Initialize* allocates memory for the seeds, iteration counts, and the result of the bootstrap. These three are C++ vectors of type *int*, *int* and *float*, respectively, and all three have space for n elements, where n is the number of resamplings.

On the GPU, two arrays of size n are allocated: The first to store the PRNG seeds, the second to store the state of the PRNGs. A third array of type float is also allocated, which is of size $l+1$, to store the cumulative sum of the input time series.

Finally, the cumulative sum of the time series is copied from CPU memory to the GPU memory that was just allocated.

Generate seeds CPU

Generate seeds CPU generates n numbers to be used for seeds for the PRNGs on the GPU. It uses a Mersenne Twister [98] with C++s std::uniform_int_distribution [Ref17d] to draw random numbers in the range $[0, 2^{31}]$, and store them in a C++ vector. The memory for the vector is not allocated by this operation, but by the

Initialize operation.

Generate seeds GPU

Generation of random seeds on the GPU is done by first drawing a single random number, called the global seed, from the Mersenne Twister on the CPU. The *generate seeds* kernel (see listing B.4) is then executed with that random number as a parameter. Each thread calculates its own unique *id*, and initializes a PRNG with the sum of the global seed and the unique *id* as seed. A random number is then drawn from the PRNG and stored in the output array.

After the seeds are drawn, they are sorted in ascending order in order to achieve better spatial locality in the Get iteration counts 1 & 2 operations. Note that this optimization was implemented very late in the project, and there was not enough time to implement it for the *Generate seeds CPU* operation.

Get iteration counts 1 & 2

Get iteration counts 1 and *Get iteration counts 2* have the same underlying function.

For all seeds in the seed vector, the corresponding iteration count is loaded from the seeds to iteration counts mapping and stored in a vector that has previously been allocated.

The seeds array must exist in CPU memory and the seeds must have been generated when this operation is executed.

Initialize PRNG 1 & 2

Initialize PRNG 1 and *Initialize PRNG 2* have the same underlying function.

Each thread calculates its unique *id*. The PRNG state of the thread is loaded from

the array of PRNG states, and seeded with the threads unique seed, which is in the seeds array.

Sort seeds on the CPU

Two of the preconditions of this operation is that the seeds has been drawn and exist in CPU memory, and that the iteration counts corresponding to the seeds has been loaded into CPU memory. The maximum value of the iteration counts array is determined, and an array, called m, of that size is allocated. Each element of that array is itself a vector, with no preallocated memory. Then, assuming s is the seed, and i is the iteration corresponding to that seed, s is appended to the vector at position i in m. Lastly, m is flattened and stored in the seeds array, which is now sorted.

The algorithm is a linear sort similar to counting sort. Counting sort could not have been used directly, because even though the iteration counts would be correct, the sizes of the windows drawn would not have been correct.

Sort seeds on the GPU

Both the seeds array and the associated iteration counts array must exist in GPU memory when this operation is executed. thrust::sort_by_key [NVI17d] is used for sorting the seeds according to the iteration counts, and the iteration counts array memory is released after sorting.

Copy seeds from GPU to CPU

The array of PRNG seeds are copied from GPU memory to CPU memory. *Generate seeds GPU* must have been executed beforehand.

Copy seeds from the CPU to GPU

The array of seeds are copied from CPU memory to GPU memory. *Generate seeds CPU* must have been executed beforehand.

Copy iteration counts from CPU to GPU

The array of iteration counts are copied from CPU memory to GPU memory. *Get iteration counts 1 or 2* must have been executed beforehand so that the iteration counts array are populated with the iteration counts corresponding to the seeds.

Copy seeds and iteration counts from CPU to GPU

Both the seeds array and the iteration counts array are copied from CPU memory to GPU memory.

Warp-friendly bootstrap

The bootstrap kernel is run. The PRNGs must have been seeded with **reorganized** seeds before the operation is executed. The result of the resamplings are stored in the output array.

Non warp-friendly bootstrap

The bootstrap kernel is run. The PRNGs must have been seeded with the generated seeds before the operation is executed. The result of the resamplings are stored in the output array.

Copy result from GPU to CPU

The output array is copied from GPU memory to CPU memory. Either of the bootstrap operations must have been run before this operation is executed.

Finalize

Memory allocated on the GPU is released, which includes the input cumulative time series, the output array, PRNG states array, seeds array and iteration counts array. Memory that has already been released will not be released again.

The allocated CPU memory is allocated on the stack and therefore released automatically at a later point in the program.

Chapter 5

Test cases

In this chapter we present our experiments. The chapter is divided into three main sections. The first section describes the experiments for evaluating our method of reorganizing the workload before execution on the GPU. The second section presents the experiments that was performed to evaluate the task graph method. Finally, the experiments for evaluating our approach for partitioning the workload between the CPU and GPU is described.

For many of the experiments, similar program arguments were used. For the time series length, l, the value 12000 was often used. This is approximately the maximum number that allows for the shared memory kernel to be used. Larger values means the time series does not fit in the shared memory of a thread block, and the global memory kernel must be used. The shared memory kernel exhibits bigger difference between the warp-friendly and non warp-friendly bootstraps, which is why the shared memory kernel was preferred for many experiments.

Because of insufficient global memory on the GTX 480, the approximate maxi-

mum value of n is 20,000,000. To compare the hardware configurations, this value is therefore often used because it maximizes the execution time while being able to run on all configurations.

5.1 Reorganization of the workload

5.1.1 Warp execution efficiency

Warp execution efficiency (WEE) is a "ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor expressed as percentage" [nvp17]. In other words, it is the ratio between the sum of the green bars in Figure 3.4a divided by the sum of the green and red bars in the same figure. The metric is defined on the interval $(0.0, 1.0]$. A ratio close to 0.0 means most of the execution time is spent with most threads deactivated, while a ratio of 1.0 means no thread was deactivated during the execution, and therefore no thread time was wasted. The metric is very useful in validating that warp-friendly bootstrap uses the GPU resources better than non warp-friendly bootstrap.

The metric was computed for these cases:

- $l \in [100, 20000]$, $p = 0.01$, $n = 20000000$
- $l = 12000$, $p \in [0.001, 0.2]$, $n = 20000000$
- $l = 12000$, $p = 0.1$, $n \in [1000, 20000000]$

The warp execution efficiency was computed for both the warp-friendly and non warp-friendly bootstrap in all cases.

5.1.2 Execution time

To evaluate the performance of warp-friendly (WF) bootstrap compared to non warp-friendly (NWF) bootstrap, a set of metrics was collected during execution of the kernels. The execution time of the WF and NWF kernels, along with the execution time

of the overhead of WF are important, as the main motivator for performing warp-friendly bootstrap in this thesis is to achieve a lower execution time overall.

The following metrics were all collected for the same set of parameters. The experiment was run on hardware configurations 1 and 2. Execution path 3 (A.4) was used for recording metrics for warp-friendly bootstrap, as it was found to be the faster of the three WF paths for all inputs except very low n. The load balancer was manually overridden to delegate all work to the GPU.

Kernel including overheads

The total execution time of the GPU bootstrap module was recorded for both WF and NWF bootstrap. Inputs where this metric is lower for WF bootstrap than NWF bootstrap benefit from workload reorganization. Note that the metric does not include the time to load the seeds to iterations file.

Kernel time

The execution time of the warp-friendly and non warp-friendly kernels is a useful metric for determining whether or not the reorganization of the workload leads to better performance of the kernel.

WF kernel and overheads ratio of total execution time

The execution times of the bootstrap kernel along with the 3 operations that contributed the most to the total execution time of WF bootstrap was recorded. This is useful to identify operations that are time consuming and may be subject to optimizations.

Total execution time

The time from program start to end was recorded, therefore also including the time to load the seeds to iterations file.

Speedup of the WF kernel over the NWF kernel

The speedup of a process t_1 over a process t_2 is defined as

$$r_{\text{speedup}} = \frac{t_2}{t_1} \tag{5.1}$$

The metric is useful for determining how much faster one process is over another. For example, a speedup of 2 for process 1 over process 2, means process 1 finishes in half the time of process 2.

The speedup as measured is compared to the speedup as estimated by dividing the warp execution efficiency (WEE) of WF bootstrap over the WEE of NWF bootstrap.

5.1.3 Seeds to iterations file load time

The files containing the mapping from seeds to iteration counts are very large (8 GiB), and loading them from disk into memory may increase the total time of the program by a non-trivial amount.

We distinguish between two different metrics. The first is called "hot loading", which is when the program was very recently run, and then started again. It is expected for hardware configurations with sufficient memory available, that the pages containing the file are still resident in main memory, and that fewer disk accesses are necessary. The other metric is called "cold load", which is when the entire file

must be read from disk.

To measure hot load times, we first performed a warm-up run to load the file into memory. Then, the program was restarted a number of times with the same parameters. To measure cold load times, two unique sets of parameters were chosen, and the program was started with these parameters alternating.

5.2 Task graph

5.2.1 Execution path selection

It is important that the task graph optimizer is able to select an execution path that minimizes the execution time. Over 40 executions of the program for each of hardware configurations 1 and 2, the execution path selected by the task graph optimizer was compared to the actual optimal execution path.

5.2.2 Determining optimal execution path time

The time for finding the optimal execution path was recorded over 40 program executions for hardware configurations 1 and 2. It is important that the time used for finding the optimal execution path is low, because it must be found in each iteration of the load balancer. Even if the load balancer is not run, the optimal execution path must be found at least once, assuming the GPU is assigned any resamplings at all.

5.3 Load balancer

The performance of the load balancer is crucial to achieving good performance of the overall program. If the load balancer is unable to find a good balance between

the CPU and GPU, or if it uses a very long time to find such a balance, it will be detrimental to the performance of the entire program.

It is important to note that the load balancers ability to find a fraction f_{cpu} that minimizes the objective function in (3.5) is of importance, and not the actual result of the execution. The reasoning is that if the load balancer finds a fraction that balances the *estimated* execution time of the CPU and GPU well, but the actual execution suffers from poor balance, that means that the performance models yielded poor estimates of the execution time. The load balancer is therefore evaluated on its ability to find a fraction that minimizes the difference in *estimated* execution times, and not the actual execution times.

The first four metrics were collected as such: For hardware configurations 1 and 2, the performance models was minimally trained for $l = 12000$ and $p = 0.1$, meaning the execution path was forced through each path 4 times, to allow the estimators to be built. Then, using the same values of l and p, the program was run using 4 values of n. The values of n will depend on the specific GPU, due to memory limitations. The ns will not be the same as the ns used for training the estimators, so that they will have to estimate the execution time for unseen values of n. The process is repeated 10 times for each n, for a total of 40 runs per hardware configuration.

For hardware configuration 1, $n = 80,000,000$, $n = 40,000,000$, $n = 20,000,000$ and $n = 10,000,000$ were used. For hardware configuration 2, $n = 20,000,000$, $n = 10,000,000$, $n = 5,000,000$ and $n = 2,500,000$ were used.

5.3.1 Optimization time

The time used by the load balancer has a direct impact on the execution time of the program. Finding a perfect balance is only worthwhile if the time saved by balancing the load is larger than the time used for finding the balance point.

5.3.2 Estimated execution time

The estimated execution times of each device after load balancing was recorded. If the load balancer finished before reaching the maximum number of iterations, it was able to find a good partitioning of the workload. This metric is therefore not very useful by itself, but compared to the actual execution times it can be used to determine the quality of the estimators.

5.3.3 Actual execution time

The actual execution times of the devices for each execution was recorded. If the execution time of each device is equal, it means that either both the estimators and the load balancer performed fine, or a lucky combination of poor performance by both lead to good results.

5.3.4 Actual and estimated ratios

The ratio of the minimum and maximum of the CPU and GPU execution times is used to determine how well the load balancer were able to balance the workload. Equation (5.2) shows the metric, which has a range of $[0,1]$. $r = 1$ means $t_{cpu}(f^n_{cpu}) = t_{gpu}(1 - f^n_{cpu})$, which is a perfect balance. $r = 0$ means one of the devices was assigned all of the workload, while the other device is left idle, yielding the poorest balance of the workload.

$$r = \frac{\min(t_{cpu}(f^n_{cpu}), t_{gpu}(1 - f^n_{cpu}))}{\max(t_{cpu}(f^n_{cpu}), t_{gpu}(1 - f^n_{cpu}))} \tag{5.2}$$

The estimated and actual ratios were recorded. The estimated ratio is the ratio of the estimated CPU and GPU execution times, while the actual ratio is the ratio of the actual execution times of the devices.

5.3.5 Balanced and unbalanced execution time

Dividing the total workload across devices is done to achieve better performance. It is therefore crucial that performance is actually improved by performing load balancing. To evaluate this, the execution time of assigning all of the work to the CPU or GPU, and the execution time when balancing the workload between the devices, was measured. This was done for hardware configurations 1 and 2, with the parameters $l = 12000$, $p = 0.1$ and $n = 20000000$ for all executions. The experiment was repeated 10 times by executing the entire workload on the CPU, 10 times by executing the entire workload on the GPU, and 10 times by executing with the load balancer enabled.

Chapter 6

Results

This chapter present the experimental results. The chapter layout mirrors the layout of Chapter 5, where the methodology and parameters of each experiment is outlined.

Many of the test cases involve experiments on hardware configurations 1 and 2. In these cases, the result of running the experiment on the hardware configuration with the NVIDIA GTX 1080 Ti is presented in the left-hand figure, and the results of running the experiment on the other hardware configuration is presented in the right-hand figure.

The results are discussed in Chapter 7.

