Optimisation of Gaussian Process Regression Implementations for Integrated GPUs

Andre Nicholas Bododea

4th Year Project Report
Electronics and Software Engineering
School of Informatics
University of Edinburgh

2017

Abstract

Gaussian process methods for regression problems are important in many fields, an important example of which is the field of autonomous vehicles. These vehicles need to make predictions in a mobile environment based on observed inputs, and many do this by using some form of Gaussian process regression. By examining the theory behind and then subsequently profiling an implementation of a Gaussian process regression, we pinpoint the section that takes up a large majority of program execution time and then attempt to exploit parallelism available by building a parallel triangular linear equation solver.

By optimising several key aspects of the implementation in a way that is specifically geared towards integrated GPUs, we gauge the feasibility of implementing a parallel triangular solver on such a lightweight mobile device within the larger context of speeding up the Gaussian Process regression. We also outline many of the key techniques for optimising parallel programs for integrated GPUs in particular and present important results for the platform in use for this project - the ARM Mali T604 GPU. Ultimately, it appears that the parallel Gaussian process regression is in fact not faster than the serial implementation for this platform for a number of reasons.

An argument is presented that testing of the implementation developed in this paper on stronger and more recent integrated GPUs, or alternatively porting the implementation to other parallel programming frameworks, is likely to ultimately lead to the desired outcome of a faster Gaussian process regression by means of parallelisation.

Acknowledgements

To my supervisor Björn Franke, who consistently went above and beyond his role as supervisor.

To my family for always being there for me on this journey. I am profoundly grateful for each of you.

Table of Contents

Chapter 1

Introduction

The area of supervised learning can be divided into two main categories: classification and regression problems. Classification is the process of identifying to which category a new observation belongs on the basis of training data. Regression, on the other hand, is concerned with the prediction of continuous quantities, and is used frequently as a basis for many predictive algorithms. This paper will focus on regression, more specifically Gaussian process regression. Gaussian process (GP) regression works by approximating the underlying distribution of the variables under observation by using a set of Gaussian distributions.

Practical applications of GP regressions include estimating cost of movement when planning a trajectory for mobile robots [1], modeling large-scale terrain that can adequately handle uncertainty and incompleteness [2], and real-time segmenting of three-dimensional scans of various terrains for Autonomous Land Vehicles [3]. In situations where GP regressions must train and perform predictions in real-time, an optimised implementation can be the difference between an autonomous vehicle quickly predicting and responding to difficult terrain versus making slow decisions that could be damaging or even potentially fatal to passengers.

With the latest advances in General Purpose Graphics Processing Unit (GPGPU or just GPU) technology producing smaller and faster GPUs with hundreds of cores, exploiting parallelism in various domains has never been more accessible than it is today. Exploring how this ability to parallelise fits into the Gaussian process regression and where it can aid in efficiently speeding up existing implementations, is the essence of this project.

1.1 Project aims

The goal of this paper is to explore the areas of the GP regression that take up the most computational time, eventually pinpointing these areas and boosting performance via writing programs that run in parallel on a GPU. Furthermore, this paper outlines the most computationally expensive steps in a GP regression, shows how these steps can be modified and ported to OpenCL (a parallel programming language that allows code to run on a GPU), and finally describes optimisations for these various parallel programs that specifically target lightweight integrated GPUs. An analysis of these programs will then be presented and will culminate in a discussion of the results, as well as further investigation that could be conducted on the basis of these results.

The specific goals for this project are the following:

- Analyse GP regression implementation by profiling existing code
- Identify areas of the code that take up most of the computation time via profiling
- Parallelise those sections where possible
- Analyse performance of parallel programs
- Apply optimisations to the parallel code to achieve optimal results for our target platform
- Discuss the meaning of the various results produced and outline further research that can build on results gathered

1.2 Report outline

Background This chapter will provide an introduction to the central concepts of the project. These include descriptions of the hardware, an explanation of the implementation that this project is based on, an introduction to OpenCL and parallelisation, and the theory behind Gaussian Process regression.

The Gaussian Process Regression This chapter will focus on a mathematical explanation of the Gaussian Process regression. Then it will address the section of the program most likely to consume large portions of execution time. Finally, this chapter concludes with an outline of how this baseline GP regression has been implemented within the code.

Profiling the Code As the two previous chapter summarised the theory, this chapter discusses methods for profiling the code in order to ascertain which sections take up the most execution time. By the end of this chapter, we will have isolated the part(s) of the GP regression implementation that take the most time, and what should be improved in order to increase the speed of these parts.

The Triangular Solver In the previous chapter we concluded that the triangular solve, and thus the Cholesky transform, is the most time consuming operation. This chapter will detail how we will parallelise this implementation and then provide a brief mathematical overview of the serial version of the triangular solver. Finally it will show how we build the serial implementation in C, leading directly into the next chapter where this will then be parallelised via OpenCL.

Triangular Solve For GPU This chapter will focus on the process of parallelisation, where we transform the C program presented in the previous chapter to OpenCL, in order for parallel computation on the GPU to take place. It will also explain the concepts of kernel and host code, discuss various design decisions, and present initial timing results.

Efficient Data Transfer There are a number of ways to optimise OpenCL performance, however data transfer remains a primary obstacle when working with GPUs. This chapter focuses on making this as efficient as possible, discussing the two primary methods - copying and mapping of buffers. Timed tests were conducted to highlight the most effective approach at addressing this problem.

OpenCL Scheduling Overhead Upon discovering that mapping is the more efficient method, it is also important to consider another large source of overhead - the OpenCL scheduling model. This chapter discusses the contributions that the scheduling model makes towards program execution overhead and outlines various solutions, many of which are incompatible with the version of OpenCL supported on our GPU. Documentation and sources from the literature therefore lead us to conclude that the next-best technique available to us for improving overall execution time is vectorisation.

Vectorisation This chapter discusses the technique of vectorisation, where we take advantage of the vector registers available in our GPU. We discuss memory storage obstacles that must be eliminated in order for vectorisation to be possible, then give a general introduction to vectorisation, followed by an overview of how we vectorised the OpenCL triangular solver. Finally timing results are presented and discussed.

Discussion This chapter discusses where we have ended with optimisations and suggests further steps to be taken based on the results gathered in this paper.

Conclusion The conclusion circles back to the initial goals; discussing the aims that were achieved within our paper, the ones that were not able to be achieved, and the implications for our larger goal of optimising the GP regression.

Chapter 2

Background

Some background is needed to familiarise readers with the basic concepts used in this project. These include details of the hardware used, information about the pre-existing implementation on which this project is based, and a gentle introduction to transforming programs from C to OpenCL.

2.1 Development Hardware: The Arndale Development Board

All compilation and testing is done on an Arndale development board, equipped with a Cortex-A15 dual core CPU and, most importantly for this project, an ARM Mali T-604 GPU. This GPU along with similar ones from the Mali series are widely used in mobile phones and tablets. The Mali T-604 was used in the wildly popular Samsung Galaxy S5 flagship smartphone, Google Nexus 10 tablet, Samsung Chromebook and Chromebook 3, and various models of the HP Chromebook 11 [4]. Similar GPUs from the Mali family have been used in the Galaxy Note 3 and Galaxy Tab S [5], as well as the Samsung flagship Galaxy S6 [6]. The latest generation smartphones from big brands are still using latest Mali GPUs. The Samsung Galaxy S7, Samsung's latest flagship smartphone as of this paper, uses the Mali T-880 [7] while the latest flagship smartphone from Huawei, the Huawei P10, uses the Mali G71 [8].

Therefore this project can be carried over to these other GPUs which are similar and yet stronger than the T-604 in many aspects as they are generations newer. These GPUs also use a unified memory model [9], just like the T-604. This, among other similarities, mean that many of the techniques for design and optimisation presented in this paper are also generally applicable to other Mali GPUs and indeed most integrated GPUs in general when specific details are abstracted away.

2.2 The Codebase

The codebase was forked from a project by Manuel Blum at the University of Freiburg [10]. This implementation is based heavily on Rasmussen and Williams' seminal textbook *Gaussian Processes for Machine Learning* [11], therefore much of the theory discussed will be cited from this text and conventions from the text will be maintained here.

The codebase was ideally suited for this project, as the original Gaussian process regression implementation was developed in C++ and OpenCL integration with C++ is very good. C++ programs also tend to run quickly [12] and modern C++ makes memory management very straightforward for the most part, while also offering comprehensive facilities for more diverse memory operations. This makes it an ideal language for working with matrices and arrays, while also, allowing us to maintain tight control over the way data is stored in memory when needed.

This particular implementation also makes use of the Eigen library [13], which is a BLAS (Basic Linear Algebra Subprograms) library. Matrices and vectors are stored within `Eigen::Matrix` and `Eigen::Vector` objects, allowing matrix operations within the code to be performed using the various functions available from the Eigen library. This keeps matrix and vector operations quick and simple within the implementation.

2.3 Parallelisation with OpenCL

There are two main frameworks used for GPGPU programming - CUDA and OpenCL. OpenCL was chosen for this project as the Mali T-604 has support for OpenCL 1.1. OpenCL is a portable cross-platform language, while CUDA can only be used for NVIDIA devices. Thus this implementation can be used and duplicated on various GPUs. While OpenCL is portable, it does requires tweaks and platform-specific changes to perform well on any given GPU and is therefore not effective if simply ported without customisation.

The techniques presented in this paper are specific to integrated GPUs, and therefore must drop many of the techniques that are standard when writing and optimising parallel programs on most desktop or server GPUs. The same methods that are applied to the Mali T-604 device can be used for similar integrated GPUs, and indeed one of the aims of this project is to explain optimisations in a general enough way that with small changes, the code can be run on various integrated GPUs and results can be measured accordingly.

A basic example for taking C code and parallelising using OpenCL will be provided in this section for those who are not familiar with it. This will allow an inexperienced reader to follow along with the process of creating OpenCL code from a baseline implementation of our algorithm from Chapter 6 and after.

2.3.1 Parallelising A Basic C Program With OpenCL

In this example, let's consider a very basic C program that takes two input arrays as input, and returns an array. This output array at each index will be the sum of the values of the two input arrays at that index.

Assuming that the input arrays have 80 elements each, the program looks like this:

```c
int* add_two_arrays(int* input_arr_one, int* input_arr_two)
{
    // We assume that each input array has 80 elements
    int output_arr[80];

    // Add elements of input arrays, assign result to output_arr
    for(int i=0; i < 80; i++){
        output_arr[i] = input_arr_one[i] + input_arr_two[i];
    }

    return output_arr;
}
```

Listing 2.1: Example C program that adds two arrays

In our example, we have a loop that iterates 80 times. Notice that each iteration is independent of the last, e.g. each iteration does not require the result of the previous operation in order to correctly perform its operation. This means that there is parallelism that can be exploited, and we do this by writing an OpenCL program. In order to do this, we focus on the calculation being performed within the loop, which in this case is an addition.

Instead of looping 80 times, we simply set up a *kernel* that performs this addition. A kernel is a very lightweight program that runs on the GPU in parallel. Each GPU thread, in OpenCL a GPU thread is called a *work-item*, runs the kernel once. So if our GPU core has 10 work-items, then 10 instances of the kernel can be run in parallel at once. Our *work-group* size in this case, is therefore 8.

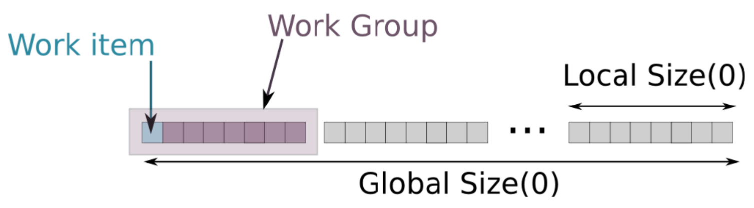


Figure 2.1: 1D Data-Parallel execution in OpenCL [14]

In this program, however, we want to call the kernel 80 times in order to complete the task. As this OpenCL program needs to run a kernel 80 times to complete its calculations, its *global work size* will be 80.

The kernel is written as the following:

```
__kernel void vec_example(__global int* input_arr_one,
                          __global int* input_arr_two,
                          __global int* output_arr)
{
  /*
   * The index gives the current work item number
   * In this example, it will go from from 0 to 80,
   * as we are calling the kernel 80 times
   */
    int i = get_global_id(0);

    // Performs the operation
    output_arr[i] = input_arr_one[i]+input_arr_two[i];
}
```

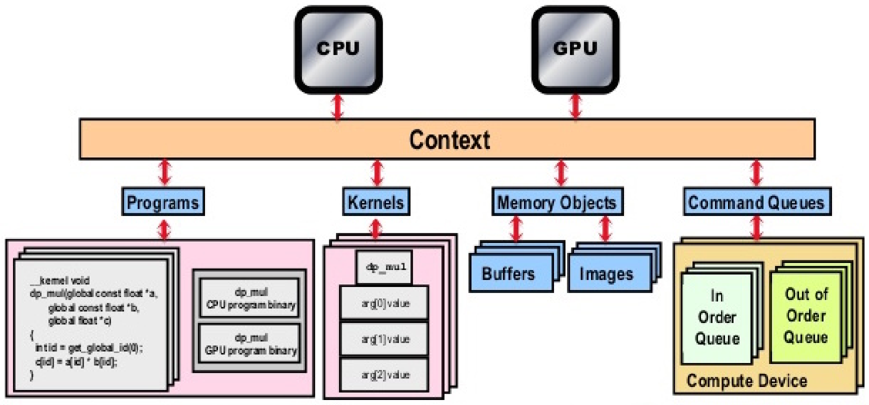
Listing 2.2: Toy OpenCL kernel

The `__kernel` decorator specifies that this is an OpenCL kernel, while the `__global` decorator specifies that the pointer passed to the kernel is referring to the global device memory space. Aside from these decorators, normal C syntax is used. Also, the kernel always has return type void. If the kernel above is run 80 times, we will achieve the same result as the serial C program but hopefully faster because we exposed the parallelism available in the program.

2.3.2 Running A Kernel In OpenCL

Now that we have a kernel that we would like to run, we need to understand how a kernel is run. We have already introduced the OpenCL kernel, however in order to use the kernel we need to call it from somewhere. This somewhere is the *host*. This is a program, written in C in this example's case, that calls the kernel. This program executes on the "host side", which is the CPU in our case. The CPU runs this program, and then this program will send kernels to the GPU.

Specific details of implementing an OpenCL host program will not be discussed for the sake of brevity, as these details can be found in any OpenCL textbook. This section will therefore only touch on the main aspects of the host program. In order to run a kernel on the GPU we first choose a device (a GPU for us) via a `cl_device_id` object, then create a *program* via a `cl_program` object. Finally we create a context via `cl_context`. Put simply, a context allows devices to receive kernels and exchange data. Then we create a command queue via a `cl_command_queue` object. The device receives kernels through a command queue, and it does exactly what it sounds like: it places kernels in a queue that are then executed in the order in which they were enqueued.



```
__kernel void
dp_mul(global const float *a,
      global const float *b,
      global float *c)
{
  int id = get_global_id(0);
  c[id] = a[id] * b[id];
}
```

Figure 2.2: Basic Platform and Runtime APIs in OpenCL [15]

We then compile the kernel, and then create our *buffers* which transfer data between the *host* and the *device*, in this case our CPU and GPU respectively. The buffers must be created via `clCreateBuffer()`, and then data must be either copied or mapped into these buffers before we can use them to pass data to/from the device. The difference between copying and mapping will be covered in detail later in the paper.

Finally we launch the kernel. To do this, we add necessary arguments (usually buffers and integers) that will be passed to the device. We then set the local and global work sizes, which were previously introduced, however setting these is an implementation-specific process and is therefore discussed in-depth in chapter 6. Finally, the kernel is enqueued via the `clEnqueueNDRangeKernel()` function.

For a basic overview of OpenCL at present, it is enough to understand kernels, the basics of the host program, and the command queue. Other more subtle concepts and ideas will be introduced later as needed, however these overarching concepts will be prominent throughout the work.