6.1 Reorganization of the workload

6.1.1 Warp execution efficiency

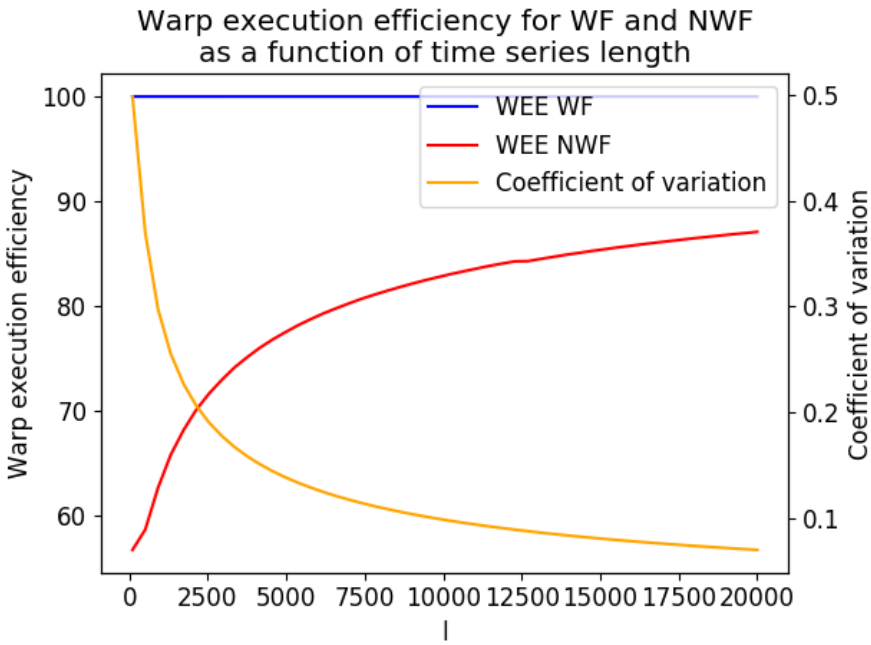


Figure 6.1: Warp execution efficiency and coefficient of variation as a function of l.

The warp-execution efficiency is improved to a perfect score of 100% by reorganizing the workload for all values of l. When not reorganizing the workload, the warp execution efficiency seems to be negatively correlated with the coefficient of variation.

The Pearson correlation coefficient [Lin89] for the warp execution efficiency of NWF and the coefficient of variation is -0.962 for this data set.

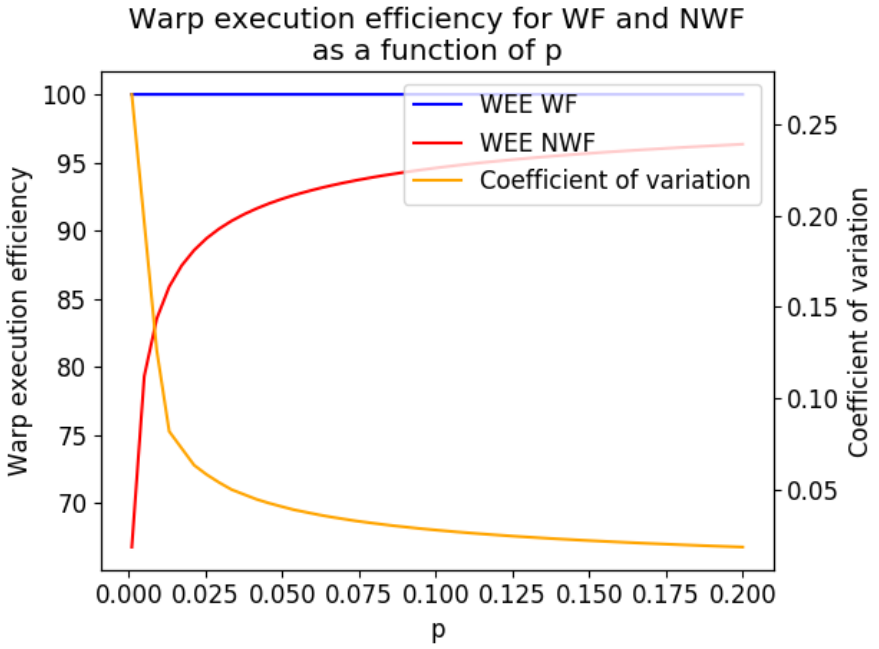


Figure 6.2: Warp execution efficiency and coefficient of variation as a function of p.

The warp execution efficiency is also improved to a perfect score of 100% for all p. For NWF bootstrap, the warp execution efficiency is negatively correlated with the coefficient of variation, as in the previous figure.

The Pearson correlation coefficient for the warp execution efficiency of NWF and the coefficient of variation is -0.981 for this data set.

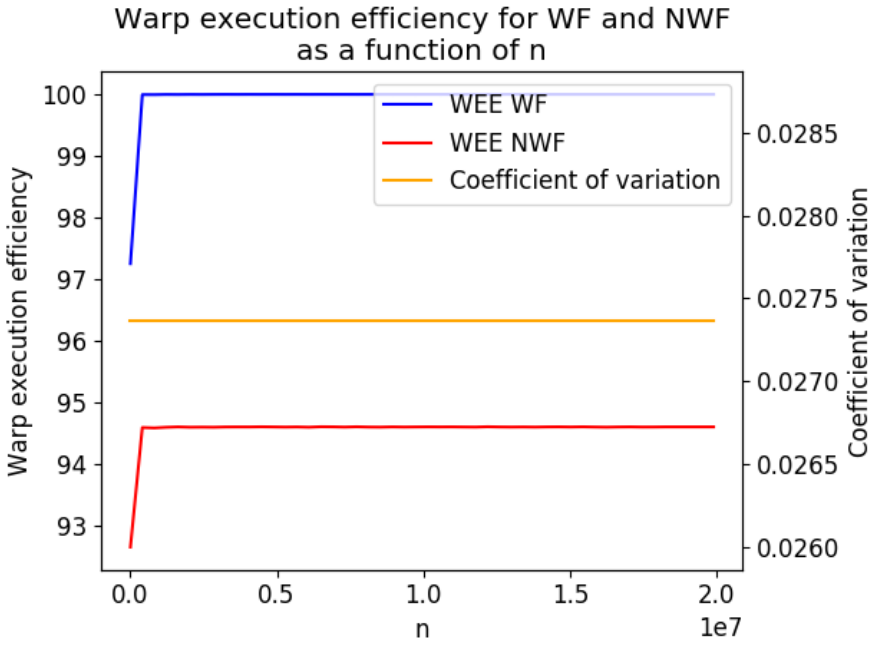


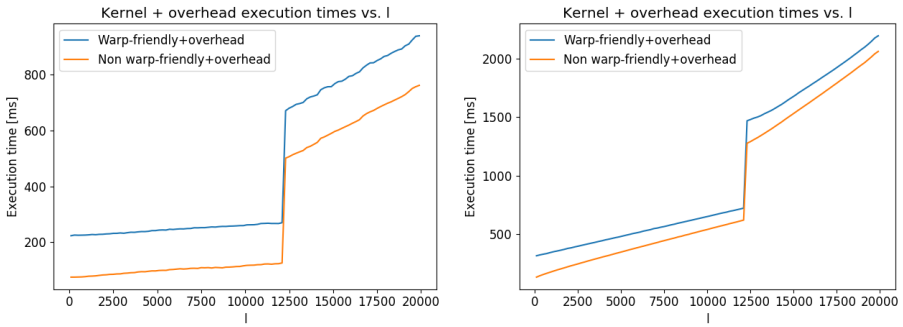
Figure 6.3: Warp execution efficiency and coefficient of variation as a function of n.

The warp execution efficiency is improved from around 94.5% to a perfect score of 100% for n above $n \approx 400,000$. Note that the warp execution efficiency was measured from $n = 1000$ to $n = 20,000,000$ in increments of approximately $400,000$, and then linearly interpolated to generate the figure. The warp execution efficiency is therefore not necessarily linear between $n = 1000$ and $n = 400,000$ (or between any other points), even if it appears so in the figure.

The Pearson correlation coefficient for the warp execution efficiency of NWF and the coefficient of variation is 0 for this data set.

6.1.2 Execution time

WF and NWF times with overheads

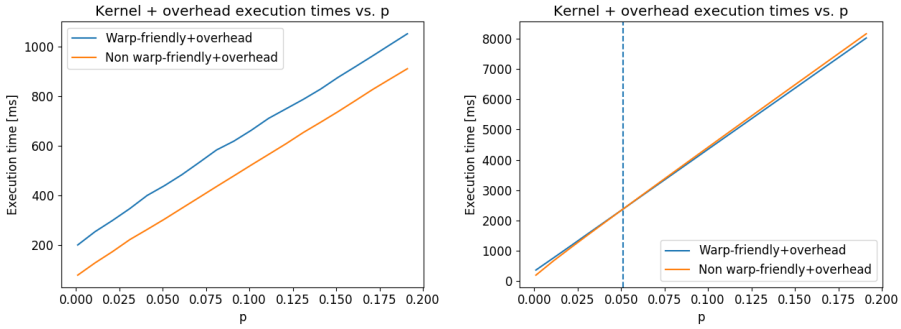


(a) NVIDIA GTX 1080 Ti.

(b) NVIDIA GTX 480.

Figure 6.4: Execution times of the WF and NWF kernels including overheads as the time series length is increased.

At $l = 12000$ we can see the increase in execution time as the global memory kernel is used instead of the shared memory kernel.

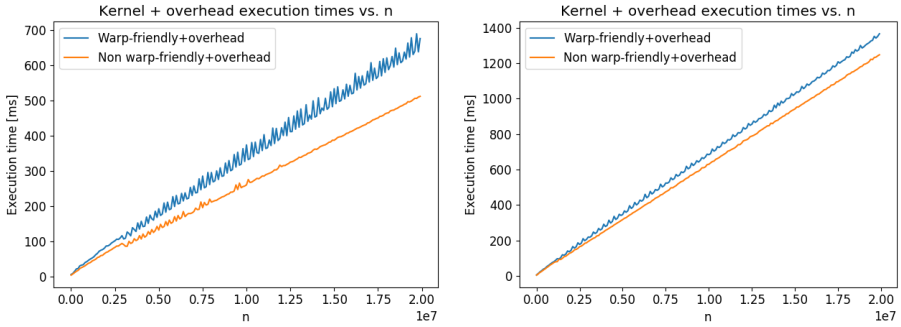


(a) NVIDIA GTX 1080 Ti.

(b) NVIDIA GTX 480.

Figure 6.5: Execution times of the WF and NWF kernels including overheads as p is increased.

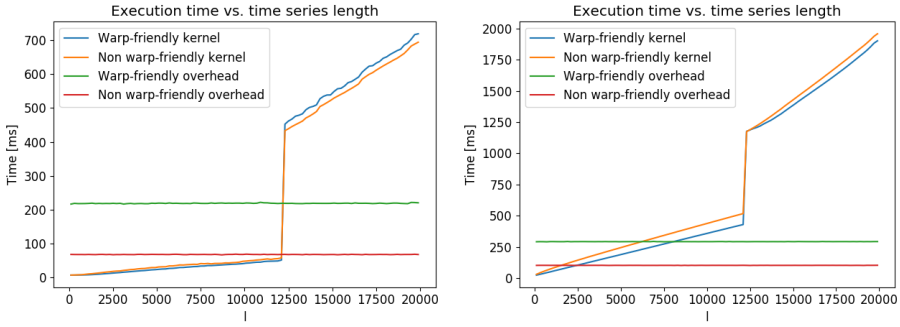
The dotted vertical line in the right-hand plot is the point where the total execution time of the WF kernel with overhead is less than the total execution time of the NWF kernel with overhead. As we can see, at $p = 0.05$, the optimal path changes from non warp-friendly bootstrap to warp-friendly bootstrap. Note that this does not include the time to load the seeds to iteration counts file.



(a) NVIDIA GTX 1080 Ti. (b) NVIDIA GTX 480.

Figure 6.6: Execution times of the WF and NWF kernels including overheads as n is increased.

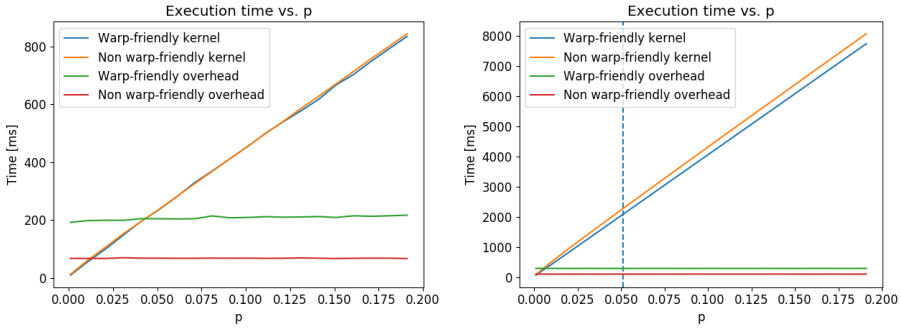
WF and NWF kernel time



(a) NVIDIA GTX 1080 Ti. (b) NVIDIA GTX 480.

Figure 6.7: Kernel execution times and overheads as l is increased.

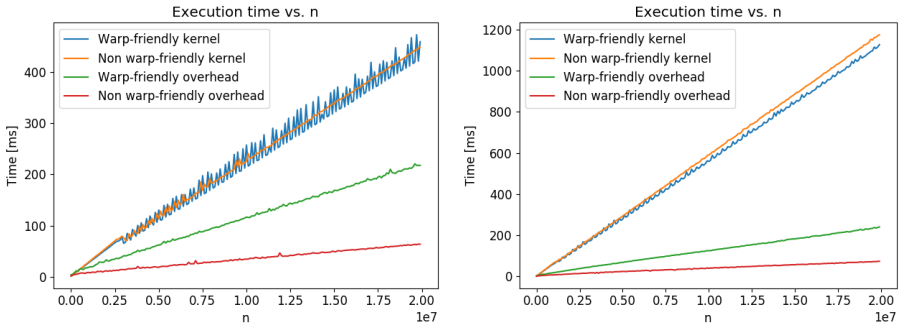
The switch from the shared memory kernel to the global memory kernel can be seen at $l = 12000$, where the execution time of both WF and NWF bootstrap increases drastically.



(a) NVIDIA GTX 1080 Ti. (b) NVIDIA GTX 480.

Figure 6.8: Kernel execution times and overheads as p is increased.

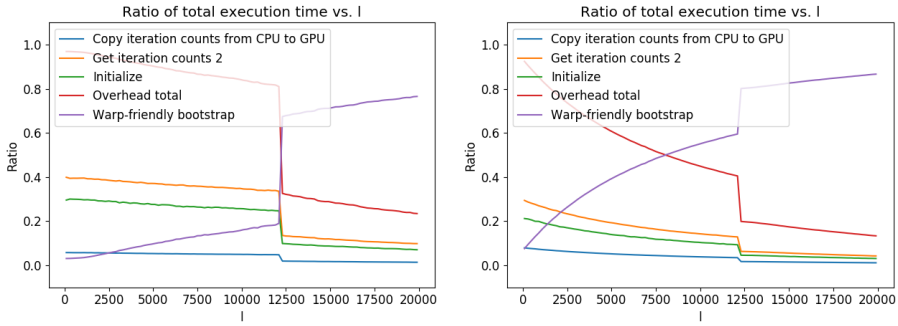
The dotted vertical line is the point where the total execution time of the WF kernel with overhead is less than the total execution time of the NWF kernel with overhead. As we can see, at $p = 0.05$, the optimal path changes from non warp-friendly bootstrap to warp-friendly bootstrap. Note that this does not include the time to load the seeds to iteration counts file.



(a) NVIDIA GTX 1080 Ti. (b) NVIDIA GTX 480.

Figure 6.9: Kernel execution times and overheads as n is increased.

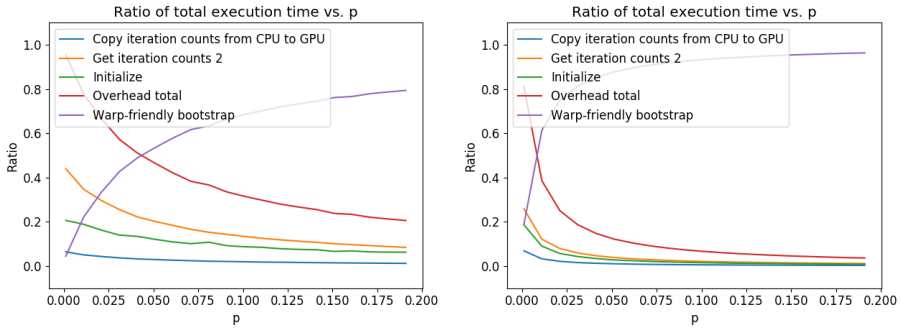
WF kernel and overheads ratio of total execution time



(a) NVIDIA GTX 1080 Ti. (b) NVIDIA GTX 480.

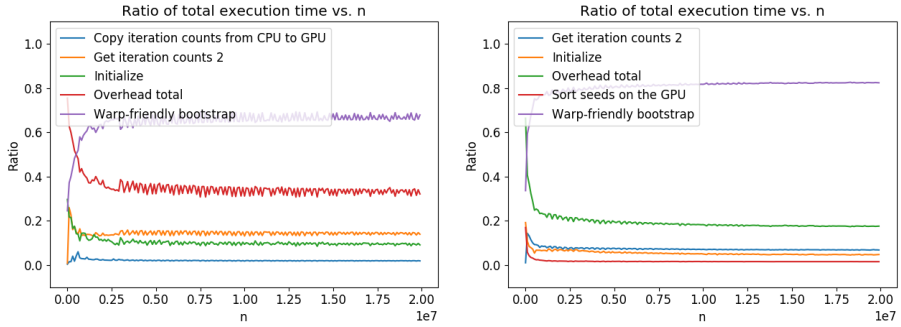
Figure 6.10: The execution time of the most time consuming operations as a ratio of the total execution time in execution path #3 as the time series length is increased.

The sudden increase in execution time of the warp-friendly bootstrap operation at $l = 12000$ is due to the change from the shared memory kernel to the global memory kernel.



(a) NVIDIA GTX 1080 Ti. (b) NVIDIA GTX 480.

Figure 6.11: The execution time of the most time consuming operations as a ratio of the total execution time in execution path #3 as p is increased.

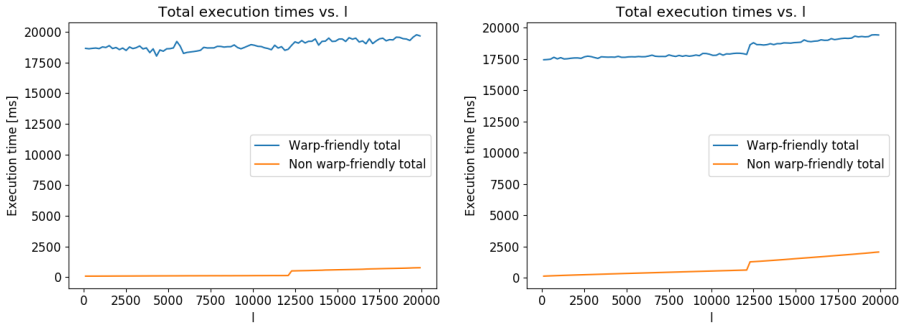


(a) NVIDIA GTX 1080 Ti.

(b) NVIDIA GTX 480.

Figure 6.12: The execution time of the most time consuming operations as a ratio of the total execution time in execution path #3 as n is increased.

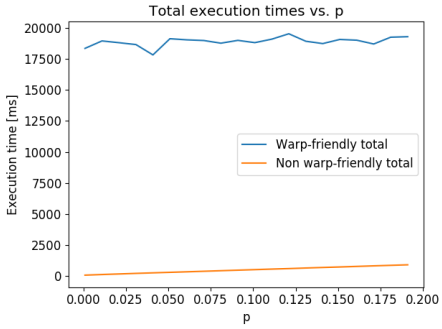
Total execution time



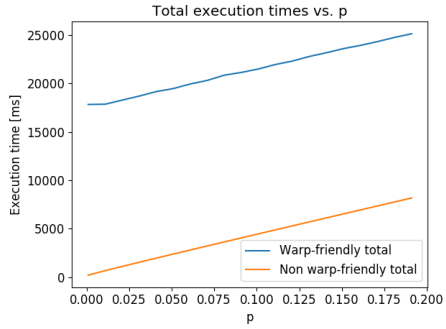
(a) NVIDIA GTX 1080 Ti.

(b) NVIDIA GTX 480.

Figure 6.13: Total execution time from program start to finish as a function of time series length. Note that the time to load the seeds to iterations file is included.

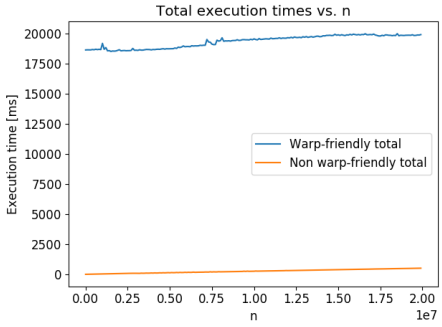


(a) NVIDIA GTX 1080 Ti.

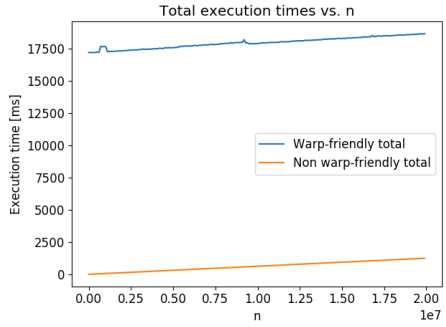


(b) NVIDIA GTX 480.

Figure 6.14: Total execution time from program start to finish as a function of p. Note that the time to load the seeds to iterations file is included.



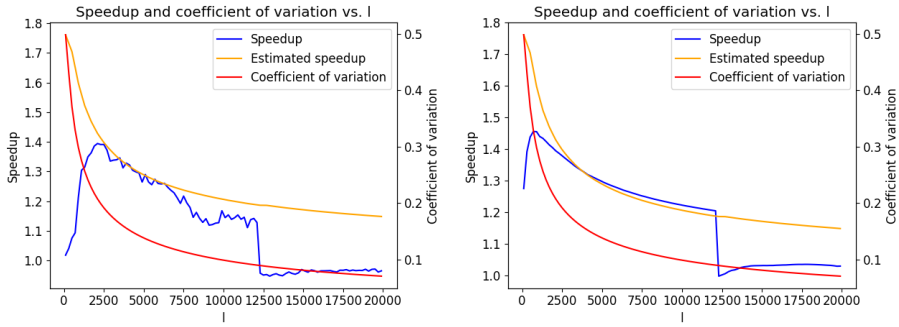
(a) NVIDIA GTX 1080 Ti.



(b) NVIDIA GTX 480.

Figure 6.15: Total execution time from program start to finish as a function of n. Note that the time to load the seeds to iterations file is included.

WF speedup over NWF

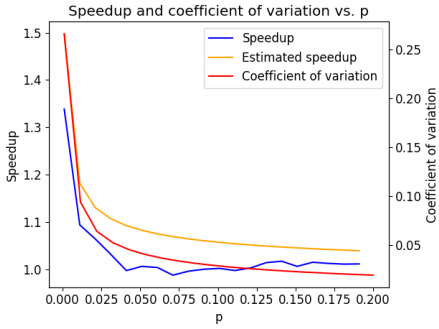


(a) NVIDIA GTX 1080 Ti. Pearson correlation coefficient for $l \in [700, 12000]$: 0.74.

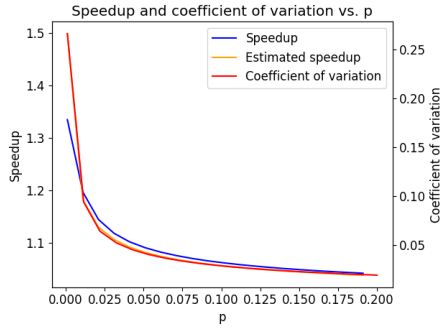
(b) NVIDIA GTX 480. Pearson correlation coefficient for $l \in [700, 12000]$: 0.98.

Figure 6.16: Speedup of the WF kernel over the NWF kernel and the coefficient of variation as the time series length is increased.

The Pearson correlation coefficient of the estimated speedup and the speedup is shown for the interval $l \in [700, 12000]$. For $l \in [0, 700]$, it is expected that the time to load the time series into the shared memory dominates the resampling time, and was therefore excluded. The interval $l \in [12000, 20000]$ was also excluded because it was out of the range of the shared memory kernel. For time series of length larger than approximately 12000, the global memory kernel is used instead of the shared memory kernel. In these figures, this can be seen by the sudden drop in the speedup.

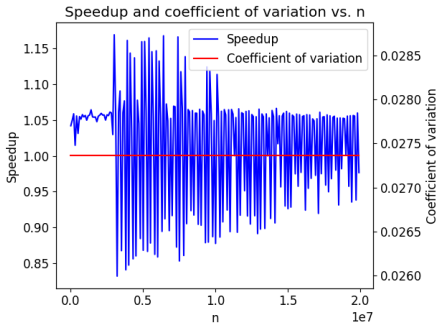


(a) NVIDIA GTX 1080 Ti. Pearson correlation coefficient: 0.98.

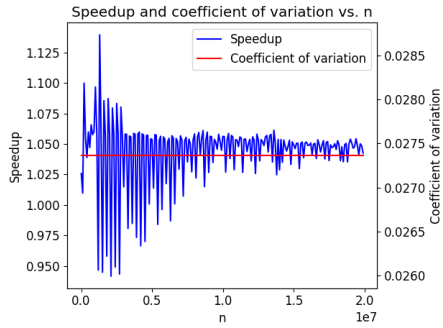


(b) NVIDIA GTX 480. Pearson correlation coefficient: 0.97.

Figure 6.17: Speedup of the WF kernel over the NWF kernel and the coefficient of variation as p is increased.



(a) NVIDIA GTX 1080 Ti. Pearson correlation coefficient: 0.



(b) NVIDIA GTX 480. Pearson correlation coefficient: 0.

Figure 6.18: Speedup of the WF kernel over the NWF kernel and the coefficient of variation as n is increased.

6.1.3 Seeds to iterations file load time

Configuration	Min	Max	Median	Stddev
Hot loads	2401.55ms	2449.79ms	2430.74ms	15.32ms
Cold loads	16742.70ms	16870.29ms	16776.42ms	46.28ms

Table 6.1: Load seeds to iterations file timings with 32 GiB of memory.

Configuration	Min	Max	Median	Stddev
Hot loads	16974.46ms	16995.96ms	16986.19ms	6.20ms
Cold loads	17138.63ms	17178.11ms	17146.86ms	12.46ms

Table 6.2: Load seeds to iterations file timings with 16 GiB of memory.

The hardware configuration with 32 GiB memory is much faster at hot loading the seeds to iterations file than the configuration with 16 GiB of memory.

6.2 Task graph

6.2.1 Execution path selection

The column titles of the following tables have the following meaning:

- n_{optimal}: Number of times the path was the optimal path.
- $n_{\text{correctly selected}}$: Number of times the path was the optimal path and was selected by the task graph optimizer.
- $n_{\text{incorrectly selected}}$: Number of times the path was selected when it was not the optimal path.

Path	n_{optimal}	$n_{\text{correctly selected}}$	$n_{\text{incorrectly selected}}$
1	1	0	0
2	0	0	0
3	0	0	0
4	58	58	1

Table 6.3: Optimal and selected paths statistics. Run with hardware configuration 1.

The task graph optimizer is able to select the optimal execution path in all but one of the cases, leading to an accuracy of 98.3%.

Path	n_{optimal}	$n_{\text{correctly selected}}$	$n_{\text{incorrectly selected}}$
1	1	1	0
2	0	0	0
3	33	32	1
4	25	24	1

Table 6.4: Optimal and selected paths statistics. Run with hardware configuration 2.

The task graph optimizer was able to select the optimal path with an accuracy of 96.6%.

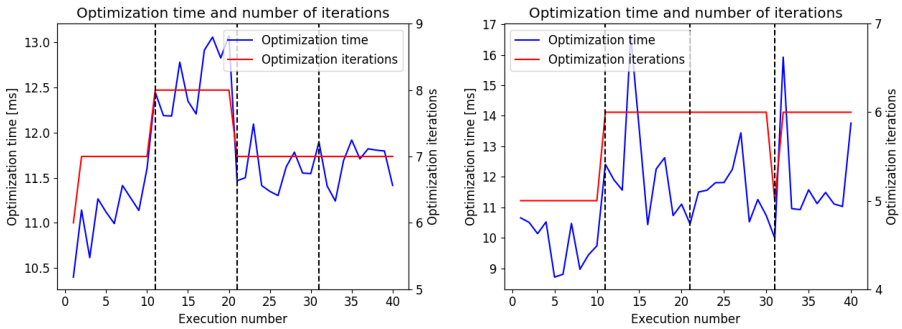
6.2.2 Determining optimal execution path time

Hardware configuration	Min	Max	Median	Stddev
Ryzen 1800x and GTX 1080 Ti	11.02ms	14.40ms	12.43ms	0.69ms
Ryzen 1800x and GTX 480	9.17ms	17.14ms	11.89ms	1.58ms

Table 6.5: Time used for determining the optimal execution path over 40 runs for each hardware configuration.