Chapter 3

The Gaussian Process Regression

The aim of this project is to find bottlenecks in the Gaussian Process regression. However in order to be able to find bottlenecks we must first understand what it is that the Gaussian Process regression does and how it does it.

From a high level, the Gaussian Process regression predicts "output" data points, based on "input" data points. It approximates the underlying distribution of the input, or test, data points by using a set of Gaussian distributions, and provides the predicted output data points based on this distribution.

The theory behind the basic GP regression is fairly complicated, however it has been covered quite extensively by Rasmussen & Williams [11]. They also provide some discussion of the computationally expensive steps of the Gaussian process regression which will be discussed in the this chapter.

An explanation of the underlying theory of the GP regression, while time consuming, is necessary in order to set expectations for and then subsequently understand profiling results in the next chapter. It will also set the framework for understanding the entirety of this paper from the perspective of the original goal - optimising the GP regression. As we delve further in-depth into optimising certain sections, the reader can always come back to this chapter to contextualise what they're reading in the larger scheme of the GP regression implementation.

3.1 A Basic Layout of the Algorithm

The basic Gaussian process regression, as discussed in Rasmussen & Williams [11], page 19, Algorithm 2.1, is discussed here.

3.1.1 Building the Covariance Matrix

The first part to the GP regression is taking input data points and transforming those data points into a covariance matrix by use of a *covariance function*, which specifies the covariance between pairs of random variables. An arbitrary function of input pairs x and x' will not, in general, be a valid covariance function; to be a valid covariance function it must be positive semidefinite. In this implementation the squared exponential is used as our covariance function. With this function, the covariance is almost unity between variables which have inputs that are very close, while the covariance decreases as the distance between the inputs increases [11].

$$\mathrm{cov}(f(\boldsymbol{x}_p), f(\boldsymbol{x}_q)) = k(\boldsymbol{x}_p, \boldsymbol{x}_q) = \exp(-\frac{1}{2}|\boldsymbol{x}_p - \boldsymbol{x}_q|^2) \tag{3.1}$$

Where x_p's represents the p-th training input point and x_q represents the current training input point.

Given n input data points an $n \times n$ covariance matrix, K, is created using 3.1 to calculate each element.

$$\begin{bmatrix} k(x_0,x_0) & k(x_0,x_1) & k(x_0,x_2) & \cdots & k(x_0,x_n) \\ k(x_1,x_0) & k(x_1,x_1) & k(x_1,x_2) & \cdots & k(x_1,x_n) \\ k(x_2,x_0) & k(x_2,x_1) & k(x_2,x_2) & \cdots & k(x_2,x_n) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ k(x_n,x_0) & k(x_n,x_1) & k(x_n,x_2) & \cdots & k(x_n,x_n) \end{bmatrix} \tag{3.2}$$

3.1.2 Computing the Posterior Mean

After this covariance matrix has been created for all input data points, the Gaussian process posterior mean is then computed as

$$\bar{\boldsymbol{f}}_* = \boldsymbol{k_*}^\intercal (K + {\sigma_n}^2 I)^{-1} \boldsymbol{y} \tag{3.3}$$

Where we write $\boldsymbol{k_*} = \boldsymbol{k(x_*)}$ to denote the vector of covariances between the test point and the n training points. $\boldsymbol{y}$ is the vector of targets. $\boldsymbol{\sigma_n^2}$ is the Gaussian noise variance, and I is the identity matrix also of size $n \times n$.

In the implementation discussed, we split the computation of the posterior mean into two operations. We use an intermediary value, $\boldsymbol{\alpha}$, as follows to calculate the posterior mean

$$\begin{aligned} \boldsymbol{\alpha} &= (K + {\sigma_n}^2 I)^{-1} \boldsymbol{y} \\ \bar{\boldsymbol{f}}_* &= \boldsymbol{k_*}^\intercal \alpha \end{aligned} \tag{3.4}$$

Looking at eq. 3.4 we can observe that this calculation involves the inversion of the covariance matrix plus added Gaussian noise, $K + {\sigma_n}^2$. As the complexity of a matrix inversion is typically $O(n^3)$, this term dominates the GP regression and causes the basic complexity of the entire algorithm to be $O(n^3)$, therefore any attempt at speeding up this algorithm will very likely be focused on improving the speed of this matrix inversion [11].

3.2 A Faster Matrix Inversion Using Cholesky Factorisation

There are a number of methods available for inverting matrices. These include Gauss-Jordan elimination, LU Decomposition, and Cholesky factorisation among others. The computational complexity of commonly used algorithms is $O(n^3)$, yet *Cholesky factorisation* typically involves about $n^3/3$ FLOPs (Floating Point Operations). This means that as a method it is about $2\times$ faster than LU decomposition, which uses $2n^3/3$ FLOPs [16]. Its speed and numerican stability make Cholesky factorisation an ideal method for matrix inversion [11].

The reason that the use of Cholesky factorisation is available in this case is that a covariance matrix has the property of being both *symmetric* and *positive definite*. The result of inverting a symmetric positive definite matrix is also a symmetric positive definite matrix [17]. This means that all the matrix mirrors its self about the main diagonal. Due to this property, by using the Cholesky transform, we can avoid operating on any elements above the main diagonal and thus avoid unnecessary calculations.

3.2.1 Adjusting The Gaussian Process Regression For Use Of Cholesky Factorisation

In order to make this switch to Cholesky factorisation, the basic steps to the Gaussian process regression must be altered as follows

$$\begin{aligned} L &= cholesky(K + {\sigma_n}^2 I) \\ \boldsymbol{\alpha} &= L^{\mathsf{T}} \backslash (L \backslash \boldsymbol{y}) \\ \bar{f}_* &= {\boldsymbol{k}_*}^{\mathsf{T}} \boldsymbol{\alpha} \end{aligned} \tag{3.5}$$

Where $\boldsymbol{L}$ is the Cholesky factor and the \ is the left matrix divide operator, where $X = A \backslash B$ is the solution to the equation $A * X = B$.

As we expect Cholesky factorisation to be the most time-consuming step in our GP regression, we will explain it in this section. Then we will explain the exact Cholesky factorisation used in our implementation and why it is implemented in this particular way.

3.2.2 Cholesky Factorisation Explained

The Cholesky factorisation is broken down in this section in order for the reader to be able to understand the next section where the Cholesky factorisation implementation for C++ is explained.

Beginning with the covariance matrix

$$K = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,n} \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & a_{n,3} & \cdots & a_{n,n} \end{bmatrix}$$

Let $k : [1, n]$ iterate through all rows. Also let $j : [1, i-1]$ iterate through all columns within the lower-triangular matrix, using $i : [1, n]$ as the limit for column iteration in order to stay within the bounds of the triangular matrix.

Each time k increments, i also increments, as row 1 has 1 column and row 2 has 2 columns, etc.

$$L = cholesky(K) = \begin{array}{cc} & \overrightarrow{\text{j=1} \qquad\qquad\qquad \text{j=i-1}} \\ \begin{array}{c} \text{k=1} \\ \\ \\ \\ \text{k=n} \end{array} \Bigg\downarrow & \begin{bmatrix} l_{1,1} & & & & \\ l_{2,1} & l_{2,2} & & & \\ l_{3,1} & l_{3,2} & l_{3,3} & & \\ \vdots & \vdots & \vdots & \ddots & \\ l_{n,1} & l_{n,2} & l_{n,3} & \cdots & l_{n,n} \end{bmatrix} \end{array}$$

Elements on the main diagonal, e.g. elements where $k = i$, are operated upon by the equation

$$l_{kk} = \sqrt{a_{kk} - \sum_{j=1}^{k-1} {l_{kj}}^2} \tag{3.6}$$

On all other elements, where $k \neq i$, the operation upon the matrix is described by the equation

$$l_{ki} = \frac{a_{ki} - \sum_{j=1}^{i-1} l_{ij} l_{kj}}{l_{ii}} \tag{3.7}$$

The order of calculation is from top to bottom and left to right. So $l_{1,1}$ is calculated first, then $l_{2,1}$, then $l_{2,2}$, then $l_{3,1}$ and so on until $l_{n,n}$ is calculated and the Cholesky factorisation is complete.

3.3 Implementation of Cholesky Factorisation In Practice

As can be seen from eq 3.6 and eq 3.7, Cholesky factorisation is inherently linear. In other words, as we are iterating through the rows of the covariance matrix we need the previous row's results in order to calculate the current row of L. Therefore we can split the Cholesky factorisation up for each row, and instead of building the covariance matrix and then performing a Cholesky factorisation on the covariance matrix at once, we simply build the Cholesky factor directly, row by row, similar to how we computed the covariance matrix.

Essentially, the Cholesky factorisation is broken down into a few steps. The first step is to build the sample vector, $\boldsymbol{k}$. The next step is to solve the lower triangular linear equation $L * \boldsymbol{k} = \boldsymbol{k}$ where L is the Cholesky factor built thus far for the n-th input. And finally the vector $\boldsymbol{k}$ is added back as the next row in the Cholesky factor L and modify the n-th element of the new row by $l_{n,n} = \sqrt{cov(x_n, x_n) - \boldsymbol{k}.\boldsymbol{k}}$. This is still a Cholesky factorisation, but broken down in multiple steps. This will make analysing it more simple in the future.

```cpp
// For each of the n sample data points
for(int n = 0; n < numberOfSamples; n++){
    for (int i = 0; i<n; ++i) {
    /* Create a vector, k, of the covariance between i-th input
     * and n-th input
     */
        k(i) = cf->get(sampleset->x(i), sampleset->x(n));
    }

    /* Solve linear equation: L*k = k where the lower triangular
     * matrix L is the Cholesky factor for the n-th input
     */
    L.topLeftCorner(n, n).triangularView<Eigen::Lower>()
                              .solveInPlace(k);

    // Setting the n-th row of L equal to the transposition of k
    L.block(n,0,1,n) = k.transpose();

    // coefficient (n,n) of L is set equal to sqrt(kappa - k.dot(k))
    double kappa = cf->get(sampleset->x(n), sampleset->x(n));
    L(n,n) = sqrt(kappa - k.dot(k));
}
```

Listing 3.1: Simplified example of the key steps to building the Cholesky factor, L, in C++ [10]

In this program, the BLAS library Eigen3 [13] is used in order to perform all of these operations. L and $\boldsymbol{k}$ are `Eigen::Matrix` and `Eigen::Vector` types respectively.

Chapter 4

Profiling the Code

Now that we suspect that the Cholesky decomposition step may constitute a significant portion of total program time, it is necessary to ascertain if this is indeed the case and also if there are any other sections that take up a large percentage of total execution time in the program. We would also like to know how much time, as a percentage, is being spent in these bottlenecks. This allows us to ensure that optimisation is worth the effort.

If, for example, there was a section that took around 20% of program time, then it may not be worth optimising this section as any improvement in this 20% would have a very small impact on the overall time it takes to run the program. In this case, optimisation would not be worthwhile. We are targeting sections that represent a large percentage of overall execution time as ideal candidates for parallelisation. Therefore we use a profiling tool, in this case Gprof, to get an overall idea of where time is being spent in the program.

4.1 Gprof Profiling

The idea with profiling at this stage is just to get a general idea of where time is being spent, and then gradually hone in on a particular line, loop, or function that takes up a significant percentage of overall program execution time. For this job, the Gprof performance analysis tool is ideal. It was used on a basic GP regression program with n = 4000 pseudo-randomly generated training points, and m = 1000 prediction points. Setting up Gprof is simple, we simply pass the `-pg` flag to the g++ compiler to enable Gprof profiling.

4.1.1 Interpreting The Flat Profile

Gprof provides two types of results: a *flat profile* and a *call graph*. The flat profile shows the total amount of time a program has spent executing the most expensive function. The important statistic generated by the flat profile is "%

time" - this is the percentage of the total execution time the program spent in various functions.[18]. The function that had the highest percentage by far was at 88.2%, and was a library function from Eigen3, the linear algebra library used by this code.

% time	cumul. (sec)	self (sec)	call count	function name
88.20	2.24	2.24	998000	`Eigen::internal::general_matrix_vector_product`
3.54	2.33	0.09	4000	`Eigen::internal::triangular_solve_vector`
2.36	2.39	0.06	1799508	`libgp::GaussianProcess::~GaussianProcess()`
1.97	2.44	0.05	12002000	`libgp::CovSEiso::get()`
1.97	2.49	0.05		`libgp::GaussianProcess::add_pattern()`
0.79	2.51	0.02	12002000	`libgp::CovNoise::get()`
0.79	2.53	0.02		`libgp::CovMatern5iso::get()`
0.39	2.54	0.01	20004000	`libgp::SampleSet::x()`

Table 4.1: Gprof flat profile

This finding is not conclusive proof of overall program time. Although it hints at the function `Eigen::internal::general_matrix_vector_product` taking up $> 80\%$ of all program time, we just can't be certain just from the flat profile. Profilers are not always accurate, and in this case we are not yet sure where this `Eigen::internal::general_matrix_vector_product` function is coming from and in what context it is executing. Presumably this Eigen library function is being used to perform a matrix-vector product, but we need more information. Luckily Gprof produces another type of output - the call graph - that will help us contextualise this information within the program and better understand where this function is being called in the program.

4.1.2 Interpreting The Call Graph

To find out more about what is using this `general_matrix_vector_product`, the Gprof *call graph* is consulted. The call graph shows how much time was spent in each function *and its children* [18]. It gives more information about the call order of various functions, and so we can begin to investigate the function exposed in the flat profile (from the previous section) that takes up so much time.

The call graph is included in Appendix A for reference.

By checking index 1, we can see that an enormous 98% of time is spent within `libgp::GaussianProcess::add_pattern()` and its children. On line 4 it is shown that that this `add_pattern()` method calls the library function `Eigen::triangular_solver_selector`. By checking the call graph it's clear that a series of sub-calls (lines 30 and 35 respectively, also lines 46 and 51 respectively) are performed from `Eigen::triangular_solver_selector`. By inspection into the code of the corresponding header file in the Eigen library and the graph at index 2, we can build up the following call graph that describes the series of calls that take up so much time.

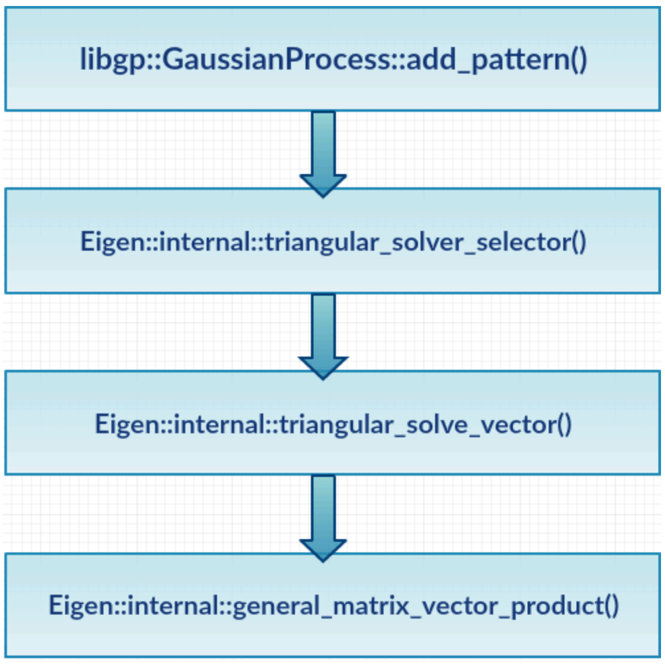

libgp::GaussianProcess::add_pattern()
Eigen::internal::triangular_solver_selector()
Eigen::internal::triangular_solve_vector()
Eigen::internal::general_matrix_vector_product()

Figure 4.1: Visual call graph generated from profiling results

This diagram allows us to link the information we gathered from the flat profile - that 88.2% of time is being spent in `Eigen::internal::general_matrix_vector_product` - with the information gathered from the call graph - that 98% of time spent in `libgp::GaussianProcess::add_pattern()` and its children functions. Further probing into the `libgp::GaussianProcess::add_pattern()` code, and remembering that some sort of triangular solver is being called using the Eigen3 template library [13], leads us to time various sections in the `add_pattern()` function that use Eigen library functionality.