6.3 Load balancer

6.3.1 Optimization time

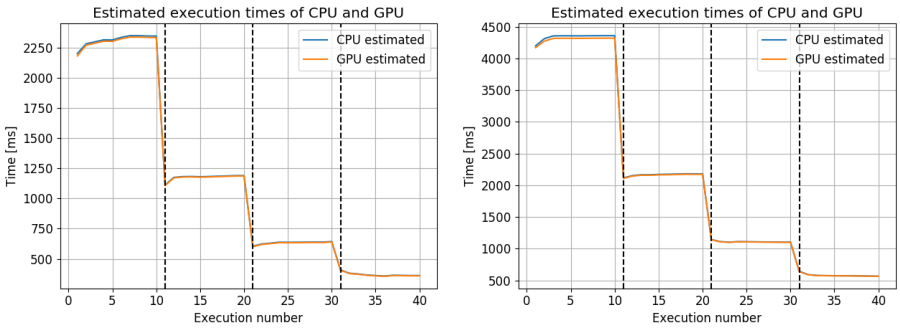


(a) Ryzen 1800x and GTX 1080 Ti.

(b) Ryzen 1800x and GTX 480.

Figure 6.19: Load balancer time and number of iterations for each run. The vertical dotted lines represent where n was changed.

6.3.2 Estimated execution time

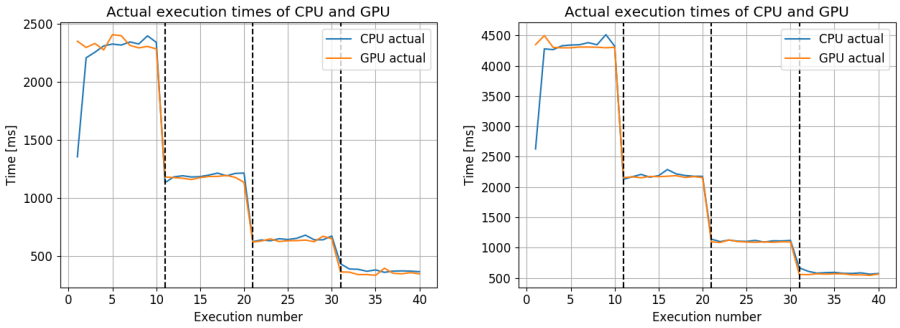


(a) Ryzen 1800x and GTX 1080 Ti.

(b) Ryzen 1800x and GTX 480.

Figure 6.20: Load balancer result for each run. Times are the estimated execution time of each device. The vertical dotted lines represent where n was changed.

6.3.3 Actual execution time

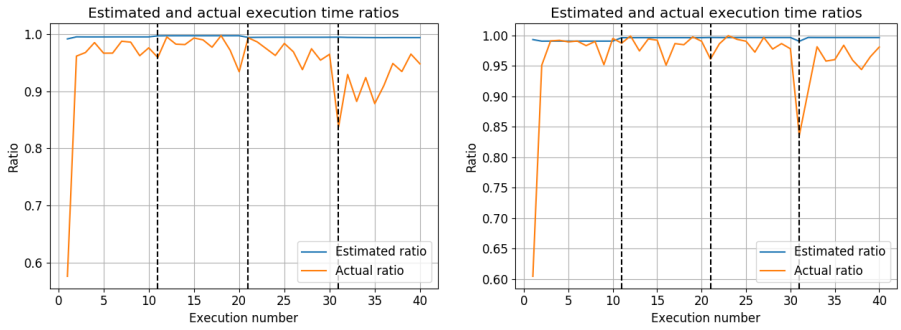


(a) Ryzen 1800x and GTX 1080 Ti.

(b) Ryzen 1800x and GTX 480.

Figure 6.21: Actual execution time for each run. The vertical dotted lines represent where n was changed.

6.3.4 Actual and estimated ratios

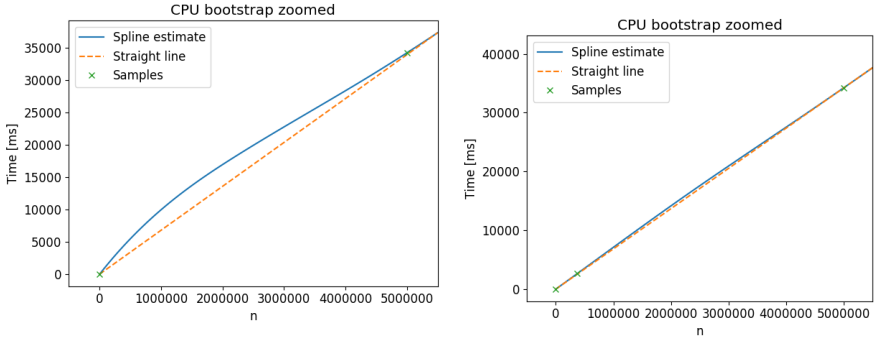


(a) Ryzen 1800x and GTX 1080 Ti.

(b) Ryzen 1800x and GTX 480.

Figure 6.22: Estimated and actual ratio of minimum to maximum execution time for the devices. The vertical dotted lines represent where n was changed.

As evidenced by the low actual ratio, the load balance was poor in the first execution of both cases. The load balancer performs much better in subsequent executions.



(a) CPU bootstrap estimator before the first run.

(b) CPU bootstrap estimator after first run.

Figure 6.23: The CPU bootstrap estimator before and after the first run.

In Figure 6.23a we see the estimated execution time of the CPU bootstrap module as a function of n. In Figure 6.23b we can see that the estimate in the interval $[0, 5000000]$ has been updated with the new measurement of the CPU bootstrap module execution time. Although hard to see, the estimate in that interval seems to have been too high, thus leading to the poor balance seen in Figure 6.22b and Figure 6.22a.

6.3.5 Balanced and unbalanced execution time

Configuration	Min	Max	Median	Stddev
All CPU	133403ms	142355ms	135150ms	2308ms
All GPU	563ms	586ms	578ms	7ms
Balanced	584ms	639ms	608ms	16ms
Balanced incl. optimization	597ms	653ms	622ms	16ms

Table 6.6: Ryzen 1800x and GTX 1080 Ti (Hardware configuration 1)

The speedup of the *All GPU* configuration over the *All CPU* configuration using the median execution times is 233.79.

Configuration	Min	Max	Median	Stddev
All CPU	133085ms	135466ms	134503ms	743ms
All GPU	4310ms	4317ms	4316ms	2ms
Balanced	4289ms	4433ms	4316ms	43ms
Balanced incl. optimization	4304ms	4450ms	4342ms	52ms

Table 6.7: Ryzen 1800x and GTX 480 (Hardware configuration 2)

The speedup of the *All GPU* configuration over the *All CPU* configuration using the median execution times is 31.16.

Chapter 7

Discussion

In this chapter we discuss the results of our experiments. We start with a discussion of the quality of our experimentation methodology. Next, we identify the similarities and differences of our approach to workload partitioning with existing approaches found in the literature. Then, the results of our experiments are discussed. The discussion of the experiments start with the discussion of the reorganization of workload, followed by a discussion of the task graph approach, and then a discussion of our method of workload partitioning between the CPU and GPU. Finally, we propose a method for estimating the reduction in execution time that is achieved by reorganizing the workload.

7.1 Quality of the method

Each unique pair of l and p requires the program to generate the seeds to iterations mapping before any experiments involving reorganization of the workload can be run. Additionally, the mapping must be loaded into memory, which takes a lot of time compared to the overall process. This makes sampling the parameter space of l and p very slow, and as a consequence of this, much of the parameter space has been left unexplored. For many of our experiments, we fixed two of the parameters while varying the third, to see how the different metrics were affected by the changes to that parameter. To cover more of the parameter space, random sampling could have been used instead. The drawback then would be that it would be harder to see how the metrics change as a function of each variable.

In the task graph experiments, it was shown that two of the execution paths were never the optimal path, except for in a single case. The evaluation of the quality of the method of using a task graph would be more interesting if there were more viable execution paths.

Both the CPU and GPU modules have execution times that are approximately linear in n. As a result of this, the objective function of the load balancer is piecewise linear. Such functions are relatively easy to find the optimal point of, as compared to non-linear functions, because there are no local minima that is not the global minimum.

7.2 Comparison to other approaches

7.2.1 Reorganization of workload

The workload reshaping approach described in Section 2.7.1 inspired our work on reorganization of the workload based on the seeds to the PRNGs. Their approach focuses on kernels consisting of one or more loops, and they map the data point to the number of iterations of the loops. This is very similar to our mapping from PRNG seed to the number of iterations of the loop in the bootstrap kernels. As in our approach, they perform a sorting of the workloads so that data points with similar number of iterations end up in the same warp.

In their approach, they perform a sampling of the workload and uses that to build a model of the workload. The reorganization of the data points is then done using this model. Our method exhaustively samples the input (the seeds), and then sort the seeds using this mapping. The approach taken by [She+13] would not work for reorganizing the seeds, as they assume that data points that are close have similar workloads. This is not true for the seeds.

After the reorganization is done, their approach identifies regions that are suitable for execution on the CPU and GPU. In our approach, this is not feasible. While it is possible to identify seeds that lead to high iteration counts, running those same seeds on the CPU would not have the same results as if they were run on the GPU. This is because a different PRNG is used in the CPU bootstrap than in the GPU kernels.

7.2.2 Task graph

StarPU (see Section 2.7.2) uses a task graph to represent the work that must be done, and the dependencies of one task to another. There are a number of similarities to our approach. The StarPU *task* is analogous to our *operation*, and their *codelet* is analogous to the underlying function of our operation. However, in StarPU each codelet may have several implementations it can run. This is not true for the operations of our task graph. In our approach, implementations for different devices are represented by different operations.

Both approaches uses a task graph to represent dependencies between tasks/operations. Additionally, both approaches model the execution time of the codelet/underlying function. In StarPU, the execution time may be modelled in a number of ways. One is history based, which records the execution time for a single input, and whenever that exact input is seen again, the recorded execution time is used as an estimate. Another model they use is regression based, like ours. The regression model they use is of the form $a * n^b + c$, which is a polynomial of degree b. As is discussed by [Sch07], spline regression is a more flexible regression method than single polynomial regression, so our spline-based approach should be able to better estimate the execution times than StarPU's method of using a single polynomial.

The main difference between our and StarPU's approach is our methods ability to select the optimal execution path. In StarPU, all tasks that are submitted are executed. To our knowledge, there is no functionality in StarPU to implement the same function in different ways as we do with our sequence of operations, and have it select the sequence of tasks/operations that is expected to minimize the overall execution time. Even if each path was represented by a different implementation in a codelet, each implementation is for a single device, and must therefore be run on

that device. In our approach, the path selected may run some operations on one device and then the next operation on another device, which is not possible with a codelet.

StarPU also offers the ability to model the energy consumption of each implementation of a codelet, and to optimize for lower energy efficiency instead of lower execution time. While not currently implemented in our task graph, this could easily be implemented by adding a secondary estimator to each operation, and train it using a measure of the energy consumption instead of the execution time. The task graph optimizer would then use the energy consumption models for determining the optimal execution path instead of the execution time models.

7.2.3 Load balancer

Our load balancer is a static load balancer which uses models to guide the optimization process. Similarly to the approach taken by [She+13] and the offline profiling of [She+16], our approach uses the execution times of previous runs as estimates of the execution time of each device. This differs from the machine learning approach by [FE16] and the online-profiling part of [She+16], where a small sample of the input data is run and the results of that run is used as an estimate for the throughput of the devices. The latter approach incurs a performance penalty as the small sample is executed, which is avoided by our method of using historical data modeled using splines.

In the offline profiling of [She+16] they build one linear regression model for each interval of input sizes that fit in different memories. This approach is not very flexible. First of all, the execution time in each interval is not necessarily linear in the input size, and second, if executed on a device with an extra layer of caching, the application code must be changed to build a model for that interval as well. Our ap-

proach of using splines for the execution time models would be able to capture the different execution times automatically, with no programmer intervention required.

7.3 Reorganization of workload

7.3.1 Warp execution efficiency

In Section 6.1.1 we saw that the coefficient of variation seems to be a good predictor for the warp execution efficiency. In Figure 6.1 and Figure 6.2, the Pearson coefficient of correlation is very near -1, which suggests that the two metrics are negatively correlated. In the third case, in Figure 6.3, the Pearson coefficient of correlation is 0. However, this is because the coefficient of variation is invariant to n, resulting in a straight line with a derivative of 0. The correlation coefficient of two sets of samples is defined in terms of the covariance of the samples, which is 0 when all samples in one or both of the sets have the same value.

The reorganization of the workload improved the warp execution efficiency to 100% in all cases except for very low n. When n decreases, the amount of warps that will be scheduled also decreases. For $n \leq 32$, a single warp will be scheduled. In this case, no matter how the seeds are reorganized they will always be executed by threads of the same warp, resulting in the same warp execution efficiency as if the seeds were not reorganized.

We can see that the warp execution efficiency of non warp-friendly bootstrap is increased as l or p is increased. This can be explained by looking at the associated coefficient of variation. As the coefficient of variation decreases, the amount of idle thread time is dominated by the total thread time, leading to a higher warp execution efficiency. The total amount of absolute idle thread time may still increase, however.

7.3.2 Execution time

In Section 6.1.2 we can see that in all but one case, the warp-friendly bootstrap is slower than the non warp-friendly bootstrap, if the overhead is included in the timing. The execution time of the overhead of WF bootstrap, which is dominated by the *Get iteration counts 2* operation (see Section 6.1.2), is larger than the time gained by reorganizing the workload.

In the interval $l \in [0, 12000]$ in Figure 6.4b, the difference between the WF and NWF bootstrap seems to decrease as l is increased. This suggests that there exists an l for which the WF bootstrap is faster than the NWF bootstrap, if the GPU had sufficient shared memory. This seems to also be the case for Figure 6.4a. However, in that case, the difference in execution times seems to decrease at a lower rate. For the global memory kernel (l larger than approximately 12000), the execution times of both the WF and NWF kernel is increasing at a similar rate, suggesting that reorganizing the workload will never be faster for a larger l and the same p and n.

7.3.3 Overhead of the warp-friendly bootstrap

The overhead of warp-friendly bootstrap, as seen in Section 6.1.2, seems to be of the same magnitude for both hardware configurations. The hardware of both hardware configurations are identical except for the GPUs and storage, which suggests that operations that run on the CPU should take approximately the same time for both configurations. The overhead operation that contribute the most to the execution time of the total overhead, the *Get iteration counts 2* operation, runs on the CPU. The second most time consuming overhead operation, the *Initialize* operation, performs memory allocation and memory copy on the GPU. We don't expect memory allocation time to vary a lot between the two GPUs, and the memory copy is lim-

ited by the bandwidth of the peripheral component interconnect (PCI) express bus, which is the same for both configurations.

The warp-friendly bootstrap execution time entirely depends on the throughput of the GPU. As the overhead is approximately the same for both hardware configurations, the ratio of the overhead to the total execution time is therefore a lot higher for faster GPUs.

Seeds to iterations file load time

The loading of the seeds to iterations file may or may not be counted in the total execution time of the bootstrap program. For a use case where p and the length of the time series are the same for every execution, the seeds to iterations mapping can be kept in memory while the program waits for more input. If this is not the case, loading the file is so time consuming that none of our experiments showed that the total time was lower for WF bootstrap than NWF bootstrap. There is a significant benefit of increasing the memory of the computer with respect to *hot loads*[1], as seen in Table 6.1. However, the total overhead would still be larger than the reduction in bootstrap kernel time for all our experiments.

Kernel execution time

For hardware configuration 2, the WF kernel is faster than the NWF kernel in all cases. It is also the true that the WF kernel execution time increases at a slower rate than that of the NWF kernel, with the exception of when the global memory kernel is used, where the execution times seem to grow at a similar pace. This is confirmed by the figures showing the speedup of the WF kernel over the NWF kernel, where the

[1] Hot loads, as defined in Chapter 3, is when the program has very recently been run with the same l and p, providing an opportunity for the operating system to reuse already existing memory pages that contain the seeds to iterations file.

speedup is positive in almost all cases, except for a few outliers when the speedup is measured against varying n. The speedup, when measured against varying l, is close to 1 in the interval where the global memory kernel is used, confirming that the execution times of the WF and NWF kernel in that interval increase at a similar rate.

The results from the same experiments on hardware configuration 1 shows similar behaviour in the shared memory interval when varying l, and in the very beginning where p is varied. The speedup has a similar shape, and, in the experiment where l is varied, is of approximately the same magnitude. However, as p is increased, the speedup of WF over NWF quickly converges to 1.

For hardware configuration 1, the WF execution time as function of n oscillates a lot starting at $n \approx 3,000,000$. This is also shown in the associated speedup figure (6.18a. It is therefore hard to tell whether or not the WF kernel time is faster than the NWF kernel in this case. The oscillations seem to stem from the WF kernel execution time, as it oscillates more than the execution time of the NWF kernel.

Slowdown of WF over NWF in the global memory kernel

In the global memory kernel interval of Figure 6.7a, we can see that the WF kernel is *slower* than the NWF kernel over the entire interval, and the difference seems to be constant as l is increased. As a consequence of the seeds being reorganized, the memory access pattern of the WF kernel differs from the NWF kernel, which can lead to different cache behaviour.

Experimental testing revealed that if sufficient time had passed since the last execution of the NWF kernel, running the NWF kernel resulted in the execution time being larger than that of the WF kernel. As an example, the first time the NWF kernel was run using $l = 20000$, $p = 0.01$ and $n = 20,000,000$, it finished executing in

749.68ms. When running it again immediately after, the execution time of the kernel was 685.08ms. Using the same method for WF bootstrap results in 747.69ms and 743.81ms respectively. Note that the entire application was run in all cases, not just the kernels.

Because the seeds to iterations file must be loaded when WF bootstrap is run, there is a much longer time between the subsequent WF kernel executions than between the NWF executions. If the caches of the GPU are cleared in the time it takes to load the seeds to iterations file, the behaviour we see may occur. This may also be the reason NWF bootstrap is slower the first time it is run after some time. However, we have been unable to confirm that this is actually the case.

Workload distribution as a predictor for speedup

The speedup of WF over NWF is shown together with the coefficient of variation and the speedup as estimated by dividing the warp execution efficiency of WF over the warp execution efficiency of NWF. The speedup and estimated speedup have the same shape when plotted as a function of p, and the Pearson correlation coefficient reflects that. For the case of varying l, in the interval where the shared memory kernel is used, the correlation coefficient is very high in the case of hardware configuration 2, and a bit lower in the case of hardware configuration 1. The speedup of the latter seems to have been affected by noise, and we suspect that the underlying function has the same shape as for the other hardware configuration, and thus a higher correlation coefficient.

As was discussed in Section 7.3.1, the coefficient of variation seems to be a good predictor for the warp execution efficiency of the NWF kernel. Further, dividing the warp execution efficiency of the WF kernel by the warp execution efficiency of the NWF kernel seems to be a good predictor for the resulting speedup when the shared

memory kernel is used. Assuming that the previous statements hold true, then by the transitive property of equality, the coefficient of variation of the workload is a good predictor for the speedup that can be achieved by reorganizing the workload.

7.3.4 Feasibility of reorganizing the workload

Figure 6.9b shows a dotted vertical line that represents the point where the execution time of the WF kernel including overhead is equal to the execution time of the NWF kernel including overhead. For $l = 12000$, $n = 20,000,000$ and $p > 0.05$ the total execution time is therefore decreased by reorganizing the workload before execution on the GTX 480, if the time to load the seeds to iterations file is disregarded. If the load time is included, there is no benefit from reorganizing the workload.

None of the experiments that were ran on the hardware configuration with the GTX 1080 Ti resulted in a speedup in the total execution time by reorganizing the workload. As noted previously, the overhead of reorganizing the workload is mostly invariant to the performance of the GPU. The absolute reduction in the execution time of WF over NWF decreases as the performance of the GPU increases, while the overhead is more or less the same. This means that as a faster GPU is used, the higher the coefficient of variation of the workload must be for the reduction in execution time to exceed the overhead of reorganizing the workload.

For workloads with a higher coefficient of variation a larger speedup is expected, as seen in Section 6.1.2. The absolute difference in execution time increases as the execution time of the kernels increases, so it is expected that for workloads that have a high coefficient of variation, *sufficient* execution time for each data point, and *sufficiently* low overhead ratio, that reorganizing the workload is beneficial with respect to the total execution time.

7.4 Task graph

7.4.1 Execution path selection

Table 6.3 and 6.4 show that the task graph optimizer is able to select the optimal path with an accuracy of over 96% for both hardware configurations. Although a very high accuracy, the results must be seen in the light of the total execution times (see Section 6.1.2). For hardware configuration 1, the time of the kernels including the overhead shows that the execution time of the warp-friendly bootstrap is much larger than the execution time of the non warp-friendly bootstrap. Only for very low n does the absolute value of the difference seem to be small. Because the difference is so large for most parameters, selecting the optimal path should be trivial.

For both hardware configurations, execution path 1 was the optimal path when $l = 12000$, $p = 0.1$ and $n = 1000$. The measured (not estimated by the task graph) execution times for path 1 for those parameters were 2.72ms and 4.24ms for hardware configuration 1 and 2, respectively. The execution times of path 3 were 4.34ms and 4.51ms. Execution path 1 is clearly the faster path for both hardware configurations. We suspect, since the correct path was selected by hardware configuration 2, that the initial training of the task graph of hardware configuration 1 may have been subject to noise.

In the case of hardware configuration 2, the task graph optimizer is able to select the optimal path in 96.6% of the executions. The one time it selected path 3 when it was not the optimal path, was for the parameters $l = 12000$ and $p \approx 0.043$. At that point, the execution times are very similar, as shown in Figure 6.8b, so selecting the wrong path does not have a large impact on the overall execution time. The other failure was when path 3 was the optimal but path 4 was chosen, for the execution with $p = 0.2$. In this case, the measured execution time were 8372.63ms for path 3

and 8518.82ms for path 4. The difference in execution times is so large that we would expect the task graph optimizer to be able to select the optimal path. The optimizer was able to select the optimal path correctly for all tested values of $p \in [0.05, 0.19]$, so it is reasonable to expect that an error at selecting the optimal path for $p = 0.2$ - which should have an even larger difference in execution times (see Figure 6.14b) - is because of a noisy sample when the task graph was trained.

7.4.2 Time usage

The time used for determining the optimal execution path is in most cases very low. For hardware configuration 2, the maximum time is almost double that of the minimum time, suggesting a high variation in the time used for finding the optimal execution path. In some circumstances this can be bad, as having a predictable execution time is useful when determining whether to run the task graph optimizer to find the best path, or to avoid the overhead and simply run an execution path that is known to have decent performance for all inputs (if such a path exists).

7.5 Load balancer

7.5.1 Optimization time and load balancing result

The time used by the load balancer for each of the 40 executions it was tested on can be seen in Section 6.3.1. The number of iterations of the optimizer is well below the maximum number of iterations for all executions. This means that the load balancer was able to find a partitioning that reduced the optimization gap to less than the threshold. This is also shown in Figure 6.20 and 6.22.

In Section 6.3.5 the execution times with and without load balancing between

the CPU and GPU is shown. With a speedup of up to 233.79 for the GPU over the CPU, it is very important that the partitioning is extremely precise so that the CPU is not assigned too much work, thus increasing the execution time instead of lowering it. For each extra workload Δn that is assigned to the CPU, the increase Δt_{cpu} is much larger than the Δt_{gpu} for the same increase of workload to the GPU. In Table 6.6, we can see that the minimum, maximum and median execution time when using the load balancer is greater than the execution time of just using the GPU. This is because the CPU was assigned too much work, thus increasing the total execution time. Including the time used by the load balancer yields even worse results.

A good strategy to combat this problem would be to overestimate the execution time of the slower device on purpose, to ensure that it will not become the bottleneck of the execution. Assigning a larger workload than the optimal to the device with the most performance will only lead to a slight increase in the execution time. Doing the same for the slower device will have a much larger impact, especially when the speedup of the faster device over the slower is as large as in this case.

For hardware configuration 2, the minimum execution time of balanced execution including the load balancer overhead is less than that of the *All GPU* configuration. The speedup of the GPU over the CPU in this case is much lower than in the previously discussed case, so transferring work from the GPU to the CPU has a larger effect on the GPU execution time. The speedup of the *Balanced incl. optimization* configuration over the *All GPU* configuration is $\frac{4310.31}{4304.35} = 1.001$, or an absolute difference of less than 6ms. The maximum and median of the balanced execution is in both cases larger than that of the *All GPU* configuration, and the variability of balanced execution is much larger than that of the *All GPU* configuration. This increased variability is a result of the large variability of the CPU execution time and the variability of the partitioning as found by the load balancer. We conclude that

running the load balancer may have a slight benefit, but is likely to increase the overall execution time.

7.5.2 Actual execution time

The goal of the load balancer is to find a partitioning that results in a good balance of the *estimated* execution times of each device. If the estimates are perfect, this would lead to a good balance of the actual execution as well. The figures in Section 6.3.4 show that this is not always the case. The first execution of both hardware configurations show a particularly bad balance, with a ratio of approximately 0.6 in both cases. To explain this result, two figures of the estimator of the CPU bootstrap module was included in the results. The first figure shows the state of the estimator when it was used by the load balancer to get an estimate of the execution time of the CPU module, and the second shows the state after it has been updated with the result of the execution. We can see that the estimator overestimated the execution time, which lead to the load balancer finding a partitioning that resulted in unbalanced execution times.

As a result of the CPU bootstrap estimator being updated with the new sample, the estimator provides a much better estimate for the second execution, and the load balancer is able to achieve an actual ratio of 0.95 or better for both hardware configurations. For subsequent executions, the ratio is above 0.9 for all but 3 executions using hardware configuration 1, and at or above 0.95 for all but 2 executions using hardware configuration 2. At execution 31 for both configurations, we see the same problem as in the first execution, only this time it was the estimators of the task graph that overestimated the execution time of the GPU. The estimators are then updated, and a better result is seen in the subsequent executions.

7.6 Estimating the reduction in execution time

We propose a method for estimating the absolute difference in execution times of the WF and NWF kernels based on the distribution of the workload. Here, we assume that n is constant and very large, to ensure that the warp execution efficiency of the WF kernel is close to 100%, as discussed in Section 7.3.1. Based on the observation of the correlation between the coefficient of variation and the speedup, we assume that the following relation is true:

$$s(l,p) = ac_{\text{v}}(l,p) = a\frac{\sigma_{\text{iter}}(l,p)}{\mu_{\text{iter}}(l,p)}, \tag{7.1}$$

where a is a coefficient that is dependent on the performance of the device used, and c_v is the coefficient of variation of the workload. μ_{iter} is the expected number of windows that must be drawn to fully resample the time series, or the number of iterations of the loop in the bootstrap kernel, and σ_{iter} is the standard deviation. We then define

$$t_{\text{nwf}}(l,p) = c\mu_{\text{nwf}}(l,p), \tag{7.2}$$

and

$$t_{\text{wf}}(l,p) = c\mu_{\text{wf}}(l,p), \tag{7.3}$$

where c is the execution time of each iteration in the loop in the bootstrap kernel and therefore device dependent, and $t_{\text{nwf}}(l,p)$ is the average number of iterations for the NWF kernel, **including** inactive threads. μ_{nwf} will be larger than μ because the number of iterations of a warp is equal to the number of iterations of the thread with the most iterations. As shown in Section 3.2.2, μ_{wf} will be equal to μ_{nwf} or lower. When the warp execution efficiency is 100%, $\mu_{\text{wf}} = \mu_{\text{iter}}$ because no thread is idle.