By doing this, we find that the section of code that takes by far the longest to execute is the line:

```
L.topLeftCorner(n, n).triangularView<Eigen::Lower>().solveInPlace(k);
```

This expensive calculation that uses Eigen library functions [13], is a lower-triangular linear equation solver. It solves the system of linear equations $L \times \boldsymbol{k} = \boldsymbol{k}$ where L is the Cholesky factor and $\boldsymbol{k}$ is the covariance vector, both explained in depth in the previous chapter.

We already had the expectation that inversion of a matrix would be one of the most, if not the most, computationally expensive steps in the GP regression. This result, however, provides confirmation. We are already using Cholesky decomposition to attempt to quicken the matrix inversion, but there is more that can be done.

By using OpenCL, we can attempt to parallelise the linear equation solver step within the Cholesky factorisation, and thereby speed up the entire program. Because we are iteratively building the Cholesky factor, we must perform this triangular solve for L as an $n \times n$ triangular matrix, and $\boldsymbol{k}$ as a vector of length n where n takes integer values from 0 to the total number of input data points.

This means that even when performing the GP regression for a large number of input data points, there will always be small dimensional systems that need to be solved. Generally, parallel code running on GPUs performs better, relative to its linear counterpart, when the number of elements or "work items" is large. Therefore it would be ideal to avoid performing these small linear equation solves on the GPU, and therefore avoid incurring the overhead that goes along with setting up an OpenCL program, the associated data transfer, and other such factors.

The solution to this is to use a CPU serial approach using BLAS (such as the Eigen library-dependent code mentioned on page 21) for small systems, and for larger systems use a parallel GPU version. This way, unnecessary OpenCL overhead can be avoided for small matrices that will not benefit from parallel computation.

Chapter 5

The Triangular Solver

Now that it's clear that the lower-triangular solver, and by extension the Cholesky transform, is by far the most computationally heavy operation in our program, we need to be sure that we can parallelise it. This chapter discusses the methods we chose to use and why, and then provide a brief mathematical background of the triangular solver. It ends with a basic serial implementation in C, which will then be parallelised in the next chapter.

5.1 From Cholesky factorisation To The Triangular Solver

Recall that in 3.3 we discussed the Cholesky factorisation, and provided an implementation that involves a triangular solver with some modifications. We discovered that this triangular solve step takes up a large majority of program time.

The decision was made to parallelise the triangular solve rather than the entire Cholesky decomposition which has been covered in various papers and implementations [19, 20]. Specifically the paper by Brunelli et all [20] describes a parallel Cholesky factorisation that makes use of a desktop GPU. We effectively chose to extract the second step of this method, the ”scaling” step, and develop a linear solver based on parallelising this step.

By breaking the Cholesky factorisation down like this and parallelising only this step on the GPU, we are allowed the option to perform the triangular solve on the CPU using BLAS for small matrix sizes, while performing large matrix operations on the GPU. This allows us to avoid the GPU overhead for smaller matrices that would not be large enough to be improved via parallelisation. Therefore our implementation will begin iteratively building the Cholesky factor, L, via BLAS as shown in 3.3. Once we reach a matrix size where the parallel implementation has begin to outperform the BLAS implementation, we switch over to our OpenCL version.

In order to build the serial implementation, we must first understand the mathematics behind what the program is doing. This will make it possible to understand the OpenCL implementation and then the optimisations that follow that, which require a basic mathematical understanding of the algorithm to follow.

5.2 Introducing The Triangular Solver

Developing efficient parallel triangular solvers for GPU is a challenging field, and one that is still young [21, 22, 23, 24, 25]. Much of the latest and most complex work in this field is focused on sparse triangular systems, however developing an effective solver for our dense system is something that has been done effectively for GPU [25, 23]. An added bonus to this work is that parallel triangular solvers are not only useful in Cholesky decomposition, but also fundamental to incomplete LU factorization family preconditioners as well as algebraic multigrid solvers and a whole host of other applications [23].

There are a number of methods for solving these triangular systems. One possibility was to implement a parallel tri-diagonal matrix solver [26], as the lower-triangular matrix is a subcategory of tri-diagonal matrices. However by building a solver specifically for the lower-triangular matrix and not for general tri-diagonal matrices, we can very much simplify the algorithm needed to work with.

The method for solving linear equations in the form of a lower triangular matrix is straightforward. We take advantage of the property of a triangular matrix, by performing back-substitution and parallelising each iteration of the back-substitution. Similar methods are used across a range of other reference papers when solving for dense triangular systems [23, 25].

In order to build a parallel implementation, however, it is wise to first build a serial implementation in C++ that will we will then be able to port to OpenCL. The goal of this chapter is therefore to outline the basic method for solving a lower-triangular matrix that will then be coded into a serial implementation [25]. The chapter after this will then go on to transform this serial code into a parallel implementation using OpenCL.

The emphasis from here on out will be to outline a basic triangular solver without all the bells and whistles, and then see how it performs on our GPU. Mathematically tweaking the triangular solve its self is outwith the scope of this paper as this is complex mathematical work, whereas this paper's focus is on performance of GPUs when applying a solver to them. While we do not want to spend too much time on the mathematics behind the solver, we need to first understand the algorithm behind a good triangular solve method that has been shown to be robust to parallelisation.

5.3 Solving a Lower-Triangular Matrix

This method has two steps. First, the matrix is normalised e.g. the matrix is transformed by dividing each row such that all values in the main diagonal are equal to 1. The details will be explained in the next section.

The second step is the critical step, and will be very central to our efforts towards parallelisation using OpenCL in the next few chapters. In this step we perform back-substitution for all rows of the matrix. This essentially means that we plug in the first row of the matrix into the second row. Then we plug the second row into the third, and so forth until we have solved the matrix. The details for this will also be outlined later.

From here onward, the left-hand $n \times n$ matrix will be referred to as A, and the right-hand vector of length n will be refered to as $\boldsymbol{k}$.

Step 1: Matrix Normalisation

In this step, the matrix, A, is normalised. This means dividing each row such that all values in the main diagonal become equal to 1.

We begin with a lower triangular matrix of form

$$\begin{bmatrix} a_{1,1} & 0 & 0 & \cdots & 0 \\ a_{2,1} & a_{2,2} & 0 & \cdots & 0 \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & a_{n,3} & \cdots & 1 \end{bmatrix} \quad \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$

The first step is to normalise the matrix by dividing every row, R, by the diagonal element in that row, $a_{i,i}$. These divisions include the vector, $\boldsymbol{k}$.

$$R_i \leftarrow R_i / a_{i,i} \tag{5.1}$$

In matrix form, we have

$$\begin{bmatrix} a_{1,1}/a_{1,1} & 0/a_{1,1} & 0/a_{1,1} & \cdots & 0/a_{1,1} \\ a_{2,1}/a_{2,2} & a_{2,2}/a_{2,2} & 0/a_{2,2} & \cdots & 0/a_{2,2} \\ a_{3,1}/a_{3,3} & a_{3,2}/a_{3,3} & a_{3,3}/a_{3,3} & \cdots & 0/a_{3,3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1}/a_{n,n} & a_{n,2}/a_{n,n} & a_{n,3}/a_{n,n} & \cdots & a_{n,n}/a_{n,n} \end{bmatrix} \quad \begin{bmatrix} x_1/a_{1,1} \\ x_2/a_{2,2} \\ x_3/a_{3,3} \\ \vdots \\ x_n/a_{n,n} \end{bmatrix}$$

Thus yielding a normalised lower triangular matrix of the form

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ a_{2,1} & 1 & 0 & \cdots & 0 \\ a_{3,1} & a_{2,2} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & a_{n,3} & \cdots & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$

Step 2: Back-Subsitution

The next step is to back-substitute. Starting from the top row, R_0, of the left-hand matrix A as well as the vector $\boldsymbol{k}$, each row is progressively substituted into the next row. We do this until we finish all rows, and thus all variables are eliminated and we have our solution in the form of a vector.

$$R_i \leftarrow R_i - a_{ij} \times R_{i-1} \tag{5.2}$$

Starting from the top of the matrix the first back-substitution is

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ a_{2,1} - a_{2,1} * 1 & 1 - a_{2,1} * 0 & 0 - a_{2,1} * 0 & \cdots & 0 - a_{2,1} * 0 \\ a_{3,1} - a_{3,1} * 1 & a_{3,2} - a_{3,1} * 0 & 1 - a_{3,1} * 0 & \cdots & 0 - a_{3,1} * 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} - a_{n,1} * 1 & a_{n,2} - a_{n,1} * 0 & a_{n,3} - a_{n,1} * 0 & \cdots & 1 - a_{n,1} * 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 - a_{2,1} * x_1 \\ x_3 - a_{3,1} * x_1 \\ \vdots \\ x_n - a_{n,1} * x_1 \end{bmatrix}$$

This results in a matrix of form

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & a_{3,2} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n,2} & a_{n,3} & \cdots & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$

Continuing with back-substitution for the next row

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 - a_{3,2} * 0 & a_{3,2} - a_{3,2} * 1 & 1 - a_{3,2} * 0 & \cdots & 0 - a_{3,2} * 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 - a_{n,2} * 0 & a_{n,2} - a_{n,2} * 1 & a_{n,3} - a_{n,2} * 0 & \cdots & 1 - a_{n,2} * 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 - a_{3,2} * x_2 \\ \vdots \\ x_n - a_{n,2} * x_2 \end{bmatrix}$$

Produces the matrix

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & a_{n,3} & \cdots & 1 \end{bmatrix} \quad \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$

Continuing until all rows in the matrix have been eliminated yields a final matrix where the right hand vector, k, holds the solutions to the system of linear equations.

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix} \quad \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$

5.4 Implementing The Solver

These translate to C/C++ fairly simply. Step 1 from before becomes

```

// Normalisation
for(int i = 0; i < n*n; i++){
    if(i < n){
    // diags is the array of diagonal elements of A
        k[i] = k[i]/diags[i];
    }

    int row = i/n;
    int col = i%n;
    A[row*n+col] = A[row*n+col]/diags[row];
}
```

One thing to note in the above code is the use of `diags`, an array of all diagonal elements of A. We store the diagonal elements of A before normalisation as a preparation for parallelisation. One of the properties of kernel execution in OpenCL is that we cannot guarantee any ordering of the exectution of the kernels. This means that we could change the diagonal element of A, and then attempt to read back that element assuming it was its previous value. This would cause the results of normalisation to be incorrect. Therefore we store the diagonal elements ahead of time.

The back-substitution step is then coded as

```
// Back-Substitution
for(int currentCol = 0; currentCol < n; currentCol++){
    for(int i = currentCol+1; i < n; i++){
        k[i] = k[i]-A[i*n+currentCol]*k[currentCol];
    }
}
```

Once these have been written and the entire program tested against a BLAS version of a triangular linear solver to ensure correctness, we can move on to the next chapter where we build the parallel OpenCL version.

Chapter 6

Triangular Solve For GPU

As the last chapter provided an understanding of how the triangular solver works and what it looks like when implemented, this chapter shows how the serial implementation is transformed to a parallel one in OpenCL. The process for transforming C code to OpenCL has already been discussed in the Background chapter (page 13), therefore this chapter will focus on the details of the OpenCL implementation specific to the Mali T-604 GPU. These include what the kernels look like and how they are called from the host, work sizes and various considerations when choosing those, and a first look at results of the parallel triangular solver.

6.1 Basic OpenCL Implementation

Although the back substitution is an inherently sequential algorithm for dense triangular systems [24], we can still exploit parallelism in the normalisation of the matrix and also in each iteration of the back substitution outlined in the previous chapter.

The basic OpenCL implementation is similar to that of the serial implementation. As solving the triangular matrix involves two steps, there are two kernels: the first kernel normalises the square matrix to produce a unit-diagonal matrix, and the second kernel performs the back substitution.

This section will continue with the code for each kernel, written in OpenCL, accompanied by a brief explanation, as well as an outline of how these kernels are called from the host. Afterwards we will discuss the design decisions accompanying these kernels and finish with timing results.

6.1.1 Kernel 1

```c
__kernel void kernel1(__global float* A, __global float* k,
                      __global float* diags, int wA )
{

    int index = get_global_id(0);
    int number_of_elements = wA+((wA+1)*(wA))/2

    if(index < wA){
        k[index] = k[index]/diags[index];
    }
    else if( index >= wA && index < number_of_elements){
        index = index-wA;
        int row = floor(-0.5 + sqrt(0.25 + 2 * index));
        int col = index - (row * (row + 1) / 2);
        A[row*wA+col] = A[row*wA+col]/diags[row];
    }
}
```

Listing 6.1: Kernel 1

The number of elements including and below the main diagonal of a lower-triangular matrix is given as

$$\sum_{n=1}^{N} n = \frac{(1+N)\,N}{2}$$

By operating on $\frac{(1+N)\,N}{2}$ elements instead of N^2 elements when normalising the matrix in kernel 1, we cut down on the number of operations to be performed.

6.1.2 Kernel 2

```c
__kernel void kernel2(__global float* A, __global float* k,
                      int hA, int currentCol)
{
   int row = get_global_id(0)+currentCol+1;
         if(row<hA ){
            k[row] = k[row]-A[row*hA+currentCol]*k[currentCol];
         }
}
```

Listing 6.2: Kernel 2

In kernel 2 we still operate as if we are dealing with a full square $n \times n$ matrix, but we cut down the number of operations by deviating from the method outlined in the previous chapter, and operating only on the right hand vector, k, while leaving the matrix, A, untouched.

This allows us to map the matrix, A only once for all iterations of kernel 2 and save on reads and writes of the matrix (mapping will be discussed further in the next chapter). It also means that we only need to write back the vector k, rather than write back both k and the matrix A.

6.1.3 Calling The Kernels From The Host Program

The way that these two kernels are called from the host is as follows:

```
/* All OpenCL Setup */

/* Kernel 1 */

for(currentCol=0; currentCol < n; currentCol++)
{
    /* Kernel 2 */
}
```

Listing 6.3: Host program calling the kernels

As can be seen, kernel 2 is called n times, with n being the size of the matrix. This is because there is no way to globally synchronise across work-groups in OpenCL as of yet. And because we must wait for each column to finish computing it's results, as was pointed out in the last chapter, this global synchronisation is necessary. Therefore we have no choice but to loop over Kernel 2 n times, where each iteration performs operations on a given column of the matrix [27]. This is the only way to ensure that all operations on the previous column have been finished before operations on the next column begin.

There are a number of important design decisions to be made when calling or *enqueueing* the kernels. These include considering registers and work-item sizes that maximise performance of the kernels. These various decisions will be layed out and justified based on the specifications of our hardware in the next few sections.

6.2 Registers and Maximum Number of Work-Items

In order to achieve the maximum number of work-items available on a single core, $\max(I) = 256$, it is necessary to limit the number of registers being used [28]. The maximum number of active work-items I is determined by the number of registers R used by a single kernel

$$I = \begin{cases} 256, & 0 < R \leq 4, \\ 128, & 4 < R \leq 8, \\ 64, & 8 < R \leq 16. \end{cases}$$

Each kernel must therefore be limited to using ≤ 4 registers. The local work size of a given kernel can be queried with the function `clGetKernelWorkGroupInfo()`, and at every change in the kernel these sizes must be checked to ensure that the maximum size is maintained. For the current implementation, the local work group sizes of kernel 1 and kernel 2 are both shown to be at $I = 256$. If over the course of optimisation these figures drop below this value, the code must be reassessed as keeping these figures at the maximum is crucial to the GPU working as efficiently as possible.

This brings up the issue of whether to use single or double precision floating-point precision. In order to use double precision in OpenCL 1.1, support must be enabled via an optional extension. Once enabling support for double floating-point precision, however, the `clGetKernelWorkGroupInfo()` function shows that kernels are using only $I = 128$ work-items. This is because double precision values take up a much higher number of registers, therefore using over 4 registers and cutting the number of work-items available on a single core to half the maximum. This would be an unacceptable performance hit, therefore the choice must be made to settle for single floating-point precision.

6.3 Global Work Size

The global work size of a program is equal to the number of work-items needed to complete an iteration of your program. Each work-item executes the kernel once, so for example if some program executes a kernel once for each element of an array of size n, then n work-items are needed.

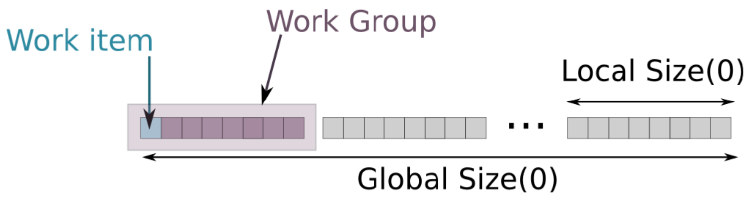


Figure 6.1: 1D Data-Parallel execution in OpenCL [14]

OpenCL dictates that the global work size must be a multiple of the local work size (also known as the work group size). The local size is variable, however we maximise the computational power of the GPU by using the highest available local work size, $I = 256$ [29, 30, 31].

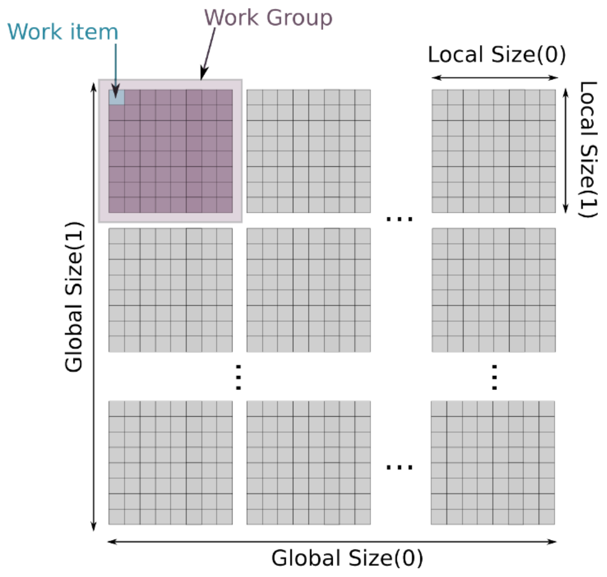


Figure 6.2: 2D Data-Parallel execution in OpenCL [14]

The ideal global work size, however, is quite straightforward to compute. OpenCL stipulates that the global work size must be a multiple of the local work size times the number of shader cores times a constant. In the Mali T-604, there are 4 shader cores [29].

Therefore global work size is given by

$\texttt{global work size} = \texttt{local work size} \times \texttt{num of shader cores}$

6.3.1 Rounding Up The Global Size

In this implementation, both kernel 1 and 2 need $\frac{(1+N)N}{2}$ work-items. Ideally, we could simply plug this global work size into the above formula and move on, however this is not enough. The problem is that the global work size is not always perfectly divisible by the *local work size* $\times$ *number of shader cores*.

The standard solution is to round up the global work size to the nearest number that is divisible by the local work size, and assign empty work-items to these remaining "empty" work-items in order to allow rounding without causing errors. This means that the GPU will use more work-items than would be ideal, however it is unavoidable since without rounding, we cannot satisfy the equation above. Programatically, this has been implemented as a bound check inside the kernel to skip any computation for these "empty work-items", as can be seen in the second condition on line 13 in Kernel 1.

6.3.2 Global Sizes Should Be Large

The final consideration for global work sizes is that the optimal global work size also needs to be high - several thousand work-items - in order to ensure good performance [29]. This means that our OpenCL program should theoretically perform better relative to its serial counterpart when performed on larger matrices. If we operated on small matrices using OpenCL, the the size of the global work sizes when operating on small matrices would be too small to work well and the overhead would make solving using OpenCL useless. Therefore, if the point is reached at which OpenCL outperforms BLAS, then experiments will be done to choose a value of n in the GP regression algorithm at which the program will switch from using a serial BLAS program for solving triangular systems to a parallel OpenCL one. This ensures that overhead is minimal for small matrices, while still allowing for parallelism to be exploited effectively for large matrices.

6.4 Setting Up Experiments

This section will give some explanation on the way experiments will be run in this paper. The methods explained here will be maintained until the end of the paper every time experiments are performed. For the time being, timing is performed via a monotonic clock from the host program. A special OpenCL function will be introduced later in this paper that profiles various intervals of execution time, however this function will be discussed in more detail when its use becomes necessary.

It is important to ensure that timing results are accurate and consistent when measured, however there will inevitably be small variations in the timing data collected. In order to mitigate the effect of small changes, each experiment will be run 10 times for each matrix size and the average will be given as the presented figure. These test matrices will be lower-triangular matrices passed to the program as $n \times n$ matrices with 0's in the position above the main diagonal. The matrix sizes for small matrices will be n= 32, 64, 96, 128, 512. All matrix sizes above this will be considered large matrices.

We opted to begin by plugging the triangular solver into the GP regression implementation directly. This way, in the event that we can improve the triangular solver to the point where it performs faster than the Eigen BLAS implementation for large matrices, we will immediately be able to time the complete GP regression implementation with the parallel triangular solver implemented in place of the Cholesky factorisation. This means that we are actually computing a GP regression as was discussed in chapter 3.3, however at certain values of n - 32, 64, 96, 128, 512, 1024, 4096, and later even larger values - we instead use our parallel triangular solver `triangParallel(k_out.data(),L_out.data(),n)` in place of the BLAS solver.

This function call is in fact a call to our OpenCL host program and the call to this program takes three arguments - the vector $\boldsymbol{k}$, the Cholesky factor as has been computed thus far L which will be referred to as A from here on out within the OpenCL host and kernel code for the duration of this pape, and finally the last argument n is size of the dimensions of the matrix L and length of the vector $\boldsymbol{k}$. The output of this function is the changed vector $\boldsymbol{k}$ which will then need to be inverted and added to the next row of the Cholesky factor L as described in chapter 3.3. However we leave these operations to be performed by Eigen library function calls as they are not particularly time-consuming according to our profiling results.

For the purposes of testing, the resulting vector $\boldsymbol{k}$ that will be the returned by the host program `triangParallel()` will then be compared against a vector output by the Eigen BLAS version of the triangular solver, in order to ensure accuracy of the result produced from our OpenCL implementation. This will be done every step of the way through our vectorisation process to ensure that we have implemented the program accurately.

Input data points, as provided by the base implementation by Manuel Blum, are pseudo-random number uniformly distributed, which are then passed as arguments to a hill function. Therefore, they are some point along the hill function. The result is then multiplied by an independent, standard normally distributed random number (generated using the Box-Muller transform). Input points generated in this way are then passed to the program, and the Gaussian process regression proceeds as discussed on these points.

Before we are able to run the code, however, we need to ensure that our program can access the OpenCL drivers for the Mali T-604 GPU as well as the OpenCL header files in order for everything to work properly. In order to accomplish this we must include some compiler flags to direct the compiler and linker to the right places.

6.4.1 Compiler and Linker Flags

We cannot simply begin writing OpenCL, we must first configure CMake such that we have access to the OpenCL drivers for the Mali GPU as well as the OpenCL header files. As this project is rather large and spans many files and directories, CMake has been used to compile everything and manage cross-file dependencies. This was a decision made and implemented by the original author of the GP regression implementation [10]. Therefore there is some CMake syntax included here that will not be explained in-depth. Instead we will focus mainly on the flags.

There are three types of flags that we are concerned when configuring the `CMakeLists.txt` file. These are the `CMAKE_C_FLAGS`, `CMAKE_CXX_FLAGS`, and `CMAKE_EXE_LINKER_FLAGS`. We have set these up as follows to allow us to run our OpenCL code.

```
SET(CMAKE_C_FLAGS "-I $CMAKE_SOURCE_DIR -I/usr/include")
SET(CMAKE_CXX_FLAGS "-I $CMAKE_SOURCE_DIR -I/usr/include" )
```

```
SET(CMAKE_EXE_LINKER_FLAGS "-I $CMAKE_SOURCEDIR -L/usr/local/lib/mali/
fbdev -Wl,-rpath,/usr/local/lib/mali/fbdev/")
```

First we will discuss the compiler flags `CMAKE_C_FLAGS` and `CMAKE_CXX_FLAGS`. The only option we added to these is `-I/usr/include`. This `-Idir` command adds the directory *dir* to the head of the list of directories to be searched for header files [32]. The OpenCL header files are contained in the directory `/usr/include/` on our system, therefore we add the directory containing these header files in our search so that they will be available through the `#include` command in our OpenCL host program.

For the linker flags, `CMAKE_EXE_LINKER_FLAGS`, the flags are somewhat more complicated. We need to link against the OpenCL library, `libOpenCL.so`, which is an ICD loader. In essence, this library acts as a dispatcher to real OpenCL implementations provided as OpenCL Installable Client Drivers (ICDs) [33]. In our case, these are the drivers for the Mali T-604 GPU. The two flags added for the linker are as follows.

The flag `-L`*dir* adds the directory *dir* to the list of directories to be searched for libraries [32]. We include the directory `/usr/local/lib/mali/fbdev` because this is where we find the ICD loader discussed above, `libOpenCL.so`. This enables us to access the directory specified in this path when we link it with the next flag, `-Wl`.

The `-Wl,`*option* allows us to pass *option* as an option to the linker. If *option* contains commas, it is split into multiple options at the commas [32]. Therefore we pass `-rpath,/usr/local/lib/mali/fbdev/` to the linker as an option. `-rpath` allows us to add the directory passed after it to the runtime library search path, and the `/usr/local/lib/mali/fbdev/` path is the same directory as the `-L` flag, which again contains the OpenCL drivers for the Mali GPU.

The final step is adding the two CMake commands

```
ADD_LIBRARY(OpenCL SHARED IMPORTED)
```

```
set_property(TARGET OpenCL PROPERTY IMPORTED_LOCATION /usr/local/lib/
mali/fbdev/libOpenCL.so
```

These commands, put simply, add the OpenCL shared library to be built from the source location. The location is described by the path `usr/local/lib/mali/fbdev/libOpenCL.so`.

Now we simply add the new OpenCL host program, `triangParallel()`, in `Sources.cmake` so that we are able to call it from `gp.cc` where the rest of the GP regression is being run. And finally, we can run the `make` command, and then run the executable that is produced.

The next section goes on to give timing results from running this program, using our first implementation of the triangular solver for the data points n=32, 64, 96, 128, 512, 1024, 4096.

6.5 Preliminary Results

The end goal is to optimise the GPU OpenCL solver until it's fast enough to performs better than the CPU BLAS (Eigen) version for large matrices. Therefore we must check the performance against the BLAS version in order to see how far towards this goal we are at this initial stage.

Therefore the BLAS execution time must be measured against the early parallel OpenCL version. Also, as a point of reference, the serial implementation built in the chapter before this has been included in order to compare against both BLAS and OpenCL versions.

	n=32	**n=64**	**n=96**	**n=128**	**n=512**	**n=1024**	**n=4096**
BLAS	12μs	31μs	52μs	78μs	820μs	5,042μs	195,787μs
Serial	36μs	143μs	609μs	745μs	8,722μs	34,957μs	650,252μs
Parallel	11,491μs	22,116μs	37,866μs	56,887μs	161,003μs	462,606μs	4,573,950μs

Table 6.1: Average execution time of each triangular solver

In order to begin improving the OpenCL program, it is important to isolate which sections use the most time. As matrix size increases, the overhead of creating the OpenCL program decreases as an overall percentage of program time, to a negligible amount. Also, kernel 1 shrinks as a percentage of overall computation time as the matrix dimensions increase.

	n=32	**n=64**	**n=96**	**n=128**	**n=512**	**n=1024**	**n=4096**
Kernel 1	0.098%	0.085%	0.074%	0.066%	0.044 %	0.037%	0.023%
Kernel 2 loop	13.9%	40.2%	50.2%	57.6%	89.2%	96.2%	99.4%

Table 6.2: Percentage of overall time spent in the two key sections of the OpenCL triangular solver

This is because data must only be mapped/unmapped once for kernel 1, and also because the kernel must only be enqueued once, and thus is quite robust to scaling. Kernel 2, however, is where time is spent most. Its percentage of total computation time increases with matrix dimension because as matrix size grows, the amount of data copied back and forth to the GPU increases. Kernel 2 must also be enqueued n times, therefore the higher n is the more overhead there is due to enqueueing.

Therefore in order to improve the entire parallel solver, emphasis must be placed on improving kernel 2. In order to improve kernel 2, data transfer to and from the GPU must be addressed in order to make it as efficient as possible. This will be the focus of the next chapter. That chapter after that, will delve into the sources of overhead due to enqueueing kernel 2 n times.

Chapter 7

Efficient Data Transfer

Although OpenCL allows portability of code across devices that support the language, tuning is always necessary to achieve optimal performance on a given GPU. Some techniques are considered standard for the optimisation of GPU code, and traditionally most of the programmer's effort is expended considering techniques for exploiting local and private memory.

However the memory optimisations that are considered standard for those larger conventional GPUs are not applicable on integrated GPUs such as the Mali devices. This is because the Mali GPU architecture, and many of these other integrated GPUs, support a unified memory system. This means that global and local OpenCL address spaces get mapped to the same physical memory, the system RAM, backed by caches transparent to the programmer[29]. Instead of using a standard *malloc* operation and copying buffers from global memory to the GPU local memory, memory buffers are meant to be mapped directly into the GPU memory space.

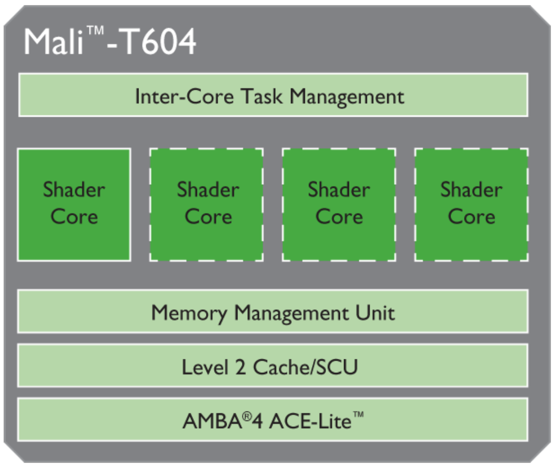
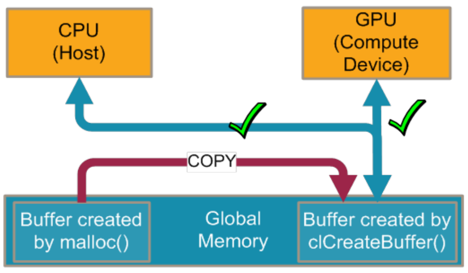


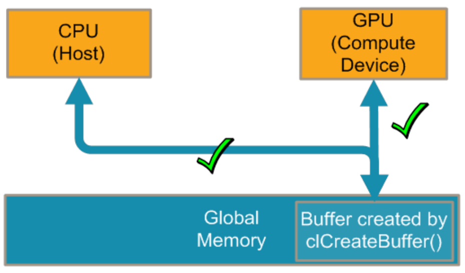
Figure 7.1: Mali T-604 Memory Model [34]

7.1 Passing Data to Kernels Efficiently

Because the Mali GPU supports a unified memory system, we can take advantage of shared memory. Normally an OpenCL buffer object is created using the `clCreateBuffer()` OpenCL function with the `CL_MEM_USE_HOST_PTR` flag (7.2a) and then data needs to be copied to and from the GPU via the `clEnqueueWriteBuffer()` and `clEnqueueReadBuffer()` OpenCL functions.