The difference in execution times is defined as

$$\Delta t(l,p) = t_{\text{nwf}}(l,p) - t_{\text{wf}}(l,p) \tag{7.4}$$

Using the assumption in Equation (7.1) and the definition of speedup, we get the following equation:

$$\frac{t_{\text{nwf}}(l,p)}{t_{\text{wf}}(l,p)} = a\frac{\sigma_{\text{iter}}(l,p)}{\mu_{\text{iter}}(l,p)} \tag{7.5}$$

By separating t_{wf} in Equation (7.5) and inserting into Equation (7.4), we get

$$\Delta t(l,p) = (a\frac{\sigma_{\text{iter}}(l,p)}{\mu_{\text{iter}}(l,p)} - 1)\, t_{\text{wf}}(l,p) \tag{7.6}$$

$$\Delta t(l,p) = ca\frac{\sigma_{\text{iter}}(l,p)\mu_{\text{wf}}(l,p)}{\mu_{\text{iter}}(l,p)} - c\mu_{\text{wf}}(l,p) \tag{7.7}$$

Now, assuming that the warp execution efficiency is 100%, which, as seen in Section 7.3.1 should be a reasonable assumption, we set $\mu_{\text{wf}}(l,p) = \mu_{\text{iter}}(l,p)$:

$$\Delta t(l,p) = ca\sigma_{\text{iter}}(l,p) - c\mu_{\text{iter}}(l,p) \tag{7.8}$$

Replacing the constant product ca by introducing a new constant, $\overline{c}$:

$$\Delta t(l,p) = \overline{c}\sigma_{\text{iter}}(l,p) - c\mu_{\text{iter}}(l,p) \tag{7.9}$$

We now have two constants that are dependent on the device used, $\overline{c}$ and c. To determine the values of these constants, we need to measure the difference in execution time for two different values of l or p. To be able to compare our estimate to the actual difference, we keep one of the parameters fixed while letting the other vary. We add a subscript i to the functions to denote that the variable is the value

of the function in point i, where i represent a pair of l and p. *E.g.*, $\mu_{\text{i,iter}}$ means the expected number of windows in point i. Then, by measuring Δt for two i, we get

$$\Delta t_1 = \overline{c}\sigma_{1,\text{iter}} - c\mu_{1,\text{iter}} \tag{7.10}$$

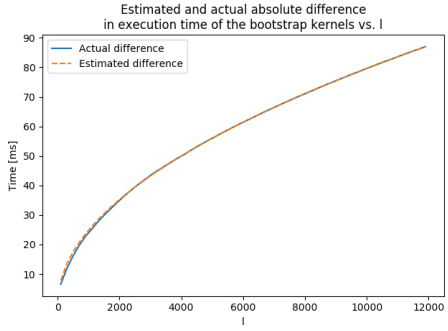
$$\Delta t_2 = \overline{c}\sigma_{2,\text{iter}} - c\mu_{2,\text{iter}} \tag{7.11}$$

We now have two equations with two unknowns, and can solve for $\overline{c}$ and c:

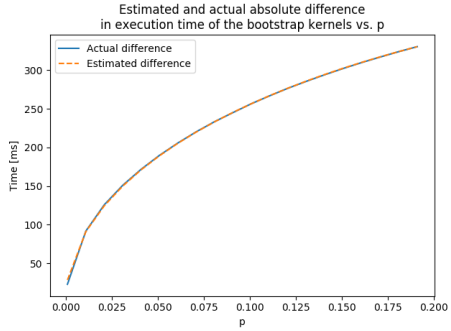
$$\overline{c} = \frac{t_2 - \frac{t_1\mu_2}{\mu_1}}{\sigma_2 - \frac{\sigma_1\mu_2}{\mu_1}} \tag{7.12}$$

$$c = \frac{\overline{c}\sigma_1 - t_1}{\mu_1} \tag{7.13}$$

To evaluate this approach, we compare the estimate from Equation (7.9) to the experimental results of hardware configuration 2.



(a) Estimated and actual difference in execution time vs. l.



(b) Estimated and actual difference in execution time vs. p.

We can see that Equation (7.9) provides a very good estimate of the difference in execution times. The good fit of the estimate indicates that our assumption in

Equation (7.1) may be correct. This estimate can be used in conjunction with the overhead estimates to determine whether or not to reorganize the workload, using only two samples and knowledge of the distribution of the workload.

Chapter 8

Conclusion

In this thesis we have shown that reorganization of irregular pseudo-random workloads with the goal of reducing branch divergence may lead to lower execution times of kernels on the GPU, for all but very small workloads ($n \leq 32$). We have also shown that the overhead of doing so in some cases is less than the difference in execution times of the kernels, providing an overall speedup even if the overhead is accounted for. The coefficient of variation was shown to be correlated with the speedup that can be achieved, but only when the shared memory of the GPU is used. Additionally, a relation between the mean and standard deviation of the workload and the reduction in execution time of the kernel has been established. The relation provides very good estimates of the reduction in execution time by only sampling two points.

Using an exhaustive mapping from the seeds of pseudo-random number generators to the associated workload proved to be slow. In particular, loading the file from disk was very slow compared to the overall execution. However, adding addi-

tional memory to the hardware leads to significantly reduced load times when the file has been loaded very recently. Loading the workload associated with each seed in the mapping was also shown to be slow, and was in all cases the operation that contributed the most to the overall overhead of reorganizing the workload, not including the time to load it from storage.

We have shown that dividing a program into logical parts, providing different implementations for the parts, and using a task graph for selecting the optimal sequence of implementations to run, may work well. However, for small workloads the overhead of selecting this sequence proved to be large, making it unsuitable for such workloads. Further, the application this method was tested on proved to have few viable distinct paths, which warrants further research into more complex programs.

For the application and hardware configuration used in this thesis, it was shown that partitioning the workload between the CPU and GPU requires a fast load balancer and very accurate estimators to reduce the execution time. Because of the very high performance of the GPUs relative to the CPU, scheduling a workload greater than the optimal to the CPU leads to an increase in the execution time, rather than a decrease.

Chapter 9

Future work

This project has done the groundwork for reorganizing the workload of PRNGs to achieve higher performance on GPUs. While some topics have been covered, there are still a lot of opportunities for further research into the area. Some of the directions that has been discovered, but not covered in this project due to resource limitations, are described here.

9.1 Static seeds

In this work, the seeds to the PRNG number generators is selected randomly each time the program is run. The overhead of loading the seeds to iteration counts file, sorting the seeds based on the associated iterations, and transferring the seeds to the GPU can be very time consuming. Using the same set of seeds, that has been reorganized beforehand, will avoid this overhead and potentially lead to a large speedup. The downside of this method is that the statistical properties of the program will be

changed, and possibly yielding an invalid bootstrap implementation.

Analyzing the statistical properties of using fixed seeds for the PRNGs in stationary bootstrap implementations is important for determining if this is a viable method or not.

9.2 Persistent seeds to iterations mapping on GPU

The overhead of generating random seeds on the GPU, transferring them to CPU memory, loading corresponding iteration counts and transferring them to GPU memory is, in all test cases in this thesis, high. Current GPUs have a lot of global memory. For instance, the GTX 1080 Ti has 11 GiB of memory, and the NVIDIA Tesla P100 [NVI17b] has 16 GiB of memory. For the latter, 8 GiB could be used for persistently storing the seeds to iterations mapping in the same format as in this project, leaving 8 GiB for other uses. The overhead of reorganizing seeds would then be reduced to generating the seeds and sorting them on the GPU / accelerator[1], which could yield better performance than the method used in this project.

9.3 Force uniform workload per warp

As seen in figure 3.4b, the method for reorganizing the workload used in this project may still lead to branch divergence. There is another approach, which is grouping threads into warps so that all threads in a warp have the same amount of work. Doing this may lead to some threads being completely deactivated, and additional warps to be scheduled, but it may lead to greater performance if the original amount of warps is low, or if the variance of the work load in the warp is large enough.

[1] The NVIDIA Tesla series are GPUs without outputs for computer displays.

9.4 Other applications

In this project only the stationary bootstrap application has been used for experimentation. Testing applications with workloads that have a larger coefficient of variation would be interesting to validate the results of reorganizing the workload in this project.

Another interesting approach would be to implement the task graph for an application where the operations are highly non-linear. Especially use cases where the optimal execution path changes frequently with respect to the inputs would be interesting to analyze for validating the approach of using a task graph to optimize the execution path.

9.5 Compression of seeds to iteration counts file

The seeds to iterations mapping in this project uses a fixed size of 32 bits per element. This wastes a lot of disk and memory space, especially when all the iteration counts are small enough to be represented by half, or even a quarter, the number of bits. The iteration counts can be analyzed and a smaller data type can be used if applicable, which should lead to significant savings in both disk and memory usage, but also disk load and iteration counts load time. Sophisticated compression algorithms could also potentially be useful, to shrink the size even more.

9.6 Scheduling large workloads on the CPU

If the same PRNG with the exact same parameters had been used on the CPU and GPU, seeds that lead to a large workload could be scheduled for execution on the CPU rather than the GPU. This could lead to better performance due to threads not

having to wait on the large workloads to finish. In this project, we have been unable to do so because of different PRNGs being used on the devices.

This method of cherry picking seeds that lead to large workloads for execution on the CPU could potentially be used without reorganizing the workload. Generating the seeds, and for each group of 32 consecutive seeds picking seeds that contribute the most to the variance within that warp and executing them on the CPU, could lead to better performance without having to sort the seeds.

9.7 Load balancer improvement

Currently, the load balancer starts at the same point every time. This could be improved by keeping track of previous solutions to similar inputs, and using that as a heuristic to achieve a better initial point.

The gradient descent solver that is currently implemented is vulnerable to local minima. For optimization problems with a higher degree of non-linearity than the one featured in this thesis, other solvers may be more able to find a good solution.

Bibliography

[98] "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator". In: *ACM Transactions on Modeling and Computer Simulation* 8.1 (1998), pp. 3–30. URL: http://portal.acm.org/citation.cfm?doid=272991.272995.

[Aug11] Cedric Auggonet. "Scheduling Tasks over Multicore machines enhanced with Accelerators : a Runtime System's Perspective". In: (2011).

[Boo72] Carl de Boor. "On Calculating with B-Splines". In: *Journal of Approximation Theory* 6 (1972), pp. 50–62. URL: http://ac.els-cdn.com/0021904572900809/1-s2.0-0021904572900809-main.pdf?%7B%5C_%7Dtid=9bccff60-4b6c-11e7-8de2-00000aacb35f%7B%5C&%7Dacdnat=1496831730%7B%5C_%7Ddc5fa88116cd9dff9ac480646aa2d52e.

[Boy+13] Michael Boyer et al. "Load balancing in a changing world: dealing with heterogeneity and performance variability." In: *Cf* (2013), p. 1. DOI: 10.1145/2482767.2482794. URL: http://www.cs.virginia.edu/%7B~%7Dmwb7w/publications/CF%7B%5C_%7D13%7B%5C_%7Dload%7B%5C_%7Dbalancing.pdf%7B%5C%%7D5Cnhttp://dl.acm.org/citation.cfm?doid=2482767.2482794.

[Che+10] Long Chen et al. “Dynamic Load Balancing on Single- and Multi-GPU Systems”. In: *Ipdps* (2010). ISSN: 1530-2075. DOI: 10.1109/IPDPS.2010.5470413.

[cpl17] cplusplus.com. *mt19937 - C++ Reference*. http://www.cplusplus.com/reference/random/mt19937/. Accessed: 2017-06-28. 2017.

[ECO] ECON 370. “More Time Series Analysis”. In: (), pp. 1–10. URL: http://people.stfx.ca/tleo/econ370term2lec6.pdf.

[Efr79] Bradley Efron. “Bootstrap Methods: Another Look at the Jackknife”. In: 7.1 (1979), pp. 1–26. URL: http://www.jstor.org/stable/2958830.

[FE16] Thomas L Falch and Anne C Elster. “ImageCL: An Image Processing Language for Performance Portability on Heterogeneous Systems”. In: (2016), pp. 1–16. DOI: 10.1109/HPCSim.2016.7568385. arXiv: arXiv:1605.06399v1.

[Gri+15] Bjarne Grimstad et al. *SPLINTER: a library for multivariate function approximation with splines*. http://github.com/bgrimstad/splinter. Accessed: 2017-04-20. 2015.

[GS16] Bjarne Grimstad and Anders Sandnes. “Global optimization with spline constraints: a new branch-and-bound method based on B-splines”. In: *Journal of Global Optimization* 65.3 (2016), pp. 401–439. ISSN: 1573-2916. DOI: 10.1007/s10898-015-0358-4.

[Hef86] Statistische Hefte. “Statistische Hefte Statistical Papers 9”. In: 6 (1986).

[IEE08] IEEE. *IEEE Standard 754-2008 for Floating-Point Arithmetic*. Vol. 2008. August. 2008, pp. 1–58. ISBN: 9780738157528. DOI: 10.1109/IEEESTD.2008.4610935. URL: http://ieeexplore.ieee.org/xpl/freeabs%

7B%5C_%7Dall.jsp?arnumber=4610935%7B%5C%%7D5Cnhttp://ieeexplore.ieee.org/servlet/opac?punumber=4610933.

[INC17] ASUSTeK COMPUTER INC. *CROSSHAIR VI HERO*. http://dlcdnet.asus.com/pub/ASUS/mb/SocketAM4/CROSSHAIR-VI-HERO/E12601_CROSSHAIR_VI_HERO_UM_V3_WEB.pdf. Accessed: 2017-06-22. 2017.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0-201-89684-2.

[Lin89] Lawrence I-Kuei Lin. "A Concordance Correlation Coefficient to Evaluate Reproducibility". In: *Biometrics* 45.1 (1989), pp. 255–268. DOI: 10.1002/0471667196.ess0146.pub2. URL: http://www.jstor.org/stable/2532051.

[Loh17] Niels Lohmann. *nlohmann::json: JSON for Modern C++*. https://github.com/nlohmann/json. Accessed: 2017-06-22. 2017.

[Mon94] Universitd De Montreal. "Uniform random number generation". In: 53 (1994), pp. 77–120.

[NP94] Dimitris N. Politis and Joseph P. Romano. "The Stationary Bootstrap". In: *Journal of the American Statistical Association* 89.428 (1994), pp. 1303–1313.

[NVI17a] NVIDIA. *cuRAND :: CUDA Toolkit Documentation*. https://developer.nvidia.com/curand. Accessed: 2017-06-28. 2017.

[NVI17b] NVIDIA. *NVIDIA® TESLA® P100 GPU ACCELERATOR*. http://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-datasheet.pdf. Accessed: 2017-07-02. 2017.

[NVI17c] NVIDIA. *Programming Guide :: CUDA Toolkit Documentation*. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed: 2017-06-28. 2017.

[NVI17d] NVIDIA. *Thrust :: CUDA Toolkit Documentation*. http://docs.nvidia.com/cuda/thrust/index.html. Accessed: 2017-06-29. 2017.

[nvp17] NVIDIA. *nvprof metrics reference*. 2017. URL: http://docs.nvidia.com/cuda/profiler-users-guide/index.html#metrics-reference.

[Ref17a] CPP Reference. *Date and time utilities*. http://en.cppreference.com/w/cpp/chrono. Accessed: 2017-06-29. 2017.