CPU (Host)
GPU (Compute Device)
COPY
Buffer created by malloc()
Global Memory
Buffer created by clCreateBuffer()

(a) CL_MEM_USE_HOST_PTR



(b) CL_MEM_ALLOC_HOST_PTR

Figure 7.2: clCreateBuffer() with both flags [30]

7.1.1 Usage of Mapping Functions

To avoid computationally expensive buffer copies, we can instead use memory mapping functions. Buffers have to be allocated using the `clCreateBuffer()` function, this time with the `CL_MEM_ALLOC_HOST_PTR` flag (7.2b). Then the functions `clEnqueueMapBuffer()` and `clEnqueueUnmapMemObject()` must be used to enable both the processor and the Mali GPU to access the data.

```
clEnqueueMapBuffer();
/* copy data into buffer */
clEnqueueUnmapMemObject();

clEnqueueMapBuffer();
/* Kernel 1 */
enqueueNDRangeKernel();
clEnqueueUnmapMemObject();

for(currentCol=0; currentCol < n; currentCol++)
{
    clEnqueueMapBuffer();
    /* Kernel 2 */
    enqueueNDRangeKernel();
    clEnqueueUnmapMemObject();
}
releaseBuffer();
```

7.1.2 Timing Comparison of Mapping vs Copying

Timing was done to compare the two methods for the loop in kernel 2 represented in the pseudocode above in lines 10-15. The results showed that mapping buffers is more robust to scaling than copying. If we're hoping for our implementation to perform better for larger matrices, we need data transfer to scale nicely with the size of our matrix, thus the smaller slope of mapping makes it favorable to copying.

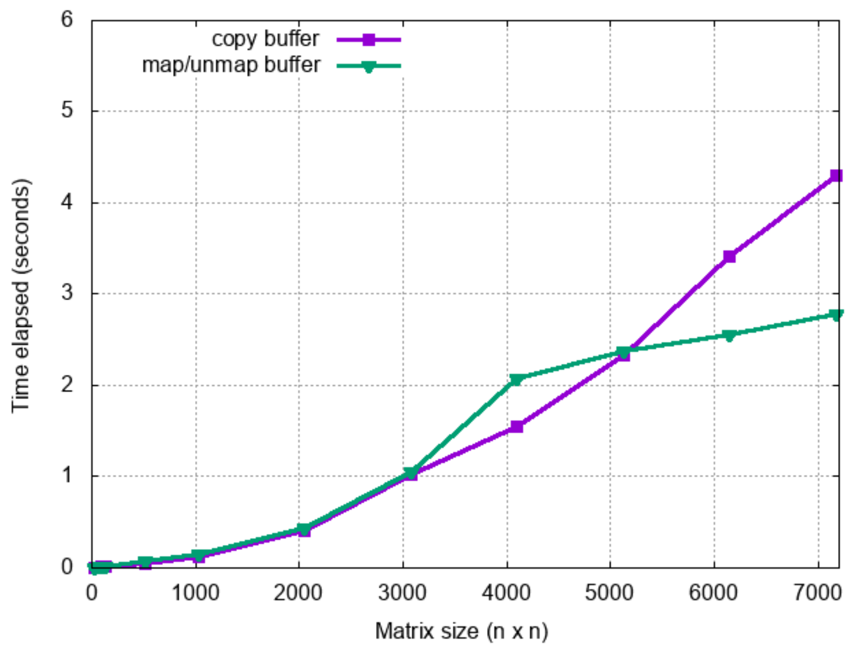


Figure 7.3: Average time taken to perform buffer operations for the back substitution step on the Arndale board

While mapping becomes the winner past matrix sizes of about n=5000 on this particular machine, it is important to note that the overhead for mapping and unmapping is still very high for large matrix values. Therefore it is important to understand just why we need to map and unmap in each iteration of the loop (lines 10-15 above), to justify the overhead that these operations create.

7.1.3 Mapping and Caches

In the Mali driver, memory is mapped when allocated and remains mapped for the lifetime of the buffer. Therefore in the loop which we have timed, the functions of `clEnqueMapBuffer()` and `clEnqueueUnmapMemObject()` are not primarily to map memory, but to perform cache maintenance - more specifically cache invalidation - on the host CPU side [35, 30].

Cache invalidation works as follows. If a processor has a local copy of data (the host memory), but an external agent (the GPU) updates main memory, then the local copy is considered outdated or *stale*. Before reading back this changed data, the processor needs to remove the stale data from caches by marking the cache lines invalid. In each iteration of the loop, this is the real function of the `clEnqueueUnmapMemObject()` command, it invalidates the cache. Therefore if we are mapping n entries, the CPU must loop through all n pages of the allocated memory to invalidate them. As our program maps a buffer of length n for an $n \times n$ matrix, n times, then it must grow in relation to the size of the matrix. We observe that this is linear, and the slope of this growth lessens as matrix sizes increase, however there will likely always be some increase of mapping time with matrix size.

In order to avoid this, one might rightly consider simply not unmapping memory at each iteration of the loop, however if we were to only map the buffer once, the GPU will have written fresh data but some cache lines in the CPU may be holding dirty data. In this scenario, it is possible that the dirty cache line can be read back onto the host, therefore producing a mistaken result. This is why unmapping, and subsequent remapping, must occur at every iteration of the loop.

7.1.4 Final Remarks On Mapping

There is little to no mention of caching to be found in discussions of mapping/unmapping operations on the Mali T-600 or similar T-x00 series integrated GPUs [31, 29, 28, 35, 30]. Most papers and documentation simply recommend mapping in order to avoid unnecessary copying of data, however discussion of the overhead involved in map/unmap operations due to cache invalidation is lacking. Although overhead due to mapping is much less than that of copying, it is still high.

In the next chapter, a more in-depth analysis overhead of enqueuing kernels will be discussed, as well as a little bit more discussion of mapping and unmapping overhead. Overall time spent performing an operation is not enough, to get a full understanding of what's happening a profile will be layed out containing timings of the various sections that slow down these operations. The goal is to understand how time is being spent and at that point, further steps can be taken to mitigate the effects of these overheads.

Chapter 8

OpenCL Scheduling Overhead

Some preliminary profiling shows that there is a significant difference in the time it takes to run any given API function such as `clEnqueMapBuffer()` and `clEnqueueNDRangeKernel()` on the GPU, versus the total time it takes to complete a function from the time it is called in the host program to the time it is seen as completed from the host CPU.

OpenCL driver overhead accounts for much of this overhead. Queueing a submission can be very expensive due to to the OpenCL scheduling model. When any OpenCL API function such as `clEnqueueNDRangeKernel()` is called, it must be queued, submitted, run on the GPU, then sent back to the CPU. Each of these steps adds some overhead, which can end up being very significant.

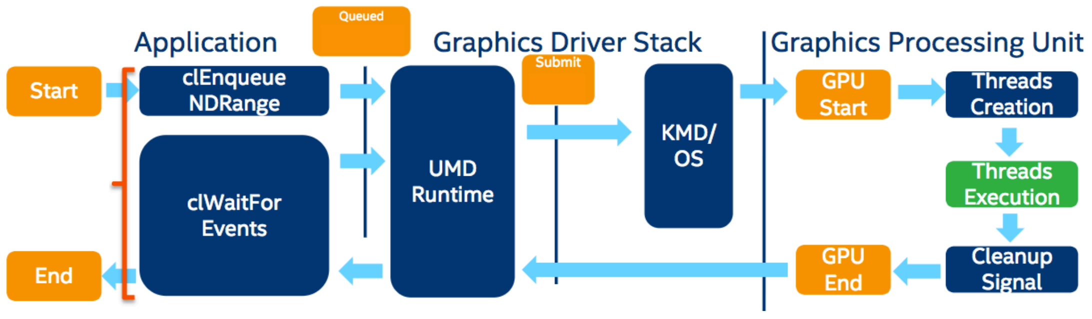
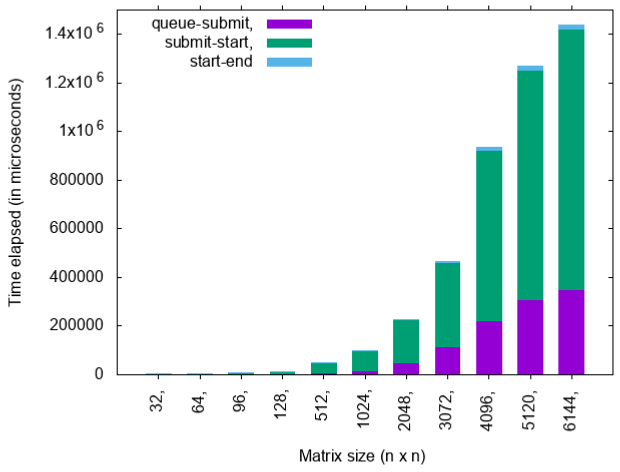


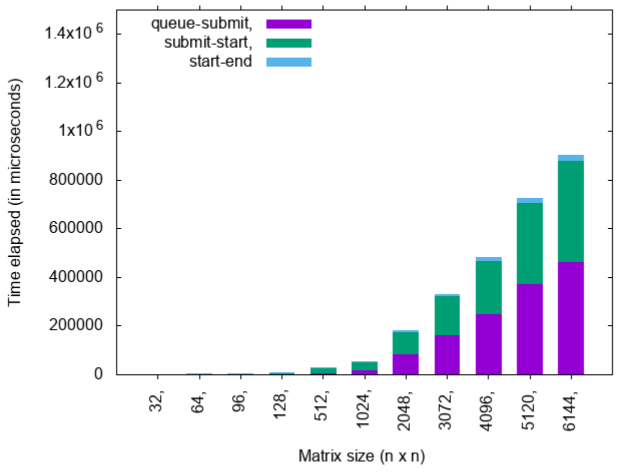
Figure 8.1: Current OpenCL scheduling model [36]

8.1 Profiling Scheduling Overhead For OpenCL Functions

These operations tend to take a lot of time and make small kernels very difficult to work with due to driver overhead. In the case of this program there is no need for a queue, as this program is simply running the same kernel over and over. Therefore profiling was done over the three main functions in our OpenCL program for the following times: between queue and submission, between submission and GPU start, and between GPU start and GPU end. The results below represent the total overhead for each function incurred in solving for an $n \times n$ matrix applied by each stage of scheduling.



(a) `clEnqueMapBuffer()` profile



(b) `clEnqueueUnmapMemObject()` profile

The map and unmap profiles show that virtually no time is spent in the GPU for these functions. At the end of the last chapter, it was noted that there is virtually no discussion about the overhead due to mapping and unmapping operations in the existing literature on the Mali T-6xx GPUs. Perhaps these results shed some light on why this is: mapping and unmapping operations have very little to do with the GPU. We already knew that much of mapping and unmapping has to do with cache maintenance which is a host-side operation that does not involve the GPU, however these results suggest that the time it takes to map and unmap is heavily dependent on how strong the CPU is, along with a host of other factors.

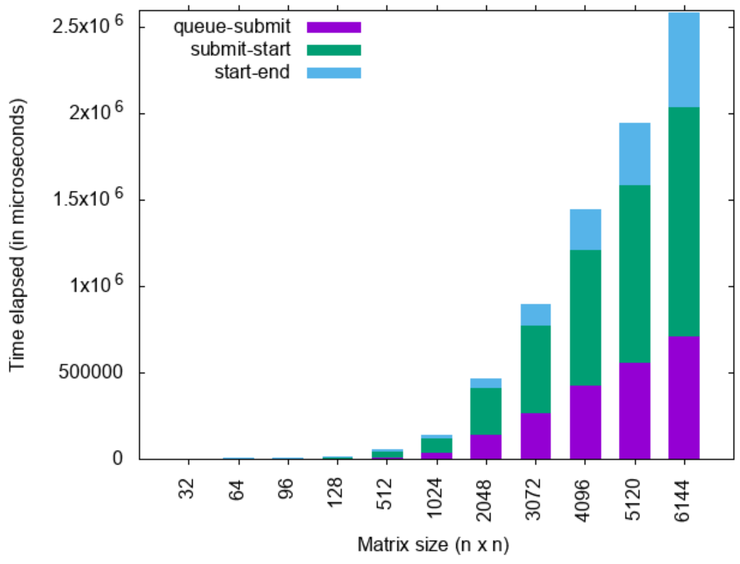


Figure 8.3: `clEnqueueNDRangeKernel()` profile

The `clEnqueueNDRangeKernel()` function spends much more time within the GPU than the map/unmap functions, as this is where calls to kernel 2 are happening. However, this still means that any optimisations of operations within the GPU such as improving the efficiency of reads and writes can only improve the speed of the blue section of the chart above, which accounts for only $\sim 20\%$ of total time incurred.

This means that if we can make the kernel 20× faster, which is already quite optimistic, we can still only achieve a total speedup of 19% of the total execution time of `clEnqueueNDRangeKernel()`. Therefore a classic approach that focuses only on kernel optimisations is not enough. We need to look at other approaches that allow us to reduce the queue-submit and submit-start times as well.

8.2 Techniques To Reduce Scheduling Overhead

In an ideal world, we would be able to do away with the queue entirely in order to submit directly to the GPU. We have no real need for a queue in this program because the same kernel is being sent to the GPU with each iteration of a loop.

Along with this, if it were possible to decrease the time spent submitting to the GPU, we could dramatically speed up our program without even bothering to try to optimise the kernel. Luckily, there are a number of methods available from OpenCL 2.0, the most notable being using GPU Daemon.

8.2.1 GPU Daemon

GPU Daemon allows the GPU to enqueue kernels without the need for host API interaction [36]. GPU Daemon is a kernel that is launched from the host, and then remains persistent on the GPU. The CPU communicates directly with active GPU threads, therefore bypassing the driver stack and allowing us to skip all queueing and submission aside from the first iteration.

Using GPU Daemon in Instant Mode, a speedup of overall kernel execution time including queueing, submission, processing and completion of compute tasks can be up to 19× [36]. Any GPU supporting OpenCL 2.0 can take advantage of this feature, however as the Mali T-604 GPU drivers only support OpenCL 1.1 it is unfortunately not able to take advantage of this technique.

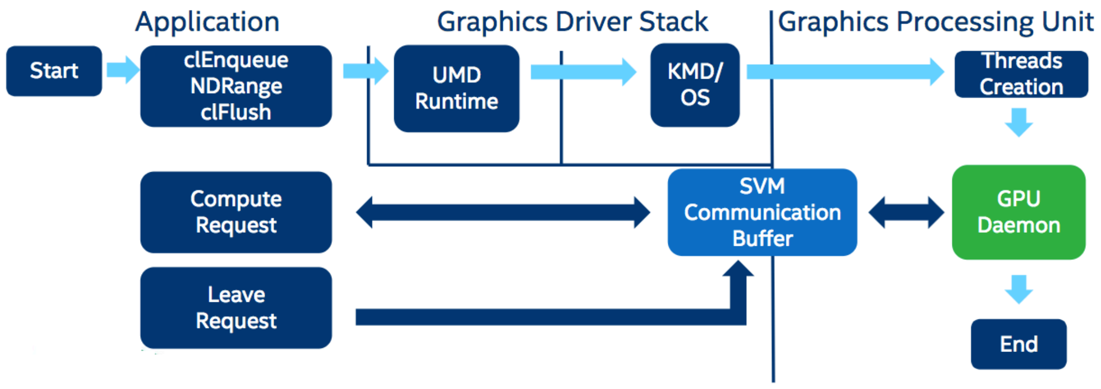


Figure 8.4: OpenCL scheduling using GPU Daemon with Instant Mode [36]

8.2.2 Vectorisation: The Next Best Option

These overheads have a very serious impact on execution times. Because our GPU does not support OpenCL 2.0 and therefore does not allow us to take advantage of GPU Daemon, we need to look for other ways to reduce overhead.

The next-best option available to us for reducing scheduling overhead is therefore to attempt to cut down on the number of calls to the kernel in general. By packing more operations in each kernel run, we can reduce the number of calls and therefore reduce the overhead due to queueing and submission that accumulates over hundreds of iterations. The technique we will use to achieve this is called vectorisation and it will be discussed in depth in the next chapter.

Chapter 9

Vectorisation

One of the most important hardware characteristic to take into account when optimising kernel code for the ARM Mali GPU Architecture is that the shader cores contain 128-bit wide vector registers [31].

These vector registers allow multiple arithmetic operations to be done simultaneously. We can utilise these in a technique called "vectorisation" in order to increase the number of operations done in each call to the kernel. By calling the kernel fewer times, the speed of the program should increase and the overhead due to kernel calls should decrease.

- First, an introduction to OpenCL vectorisation will be given, by means of a simple example to allow readers not familiar with this technique to understand the process.
- Then we must discuss memory organisation, since the way our memory is currently organised - in row-major form - prevents vectorisation of our code.
- After we have switched to column-major organisation, the details of the vectorisation process for our triangular solver will be discussed.
- Finally, after vectorisation has been explained, various vector sizes will be tested on our program to see which one performs best. Then we test overall program execution time and quantify how performance has been affected by vectorisation.

9.1 An Introduction To Vectorisation

A good way to understand vectorisation is by a simple example. This example will use values that allow us to focus on the concept rather than the maths, therefore these values will be more simple than those used in practice. In the section after this, an explanation of our vectorised kernel, complete with real values from our work, will be provided and explained.

9.1.1 Vectorising An Example OpenCL Program

In this section, an introductory guide to vectorisation will be given. This example is based on the example provided in 2.3.1 on page 13, where a C program was parallelised via OpenCL. Now we build on this example by vectorising that OpenCL code. As a reminder, the kernel was written as

```
__kernel void vec_example(__global int* input_arr_one,
                          __global int* input_arr_two,
                          __global int* output_arr)
{
    int i = get_global_id(0);
    output_arr[i] = input_arr_one[i]+input_arr_two[i];
}
```

Listing 9.1: Toy OpenCL kernel

Where each array was of size 80, and therefore the kernel was run 80 times.

Assuming we are running this example program on hardware such as the Mali T-604 GPU that makes vectorisation possible. Like the T-604, let's assume that the GPU executing this example OpenCL program contains 128-bit wide vector registers. Therefore data types or *vector types* such as `char16`, `short8`, `int4`, and `float4` are available for use. We can also perform operations such as floating point add and floating point multiply with these vector types.

The concept behind vectorisation is simple. In this example, the inputs and outputs of the program are arrays of integers. Therefore as we have an `int4` data vector type available we can simply perform four integer operations at a time instead of just one.

```
__kernel void vec_example(__global int* input_arr_one,
                          __global int* input_arr_two,
                          __global int* output_arr)
{
    int i = get_global_id(0)*4;

    // Load four consecutive integers from input_arr_one
    int4 input_arr_one_vec = vload4(i, input_arr_one[i]);

    // Load four consecutive integers from input_arr_two
    int4 input_arr_two_vec = vload4(i, input_arr_two[i]);

    // Perform a vector addition, then store the four results
    // into addresses in output_arr
    vstore4(input_arr_one_vec + input_arr_two_vec, i, output_arr);
}
```

Listing 9.2: Example OpenCL kernel

After the kernel has been vectorised, the final parameter we need to take into account is the global size. We have made the kernel 4× more efficient, therefore we need to call the kernel 4 times less. We account for this by reducing the global size by a factor of 4. So we take our previous global size of 80, since we had 80 total operations, and divide by 4 to get our vectorised global work size of 20. This is great as we have divided the number of calls to the kernel by 4, which means that we have also theoretically reduced our total kernel launch time by 4.

In the next few sections vectorisation of our triangular solver program will be discussed. First we must think about memory organisation, as the way our memory is organised currently does not allow vectorisation. After we address this issue, we will then proceed to test various vector sizes on kernel 2 and see which one is the most efficient. We will then pick the most efficient one and see how the overall program performs with it.

9.2 Memory Organisation

The first concern we must deal with before we vectorise our current OpenCL implementation is that in order to vectorise we must have all data that is accessed in contiguous sequential memory. We have thus far used row-major order to store our 2D matrix as 1D arrays

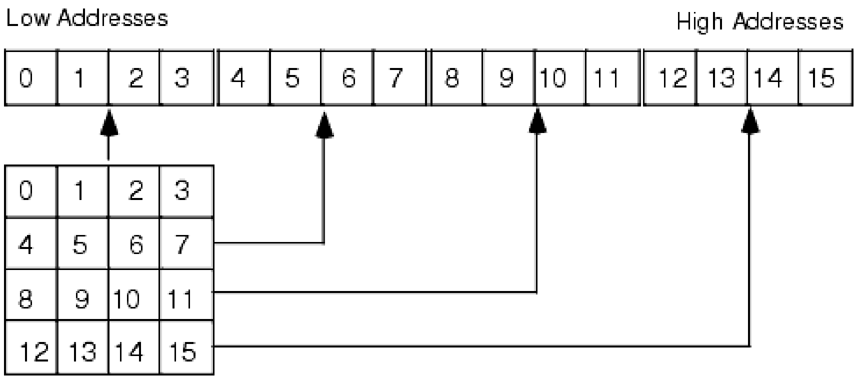


Figure 9.1: 2D matrices stored as 1D row-major arrays

The problem with row-major storage for our left-hand matrix as discussed in Chapter 5.3, is that when we vectorise we will need to access multiple memory locations at once.

9.2.1 Switching To Column-Major Storage

Currently, we perform a single operation in Kernel 2 like so:

```
k[row] = k[row]-A[row*hA+currentCol]*k[currentCol];
```

For an example 4×4 array, four subsequent accesses on matrix A, as would be the case with vectorised code with a vector size of 4, looks like:

```
// Start with currentCol = 0 and row = 0
// currentCol remains at 0
// row iterates, taking values of 0, 1, 2, 3

// A[0]
k[0] = k[0]-A[0*4+0]*k[0];

// A[4]
k[1] = k[1]-A[1*4+0]*k[0];

// A[8]
k[2] = k[2]-A[2*4+0]*k[0];

// A[12]
k[3] = k[3]-A[3*4+0]*k[0];
```

In memory, this access looks like this:

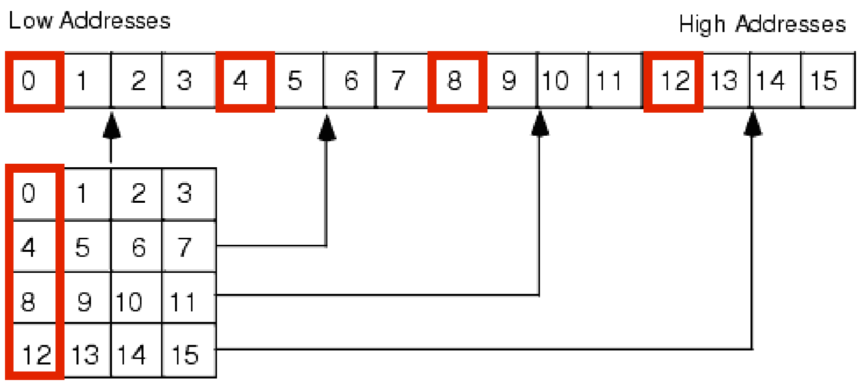


Figure 9.2: Vectorised access of 1D row-major array when accessing array A

Due to the way that the matrix is currently layed out - in row-major order - each subsequent access of A needs to skip the width of a row minus 1, in this case $4-1=3$, in order to get to the next memory location. This makes it impossible to vectorise the array because data in A that needs to be accessed in one vector operation are not in directly neighbouring memory locations.

The solution to this is to switch to column-major storage of the matrix A. This means that instead of storing by row as we did before, we store by column. This fixes the problem and allows for vectorisation, as shown below.

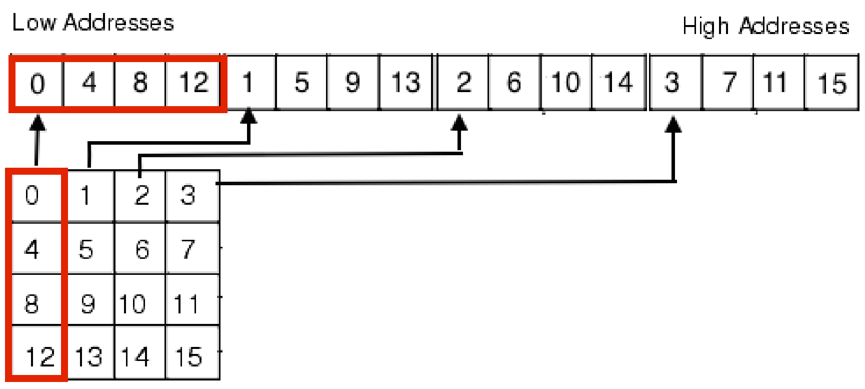


Figure 9.3: Vectorised access of 1D column-major array

9.2.2 Performance of Column-Major

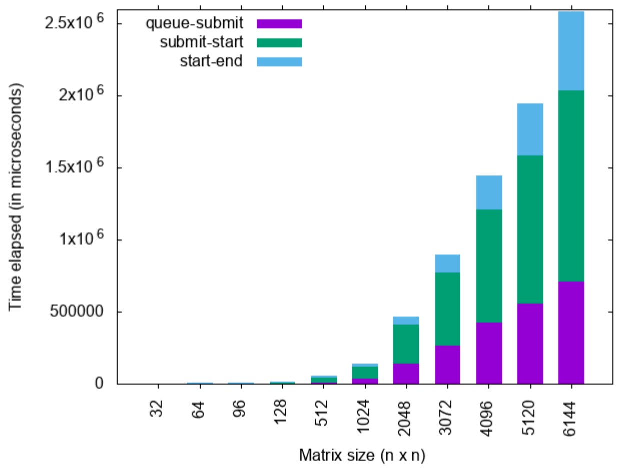
It's not enough to just switch the format of the data storage to column-major, the kernels themselves must also be modified in order to perform operations correctly on data in column-major order. After adjustment, the column-major implementation of kernel 2 is

```
__kernel void kernel2(__global float* A, __global float* k,
                      int wA, int currentRow)
{
   int col = get_global_id(0)+currentRow+1;
          // Condition: as long as we haven't overrun the size of
          // the matrix
          if(col<wA ){
            /** KERNEL #2 **/
            k[col] = k[col]-A[currentRow*wA+col]*k[currentRow];
          }
}
```

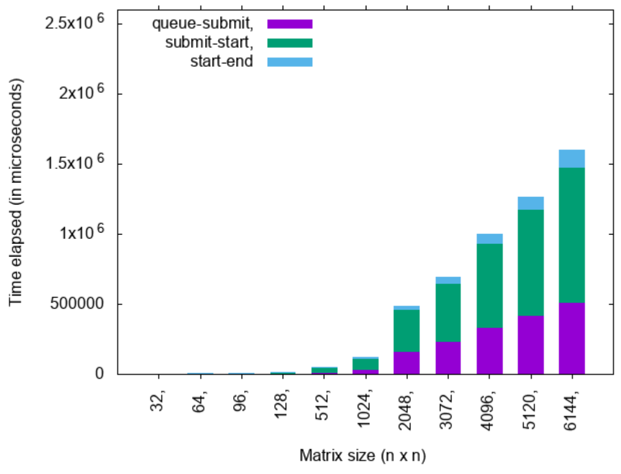
called from the host like so

```
for(currentRow = 0; currentRow < n; currentRow++){
    /* Enqueue Kernel 2 */
}
```

An added benefit of switching to column-major storage for matrix A is that we can take advantage of *spatial locality* in order to improve the effectiveness of caching. By putting data close together in the memory, we reduce the penalty of cache misses when reading values of A. This is helpful because when the cache misses, it must load new data into the cache which is a slow operation. Therefore because we have improved this aspect, we can expect this switch to column-major storage to provide an improvement in kernel execution time. This can be measured by profiling the `clEnqueueNDRangeKernel()` function for Kernel 2 again.



(a) `Row-major`



(b) `Column-major`

Figure 9.4: `clEnqueueNDRangeKernel()` profiles for kernel 2

A clear speedup in all time segments can be noted, but most pronounced is in the time elapsed between GPU start and GPU end. This is where speedups were expected most, as this is where kernel execution happens and therefore this is where reads from A happen where column-major storage has improved caching to a very large degree.

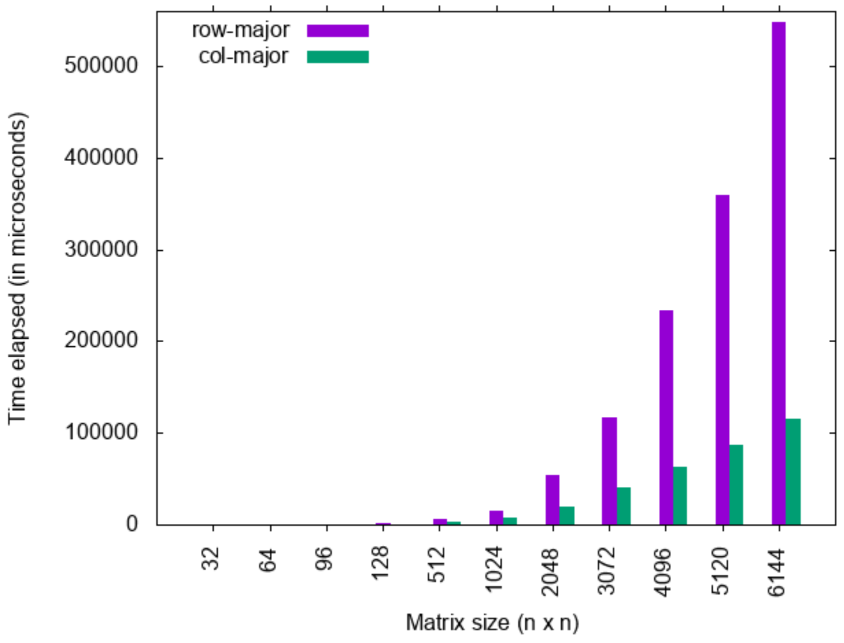


Figure 9.5: Comparison of time spent from GPU start to GPU end in row-major vs column-major program

9.3 Vectorising The OpenCL Triangular Solver

Now that column-major storage has been implemented, this section will focus on applying the vectorisation technique, introduced earlier in the chapter, to the triangular solver OpenCL program. Once vectorisation has been applied, profiling will be performed to see to what extent vectorisation can improve performance when repeatedly enqueuing kernel 2.

ARM Mali GPU shader cores contain 128-bit wide vector registers, which allow multiple arithmetic operations to be done simultaneously. These shader cores with 128-bit vector hardware can handle arithmetic operations of four 32-bit integers or floats simultaneously [31].

These 128-bit vector data types include `char16`, `short8`, `int4`, `float4` and operations available on these data types include floating point add and floating point multiply, both of which are crucial to the back substitution step.

In the next section we'll discuss the first step where kernel 2 must be modified to use vector operations. After, we'll try various vector sizes and test to determine which one is the most effective for our implementation. And finally, we will discuss the overall effect on kernel execution time and scheduling overhead.

9.3.1 Modifying The Kernel For Vector Operations

Examining the equation for the back-substitution step, we can observe that there are two operations: a floating point multiply and a floating point add (the subtraction is considered as an addition operation within the Arithmetic Logic Unit within the GPU). The multiplication and subtraction will be performed on vector data types, and thus become vector operations.

$$R_i \leftarrow R_i - a_{ij} \times R_{i-1} \tag{9.1}$$

Observe also that we have two reads R_i and R_{i-1}, as well as one write back to R_i after the multiplication and the subtraction are completed. The reads will need to become *vector loads* and the writes will need to become *vector stores*.