[Ref17b] CPP Reference. *std::async*. http://en.cppreference.com/w/cpp/thread/async. Accessed: 2017-06-28. 2017.

[Ref17c] CPP Reference. *std::geometric_distribution*. http://en.cppreference.com/w/cpp/numeric/random/geometric_distribution. Accessed: 2017-06-28. 2017.

[Ref17d] CPP Reference. *std::uniform_int_distribution*. http://en.cppreference.com/w/cpp/numeric/random/uniform_int_distribution. Accessed: 2017-06-28. 2017.

[Rud16] Sebastian Ruder. "An overview of gradient descent optimization algorithms ". In: (2016), pp. 1–14. arXiv: arXiv:1609.04747v2.

[Sch07] Larry L Schumaker. *Spline Functions: Basic Theory*. 3rd ed. Cambridge University Press, 2007. ISBN: 9780521705127.

[She+13] Jie Shen et al. "Glinda : A Framework for Accelerating Imbalanced Applications on Heterogeneous Platforms". In: *Proceedings of the ACM International Conference on Computing Frontiers - CF '13* May (2013), p. 1.

DOI: 10.1145/2482767.2482785. URL: http://dl.acm.org/citation.cfm?doid=2482767.2482785.

[She+16] Jie Shen et al. "Workload Partitioning for Accelerating Applications on Heterogeneous Platforms". In: *IEEE Transactions on Parallel and Distributed Systems* 27.9 (2016), pp. 2766–2780. ISSN: 10459219. DOI: 10.1109/TPDS.2015.2509972.

[Shi54] Alfonso Shimbel. "Structure in communication nets". In: *Proceedings of the symposium on information networks*. Vol. 4. 1954.

Appendices

Appendix A

Figures

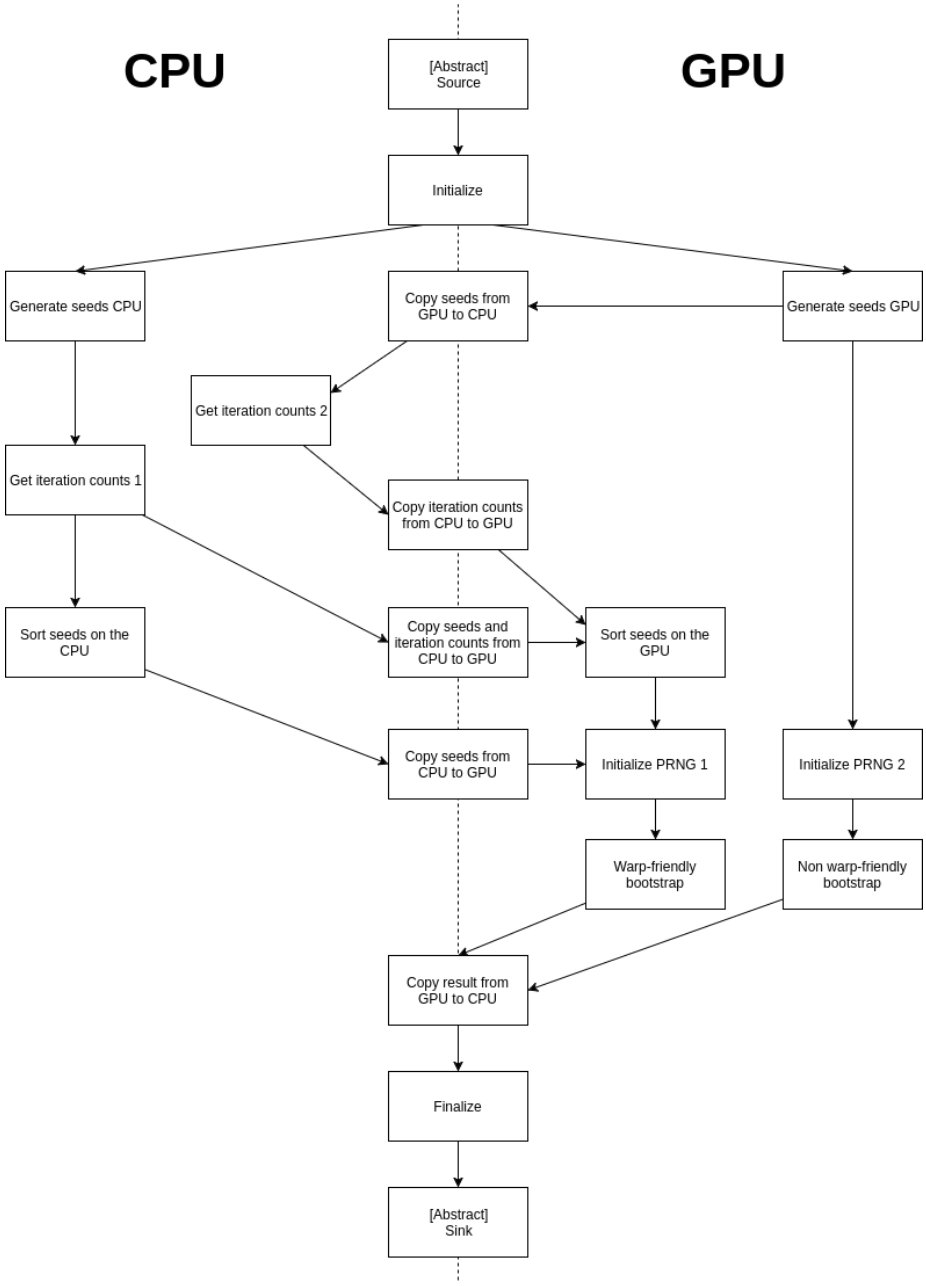


Figure A.1: Complete task graph of the bootstrap program.

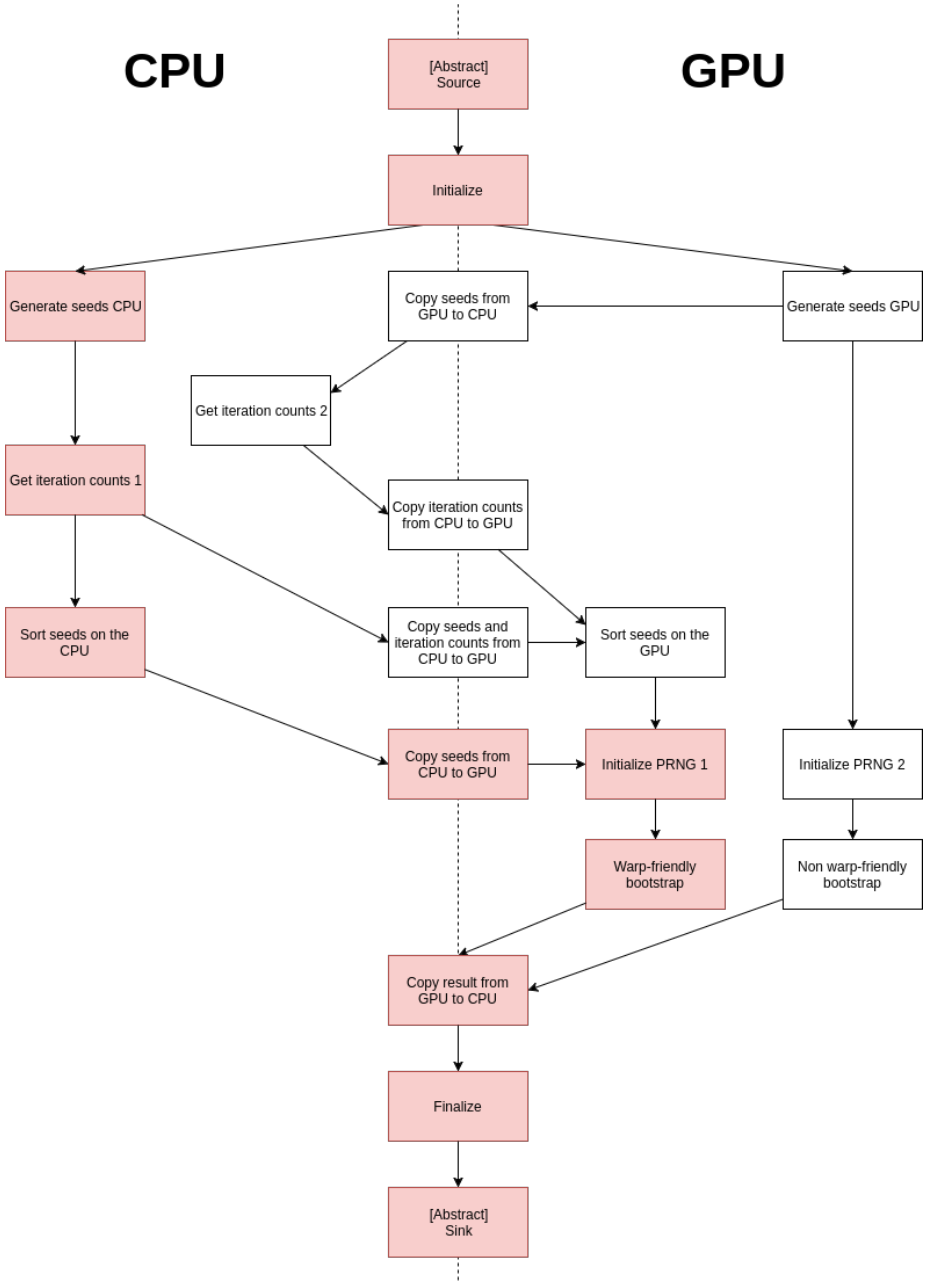


Figure A.2: Execution path 1/4.

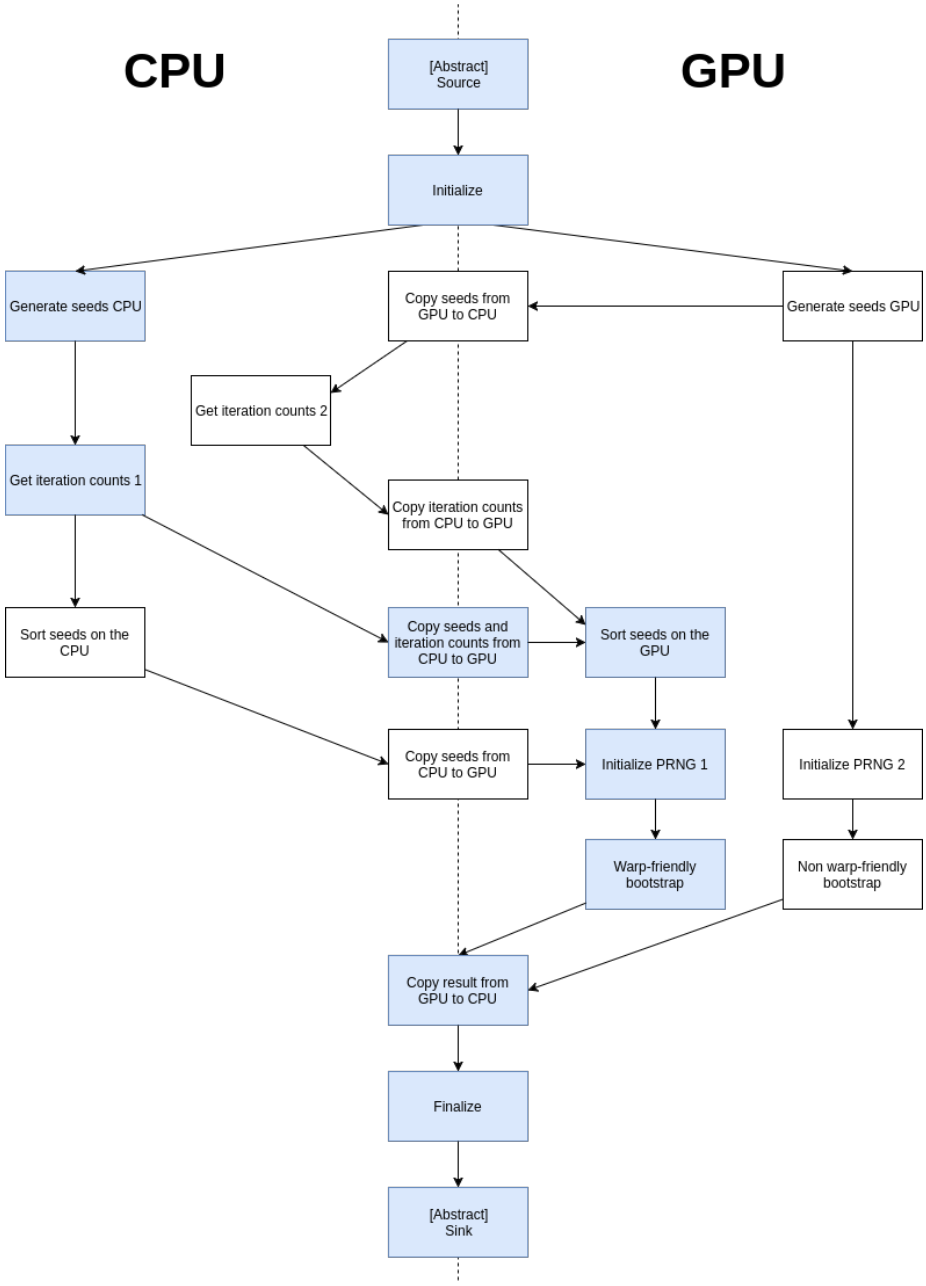


Figure A.3: Execution path 2/4.