Vector reads are done as follows:

```
float4 vector_value = vload4(i, k);
float4 matrix_value = vload4(i, A);
```

And vector operations, combined with the vector store is done as follows:

```
/*
 * Perform the multiplication and subtraction, then store
 * back into the address: solutionVec + i * 4.
 */
vstore4(vector_value - mult_value * matrix_value, i, solutionVec);
```

9.3.2 Changing the kernel

The vectorised code for vectors of length 4 looks like this:

```

__kernel void kernel2_vec4(__global float* A, __global float* k,
                                 int wA, int currentRow)
{
   int i = get_global_id(0)+1;
   int vectorSize = 4;
   int numberOfVectorOps = (wA-currentRow+1)/vectorSize;
   int numberOfRegularOps = (wA-currentRow+1)%vectorSize;
  /*
   * All operations that are perfectly divisible by the vectorSize
   * can use vector operations
   * The rest must be done with regular operations
   */
   if(i < numberOfVectorOps+1){
          i = get_global_id(0)*vectorSize+currentRow+1;

          float mult = k[currentRow];
          float4 v_mult = (float4)(mult, mult, mult, mult);

           /*
            * Load solution vector and input matrix
            * into float4 vector types
            */
           int mat_offset = currentRow*wA+i;
           float4 matrix_value = vload4(0, A+mat_offset);
           float4 vector_value = vload4(0, k+i);

           // Perform vector mult and sub, then vector store
           // k[col] = k[col]-A[currentRow*wA+col]*k[currentRow];
           vstore4(vector_value - matrix_value * v_mult, 0, k+i);
   }else if(i < numberOfVectorOps+numberOfRegularOps){
       int col = i+currentRow+vectorSize*numberOfVectorOps;
       k[col] = k[col]-A[currentRow*wA+col]*k[currentRow];
   }
}
```

Listing 9.3: Vectorised Kernel 2, with vector width of 4

By changing all `float4`, `vload4`, and `vstore4` operations to their corresponding 8-wide operations, and also changing the vector size, this kernel can be easily changed to support vector width of 8.

9.3.3 Choosing A Vector Size For The Triangular Solver

It's important to note here that for each time we make changes to the kernel code, the local work size must be checked via `clGetKernelWorkGroupInfo()` to ensure that we are running the local work group size at its maximum, 256 work items. This was discussed in more detail in chapter 6.2 but is worth bringing up again as it is an important consideration for every change of the kernel.

Running `clGetDeviceInfo()` with the `CL_DEVICE_PREFERRED_VECTOR_WIDTH_INT` flag, gives us that the preferred vector width on the Mali T-604 is 4. Due to this, we might suspect that vector operations of size 4 will perform best, however there is evidence to support that vector sizes of 8 or 16 can perform better on this GPU [28]. If achievable performance is not necessarily bound to a particular vector size, but can vary from case to case on this GPU [29], then it is worth experimenting with all the vector sizes supported in OpenCL 1.1.

We adjusted the kernel in order to support vector operations of size 4, 8, and 16 in order to see which performs best. However while testing these implementations, the kernel using vector size of 16 showed a reduction in local work group size from our maximum of $I = 256$ to half of the work items at $I = 128$. This also subsequently caused the kernel to fail with a `CL_OUT_OF_RESOURCES` error. This is likely due to the increase in the size of our vectors in the kernel, and thus each kernel instance needs more registers than are available to it. Therefore we have tested vectors of size 4 and 8, but were unable for those of size 16.

9.3.4 Performance Evaluation

Both vector sizes of 4 and 8 run local work sizes at the maximum $I = 256$.

Now, in order to figure out which size is faster we measure the total execution time of the kernel 2 loop, not just the time of the various scheduling sections of the loop. The three categories will be the basic column-major implementation, the implementation using vector size of 4, and the implementation using vector size of 8.

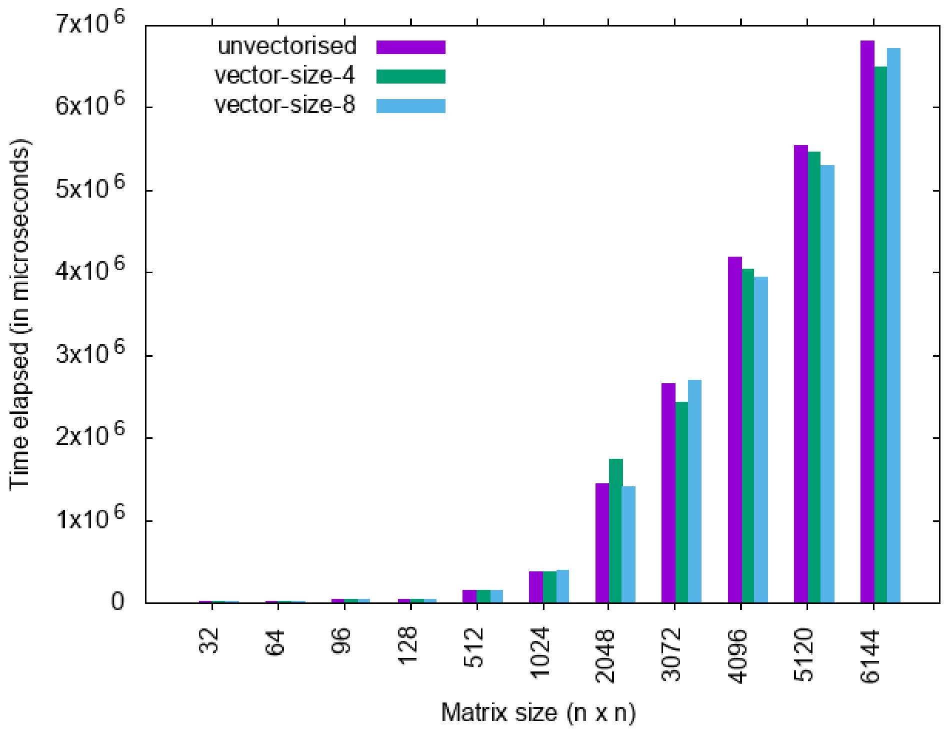
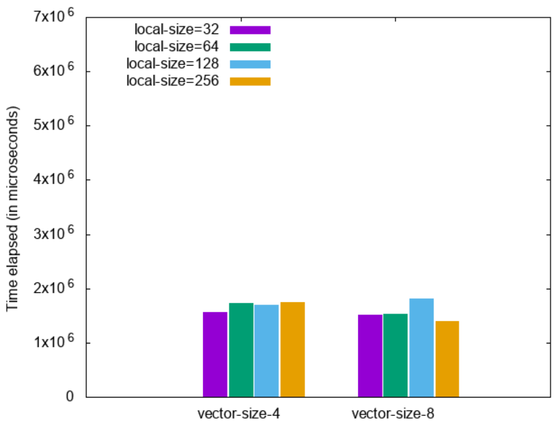
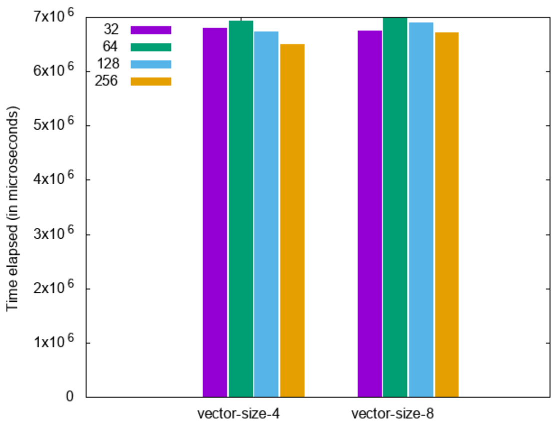


Figure 9.6: The total execution time over the kernel 2 loop, comparing the baseline column-major implementation to vector implementations of size 4 and 8.

The result are surprising as the assumption we held, and corroborated in multiple literature sources, was that total execution time would almost certainly improve significantly with vectorisation. This is clearly not true in our case, but the question is why. The first thought was to check the global work size - this size has not decreased noticeably due to the rounding discussed in chapter 6.3.1. Although we have made the kernels more efficient, we still have many ”empty” work items running at the tail end of each global work group, in order to round up to satisfy the equation `global work size` = `local work size` × `num of shader cores`. Our global work size is a function of local work size and therefore cannot be changed directly, however we can adjust our global size indirectly by adjusting our local work size (number of shader cores is a constant - 4). There is a possibility that we can increase execution time by bringing these work items down. As we are currently using 256 as our local work size, we can decrease the number of ”empty work items” due to rounding by bringing this number down. The literature maintains that work items should stay as high as possible, however as we have relatively small work groups (thousands) this is not a standard scenario where we are running a very large number of work-items, in the range of hundreds of thousands, and therefore can be experimented with.



(a) n=2048



(b) n=6144

Figure 9.7: The effect of varying local work sizes (32, 64, 128, 256) for two different input matrix sizes in our vectorised implementation

Tests show that varying the work group sizes doesn't create a clear, significant increase in performance for any given local work size. This forces us to conclude that while overall GPU execution time may be affected positively, an overall speedup simply will not observed due to vectorisation.

Therefore the most profound result found in this chapter was that switching to a column-major implementation actually allows the kernel 2 loop to execute faster, while vectorisation does not give a clear performance gain. This runs contrary to documentation and literature, however it is suspected that the lack of performance gain is due to the small global work sizes that are used in the triangular solve. We will discuss these results even further in the next chapter, and describe the next steps that could be taken following on from the results gathered and observations made in this paper.

Chapter 10

Discussion

10.0.1 Final Results

The results present a clear picture: with the cost of mapping and unmapping operations, combined with the overhead imposed upon the program by the OpenCL scheduling model, the parallel triangular solver simply will not be able to perform fast enough to beat CPU BLAS implementations on the Mali T-604 GPU. This leaves many directions in which to follow this project to gain additional insight into the findings of this paper.

We derived the average times of mapping/unmapping buffers over the kernel 2 loop earlier in the report. Given that these mapping/unmapping operations are independent of any of the changes we have made to the kernel, we can graph them as a percentage of overall execution in our kernel 2 loop, using the baseline column-major implementation of kernel 2. This will allow us to see exactly what proportion of overall execution time is consumed by the mapping/unmapping operations, what proportion is spent within the GPU, and the remainder - the overhead. These proportions of overall kernel 2 loop execution time are shown in fig 10.1.

We previously concluded that increases in mapping/unmapping time levels off as size increases. However even with this tapering off at large matrix sizes, mapping and unmapping continue to account for up a huge percentage of overall time and thus significantly limits the potential for improving the kernel execution time via various kernel optimisations.

10.0.2 Further Steps

One option is to try the implementation on many different integrated GPUs, perhaps across the range of Mali GPU products, and study the impact on performance, specifically with regards to mapping/unmapping operations. Furthermore, some newer GPUs, including those from the Mali line, will also support OpenCL 2.0 or higher. This would present an option not available to us on the

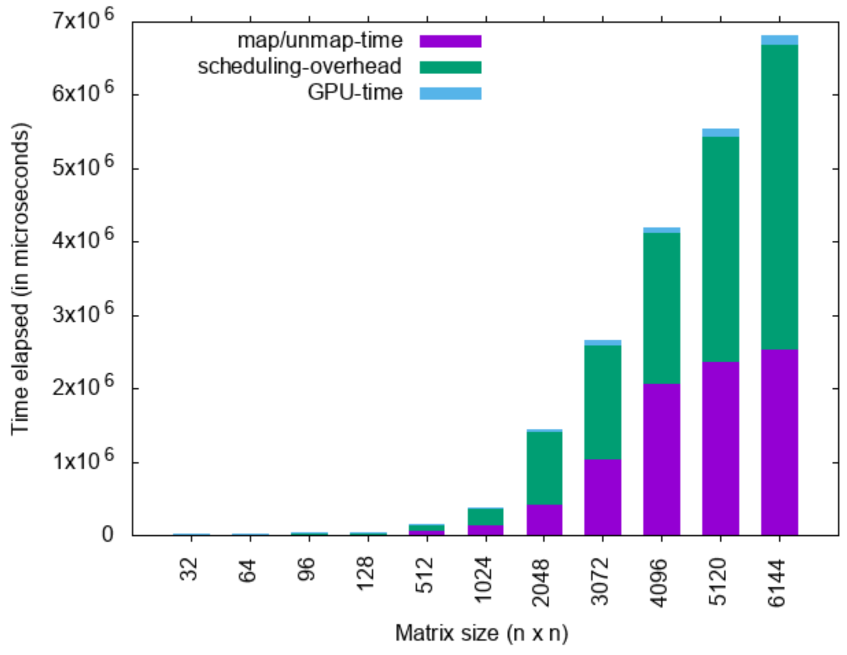


Figure 10.1: Mapping/unmapping, total scheduling overhead, and GPU execution time as proportions of overall execution time.

current T-604 GPU - GPU Daemon can be used to reduce the overhead due to OpenCL scheduling. This could be incorporated and then tested against undaemonised implementations to see how much of the OpenCL scheduling overhead can indeed be eliminated.

At that point, there would be the possibility to investigate if there is a framework better suited to this implementation than OpenCL. There is evidence to support the claim that CUDA may offer a more lean approach to parallelisation, and therefore an option is to port this implementation to CUDA and then test on an integrated GPU that supports CUDA. While CUDA is not portable and does not provide cross-platform support, it does have several advantages pertinent to this discussion [27]. These include, in many cases, faster memory sharing and read backs, along with minimal threads creation overhead [37, 38], both of which are desirable characteristics to us in this implementation. An added bonus is that CUDA has similar mapping commands to OpenCL - for instance the `cudaGraphicsMapResource()` command [39]. Therefore a unified memory structure could still be taken advantage of via memory mapping, within the CUDA framework.

Much of the literature on parallel triangular solvers provide implemenetations based on CUDA for their solvers [21, 22, 23, 24, 25], presumably as a result of the advantages previously discussed that are associated with using the framework. CUDA was not available to us on the GPU used, however an informative experiment would be to compare two GPUs with similar specifications, with one GPU running OpenCL and the other running CUDA, on the parallel solver presented in this paper. This would be useful in gaining a better understanding of the overhead costs associated with our kernel calls in CUDA. The comparison would also reveal if the time costs are less than those incurred by using OpenCL, therefore making CUDA a more robust framework for developing triangular solver implementations or indeed general implementations of parallel programs that run many subsequent small global work-groups on lightweight integrated GPUs.

It is important to note here that only NVIDIA products can currently run CUDA as it is propritary software, therefore testing on a mobile platform is limited to NVIDIA's line of mobile processors - the Tegra SoC line containing ultra low-power GPUs [40]. This limits the range of GPUs available for such an experiment as mentioned above, but does not detract from the usefulness of the experiment.

Chapter 11

Conclusion

The overarching goal set out for this project was to optimise the Gaussian process regression by means of parallelisation via an integrated GPU. In our pursuit of an optimised GP regression, we layed out several aims for the project.

The first aim was to produce an outline, leading to the discovery that a large amount of time is being spent within the matrix inversion step, which, in our algorithm's case, is the Cholesky factorisation. While this was discussed in theory, our tests show that a large majority of total program time - over 80% - is dominated by this step.

This finding then lead to the conclusion that overall execution time could be drastically decreased if the Cholesky factorisation, which in practice is implemented as a triangular solver with some modifications, was parallelised effectively. And indeed there was evidence in the literature that this had been done albeit there was no mention of an effective implementation of a triangular solve on a low-power integrated GPU such as our Mali T-604.

The most interesting discoveries, however, came after the solver was implemented in OpenCL. Our implementation involves iterating over many global work-groups that are small - generally somewhere between 0 and 7000 work-items. This means that any overhead applied when launching a kernel would be compounded over several thousand launches, and therefore total overhead could drastically reduce the overall performance time of our parallel solver. This was indeed what the findings revealed - that minimal time is spent in the GPU executing the kernel, but instead the majority is being spent either transferring data to/from the GPU or otherwise spent in various stages of the OpenCL scheduling model, amounting to unwanted overhead.

Several other discoveries were made when testing various optimisation on our implementation. Transferring data to/from the GPU was found to be faster when mapping buffers rather than copying them, and increasingly effective as matrix size increased. Moreover, column-major storage was shown to make our kernel much more efficient while also decreasing scheduling overhead when launching the kernel. And perhaps most unexpectedly of all, and in direct contradiction to much of the literature on the Mali T-604, we discovered that vectorisation did not noticeably affect execution time of the kernel in our implementation.