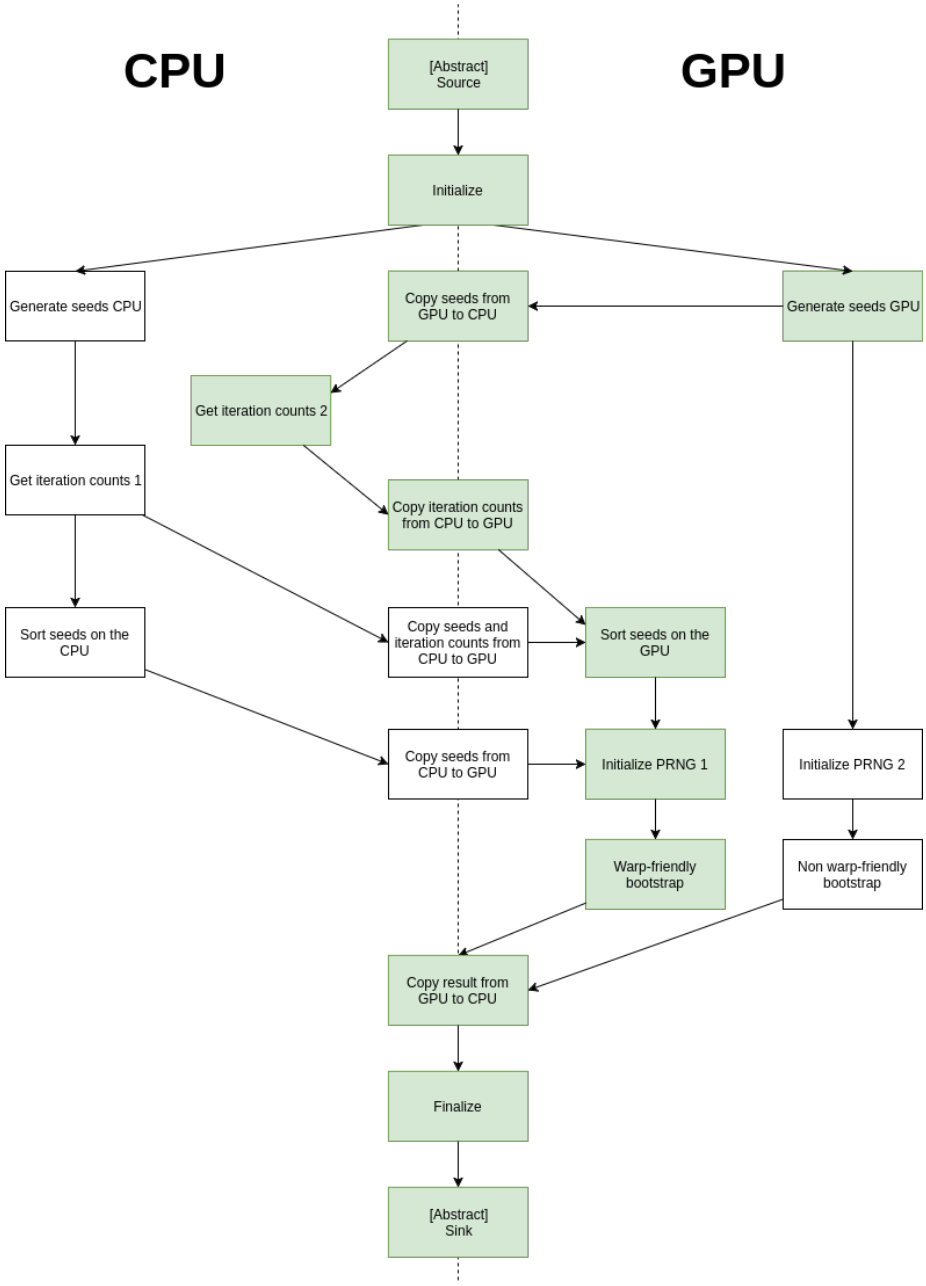


Figure A.4: Execution path 3/4.

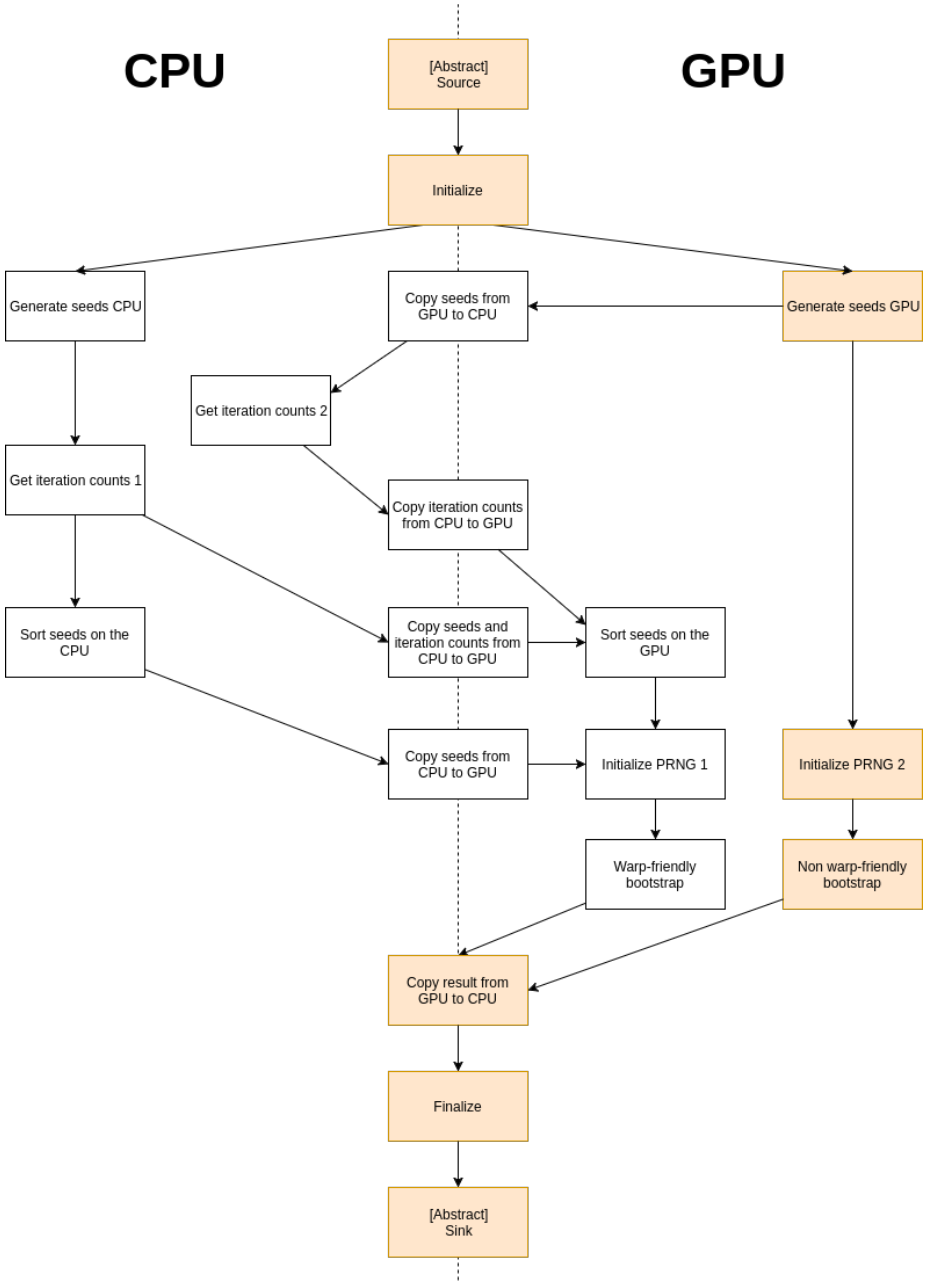


Figure A.5: Execution path 4/4.

Appendix B

Source code listings

```
__global__ void _bootstrap_block(float *cumulativeInput, int
    cumulativeInputSize, float *out, int nBootstrapIterations,
    curandState_t *states, float p) {
    int block_id = blockIdx.y * gridDim.x + blockIdx.x;
    int id = block_id * blockDim.x * blockDim.y + threadIdx.y *
        blockDim.x + threadIdx.x;

    extern __shared__ float c[];

    int inputSize = cumulativeInputSize - 1;
    int n_threads_block_dim_x = min(blockDim.x,
        nBootstrapIterations);
    int n = cumulativeInputSize / n_threads_block_dim_x;
    if (cumulativeInputSize % n_threads_block_dim_x != 0) {
        n++;
    }
    for (int i = 0; i < n; ++i) {
        int idx = threadIdx.x + n_threads_block_dim_x*i;
        if (idx < cumulativeInputSize) {
            c[idx] = cumulativeInput[idx];
        }
    }

    __syncthreads();

    curandState_t *randState = &states[id];

    if (id < nBootstrapIterations) {
        float sum = 0.0f;

        for (int available = inputSize; available > 0; ) {
            int start = _uniform(randState, 0, inputSize);
            int block_length = min(_geometric(randState, p),
```

```
                    available);
30              available -= block_length;
31              int end = start + block_length;
32              int end_clamped = min(end, inputSize);
33              int end_wrapped_around = max(0, end - inputSize);
34              sum += c[end_clamped] - c[start] + c[
                    end_wrapped_around];
35          }
36          out[id] = sum / inputSize;
37      }
38  }
```

Listing B.1: Bootstrap kernel using shared memory

```
__global__ void _bootstrap_block_long(float *cumulativeInput,
    int cumulativeInputSize, float *out, int
    nBootstrapIterations, curandState_t *states, float p) {
    int block_id = blockIdx.y * gridDim.x + blockIdx.x;
    int id = block_id * blockDim.x * blockDim.y + threadIdx.y *
        blockDim.x + threadIdx.x;

    int inputSize = cumulativeInputSize - 1;
    curandState_t *randState = &states[id];

    if (id < nBootstrapIterations) {
        float sum = 0.0f;

        for (int available = inputSize; available > 0; ) {
            int start = _uniform(randState, 0, inputSize);
            int block_length = min(_geometric(randState, p),
                available);
            available -= block_length;
            int end = start + block_length;
            int end_clamped = min(end, inputSize);
            int end_wrapped_around = max(0, end - inputSize);
            sum += cumulativeInput[end_clamped] -
                cumulativeInput[start] + cumulativeInput[
                end_wrapped_around];
        }
        out[id] = sum / inputSize;
    }
}
```

Listing B.2: Bootstrap kernel for long time series

```
__global__ void _bootstrap_block_iteration_counts(int
    cumulativeInputSize, int *out, int nBootstrapIterations,
    curandState_t *states, float p) {
    int block_id = blockIdx.y * gridDim.x + blockIdx.x;
    int id = block_id * blockDim.x * blockDim.y + threadIdx.y *
        blockDim.x + threadIdx.x;

    int inputSize = cumulativeInputSize - 1;
    curandState_t *randState = &states[id];

    if (id < nBootstrapIterations) {
        int iter = 0;
        for (int available = inputSize; available > 0; iter++) {
            int start = _uniform(randState, 0, inputSize);
            int block_length = min(_geometric(randState, p),
                available);
            available -= block_length;

            // Note to reader: Removing these next three lines
                results in the iteration count being wrong. We
                suspect an incorrect compiler optimization to be
                 the culprit. The lines are left here for the
                sake of reproducibility
            int end = start + block_length;
            int end_clamped = min(end, inputSize);
            int end_wrapped_around = max(0, end - inputSize);
        }
        out[id] = iter;
    }
}
```

Listing B.3: Kernel for getting iteration counts

```
__global__ void _generate_seeds(int global_seed, int *out, int
    n_iterations) {
    int block_id = blockIdx.y * gridDim.x + blockIdx.x;
    int id = block_id * blockDim.x * blockDim.y + threadIdx.y *
        blockDim.x + threadIdx.x;

    if (id < n_iterations) {
        curandState_t rand_state;
        // Intentional unsigned integer under/overflow
        curand_init((unsigned int) (global_seed + id), 0, 0, &
            rand_state);
        out[id] = (int) (curand(&rand_state) / 2);
    }
}
```

Listing B.4: Generate seeds on the GPU

```
std::vector<float>
CpuDevice::run(std::vector<float> cumulativeInput,
               int nIterations,
               float p) {
    int inputSize = (int) cumulativeInput.size() - 1;

    auto avgs = std::vector<float>((unsigned long) nIterations);
    std::uniform\_int\_distribution<int> uniformIntDistribution
        (0, inputSize - 1);
    std::geometric\_distribution<int> geometricDistribution(p);

    for (int i = 0; i < nIterations; i++) {

        float tot = 0;
        for (int available = inputSize; available > 0; ) {
            // Draw interval start
            int start = uniformIntDistribution(randomGenerator);
            // Draw interval length
            int blockLength = std::min(available,
                geometricDistribution(randomGenerator));
            // Calculate interval end
            int end = start + blockLength;
            // Clamp at input size
            int endClamped = std::min(end, inputSize);
            // The portion that would be wrapped around
            int endWrappedAround = std::max(0, end - inputSize);
            // Interval is now [start, endClamped) + [0,
                endWrappedAround)
            available -= endClamped - start + endWrappedAround;
            tot += cumulativeInput[endClamped] - cumulativeInput
                [start] + cumulativeInput[endWrappedAround];
        }
        avgs[i] = tot / inputSize;
```

```
30      }
31      return avgs;
32  }
```

Listing B.5: C++ implementation of the stationary bootstrap

Appendix C

Running the bootstrap program

C.1 Compilation

Dependency name	URL
Ubuntu 16.04	ubuntu.com
CUDA v8.0	nvidia.com
SPLINTER @ d473e6	github.com

Download Ubuntu and CUDA using the provided URLs, and install Ubuntu 16.04. Then, to install the required C++11 toolchain, and git (for downloading SPLINTER), run:

sudo apt-get update

sudo apt-get install build-essential git cmake

Install CUDA by running

sudo dpkg -i cuda-repo-ubuntu1604-8-0-local-ga2_8.0.61-1_amd64.deb

sudo apt-get update

sudo apt-get install cuda

Install SPLINTER by running

git clone https://github.com/bgrimstad/splinter.git

cd splinter

git checkout d473e6

mkdir build

cd build

cmake .. -DCMAKE_BUILD_TYPE=release

make -j$(nproc)

sudo make install

In the directory containing the Makefile of the bootstrap program, run

export PATH=$PATH:/usr/local/cuda/bin/

make -j$(nproc)

An executable called bootstrap will be created in the current directory. If you get an error saying libsplinter-3-0 cannot be found when running it, run

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib

and try again.

C.2 Running

You should now be able to run the bootstrap program, assuming you have a CUDA enabled device.

Examples:

./bootstrap -l=12000 -p=0.1 -n=40000000 –device-id=0 –output-file=1.json

./bootstrap -l=12000 -p=0.1 -n=40000000 –gpu-fraction=1.0 –output-file=2.json

./bootstrap -l=1000 -p=0.01 -n=20000000 –force-path=warp_friendly

For initially training the estimators, train_bootstrap.py can be used. This is only necessary if you want the task graph to have good estimates to begin with. Remember to update the path to the bootstrap program and the l and p parameters. Run
./bootstrap -l=$l -p=$p -n=1
before running train_bootstrap.py, so the seeds to iterations mapping can be generated.