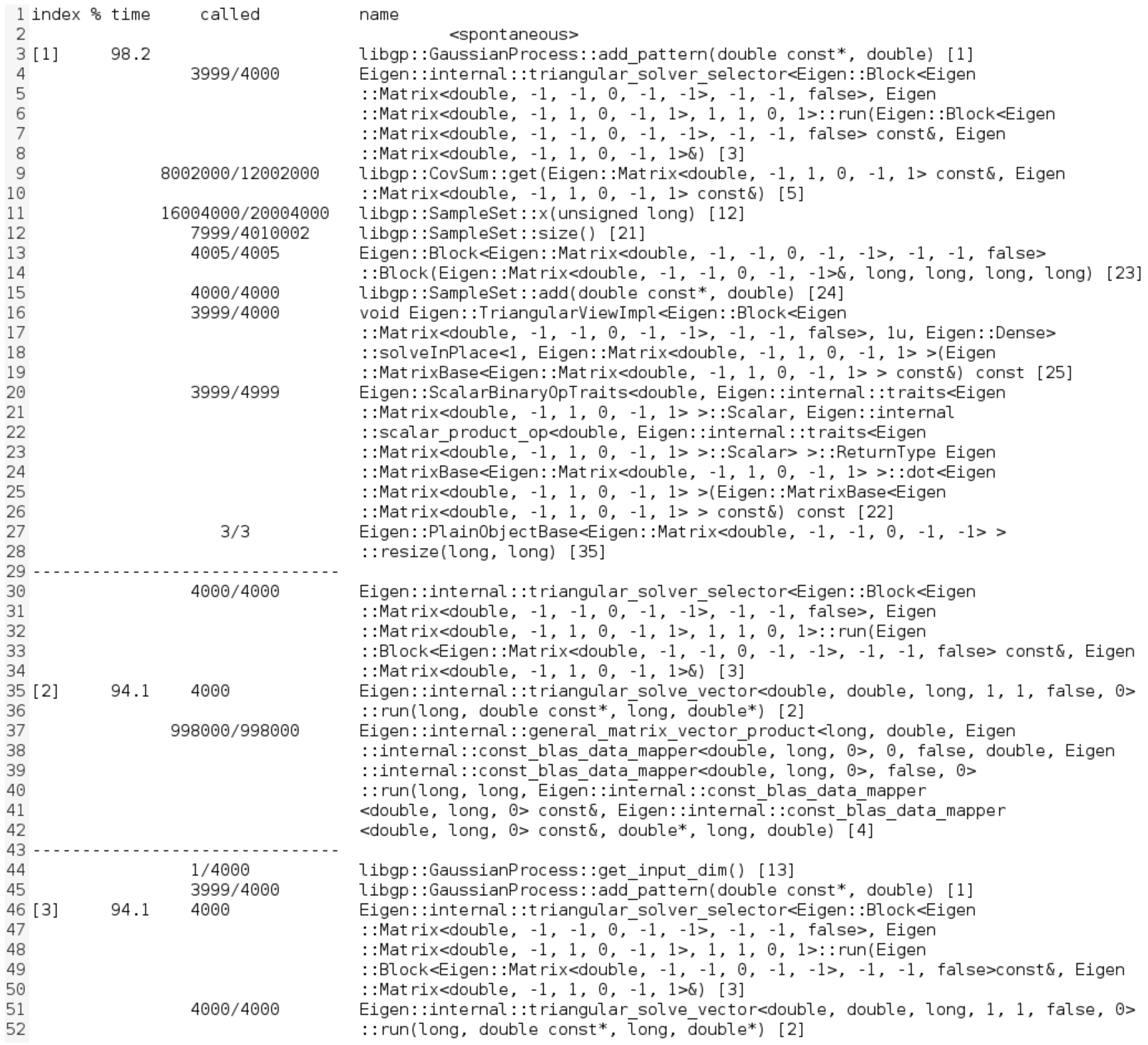
This paper demonstrates that the most important aspect of optimising the parallel triangular solve on an integrated GPU like the Mali T-604, in order of importance, are 1) efficiently sending data to/from the GPU, 2) decreasing OpenCL scheduling overhead, and 3) organising memory effectively in order to improve caching performance.

There are many further steps that can be taken based on the findings of this paper, that will lead to improvements in the execution time of the parallel triangular solver as discussed in this paper. These include running the implementation presented in this paper on faster integrated GPUs that will ideally lead to better mapping/unmapping times, using various techniques for reducing scheduling overhead available in OpenCL version 2.0 and later, and possibly even porting the implementation to CUDA.

Overall, we can conclude that the Gaussian process regression does not prove to be robust to parallelisation by means of an OpenCL triangular solver on the current Mali T-604 GPU hardware, as was available for this paper. Advances in the field of GPGPUs will continue to be made, bringing improvements in integrated GPU performance as well as advances in parallel programming frameworks like OpenCL and CUDA. Therefore we can expect that with faster GPUs and more study, the successful optimisation of Gaussian process regessions for integrated GPUs is likely to be achieved.

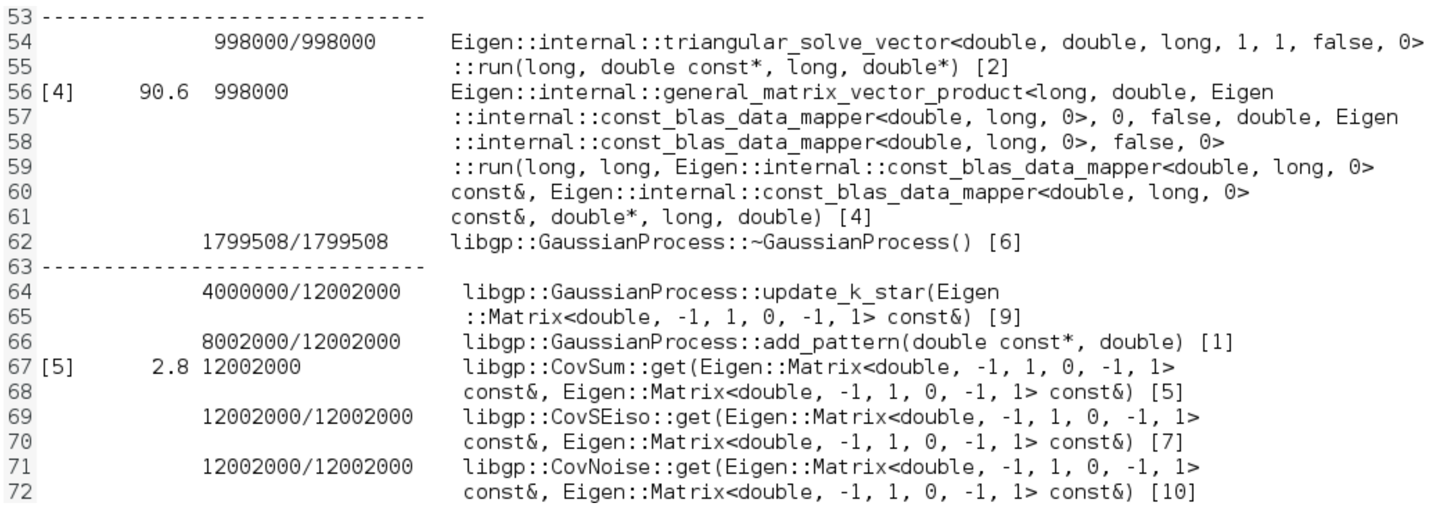
Appendix A

11.0.1 Profiler Call Graph



```
index % time     called          name
                                          <spontaneous>
[1]     98.2                     libgp::GaussianProcess::add_pattern(double const*, double) [1]
                3999/4000        Eigen::internal::triangular_solver_selector<Eigen::Block<Eigen
                                 ::Matrix<double, -1, -1, 0, -1, -1>, -1, -1, false>, Eigen
                                 ::Matrix<double, -1, 1, 0, -1, 1>, 1, 1, 0, 1>::run(Eigen::Block<Eigen
                                 ::Matrix<double, -1, -1, 0, -1, -1>, -1, -1, false> const&, Eigen
                                 ::Matrix<double, -1, 1, 0, -1, 1>&) [3]
           8002000/12002000      libgp::CovSum::get(Eigen::Matrix<double, -1, 1, 0, -1, 1> const&, Eigen
                                 ::Matrix<double, -1, 1, 0, -1, 1> const&) [5]
           16004000/20004000     libgp::SampleSet::x(unsigned long) [12]
                7999/4010002     libgp::SampleSet::size() [21]
                4005/4005        Eigen::Block<Eigen::Matrix<double, -1, -1, 0, -1, -1>, -1, -1, false>
                                 ::Block(Eigen::Matrix<double, -1, -1, 0, -1, -1>&, long, long, long, long) [23]
                4000/4000        libgp::SampleSet::add(double const*, double) [24]
                3999/4000        void Eigen::TriangularViewImpl<Eigen::Block<Eigen
                                 ::Matrix<double, -1, -1, 0, -1, -1>, -1, -1, false>, 1u, Eigen::Dense>
                                 ::solveInPlace<1, Eigen::Matrix<double, -1, 1, 0, -1, 1> >(Eigen
                                 ::MatrixBase<Eigen::Matrix<double, -1, 1, 0, -1, 1> > const&) const [25]
                3999/4999        Eigen::ScalarBinaryOpTraits<double, Eigen::internal::traits<Eigen
                                 ::Matrix<double, -1, 1, 0, -1, 1> >::Scalar, Eigen::internal
                                 ::scalar_product_op<double, Eigen::internal::traits<Eigen
                                 ::Matrix<double, -1, 1, 0, -1, 1> >::Scalar> >::ReturnType Eigen
                                 ::MatrixBase<Eigen::Matrix<double, -1, 1, 0, -1, 1> >::dot<Eigen
                                 ::Matrix<double, -1, 1, 0, -1, 1> >(Eigen::MatrixBase<Eigen
                                 ::Matrix<double, -1, 1, 0, -1, 1> > const&) const [22]
                   3/3           Eigen::PlainObjectBase<Eigen::Matrix<double, -1, -1, 0, -1, -1> >
                                 ::resize(long, long) [35]
-------------------------------
                4000/4000        Eigen::internal::triangular_solver_selector<Eigen::Block<Eigen
                                 ::Matrix<double, -1, -1, 0, -1, -1>, -1, -1, false>, Eigen
                                 ::Matrix<double, -1, 1, 0, -1, 1>, 1, 1, 0, 1>::run(Eigen
                                 ::Block<Eigen::Matrix<double, -1, -1, 0, -1, -1>, -1, -1, false> const&, Eigen
                                 ::Matrix<double, -1, 1, 0, -1, 1>&) [3]
[2]     94.1    4000             Eigen::internal::triangular_solve_vector<double, double, long, 1, 1, false, 0>
                                 ::run(long, double const*, long, double*) [2]
             998000/998000       Eigen::internal::general_matrix_vector_product<long, double, Eigen
                                 ::internal::const_blas_data_mapper<double, long, 0>, 0, false, double, Eigen
                                 ::internal::const_blas_data_mapper<double, long, 0>, false, 0>
                                 ::run(long, long, Eigen::internal::const_blas_data_mapper
                                 <double, long, 0> const&, Eigen::internal::const_blas_data_mapper
                                 <double, long, 0> const&, double*, long, double) [4]
-------------------------------
                1/4000           libgp::GaussianProcess::get_input_dim() [13]
                3999/4000        libgp::GaussianProcess::add_pattern(double const*, double) [1]
[3]     94.1    4000             Eigen::internal::triangular_solver_selector<Eigen::Block<Eigen
                                 ::Matrix<double, -1, -1, 0, -1, -1>, -1, -1, false>, Eigen
                                 ::Matrix<double, -1, 1, 0, -1, 1>, 1, 1, 0, 1>::run(Eigen
                                 ::Block<Eigen::Matrix<double, -1, -1, 0, -1, -1>, -1, -1, false>const&, Eigen
                                 ::Matrix<double, -1, 1, 0, -1, 1>&) [3]
                4000/4000        Eigen::internal::triangular_solve_vector<double, double, long, 1, 1, false, 0>
                                 ::run(long, double const*, long, double*) [2]
```

Figure 11.1: Call graph part 1, produced by Gprof



```
-------------------------------
             998000/998000      Eigen::internal::triangular_solve_vector<double, double, long, 1, 1, false, 0>
                                ::run(long, double const*, long, double*) [2]
[4]     90.6  998000            Eigen::internal::general_matrix_vector_product<long, double, Eigen
                                ::internal::const_blas_data_mapper<double, long, 0>, 0, false, double, Eigen
                                ::internal::const_blas_data_mapper<double, long, 0>, false, 0>
                                ::run(long, long, Eigen::internal::const_blas_data_mapper<double, long, 0>
                                const&, Eigen::internal::const_blas_data_mapper<double, long, 0>
                                const&, double*, long, double) [4]
            1799508/1799508     libgp::GaussianProcess::~GaussianProcess() [6]
-------------------------------
            4000000/12002000     libgp::GaussianProcess::update_k_star(Eigen
                                 ::Matrix<double, -1, 1, 0, -1, 1> const&) [9]
            8002000/12002000     libgp::GaussianProcess::add_pattern(double const*, double) [1]
[5]      2.8 12002000            libgp::CovSum::get(Eigen::Matrix<double, -1, 1, 0, -1, 1>
                                 const&, Eigen::Matrix<double, -1, 1, 0, -1, 1> const&) [5]
            12002000/12002000    libgp::CovSEiso::get(Eigen::Matrix<double, -1, 1, 0, -1, 1>
                                 const&, Eigen::Matrix<double, -1, 1, 0, -1, 1> const&) [7]
            12002000/12002000    libgp::CovNoise::get(Eigen::Matrix<double, -1, 1, 0, -1, 1>
                                 const&, Eigen::Matrix<double, -1, 1, 0, -1, 1> const&) [10]
```

Figure 11.2: Call graph part 2, produced by Gprof

Bibliography

[1] Barbara Frank, Cyrill Stachniss, Nichola Abdo, and Wolfram Burgard. Using gaussian process regression for efficient motion planning in environments with deformable objects. In *Automated Action Planning for Autonomous Mobile Robots*, 2011. http://www2.informatik.uni-freiburg.de/~stachnis/pdf/frank11pamr.pdf.

[2] Shrihari Vasudevan, Fabio Ramos, Eric Nettleton, and Hugh Durrant-Whyte. Gaussian process modeling of large-scale terrain. *Journal of Field Robotics*, 26(10):812–840, 2009. http://dx.doi.org/10.1002/rob.20309.

[3] Tongtong Chen, Bin Dai, Ruili Wang, and Daxue Liu. Gaussian-process-based real-time ground segmentation for autonomous land vehicles. *Journal of Intelligent & Robotic Systems*, 76(3):563–582, 2014. http://dx.doi.org/10.1007/s10846-013-9889-4.

[4] NotebookCheck. Arm mali-t604 mp4. https://www.notebookcheck.net/ARM-Mali-T604-MP4.116274.0.html, 2016.

[5] NotebookCheck. Arm mali-t628 mp6. https://www.notebookcheck.net/ARM-Mali-T628-MP6.110718.0.html, 2016.

[6] NotebookCheck. Arm mali-t760 mp8. https://www.notebookcheck.net/ARM-Mali-T760-MP8.140006.0.html, 2016.

[7] NotebookCheck. Arm mali-t880 mp12. https://www.notebookcheck.net/ARM-Mali-T880-MP12-Benchmarks.160645.0.html, 2016.

[8] NotebookCheck. Arm mali-g71 mp8. https://www.notebookcheck.net/ARM-Mali-G71-MP8.185277.0.html, 2016.

[9] Ian Bratt. The arm mali-t880 mobile gpu. Intel Corporation, https://www.hotchips.org/wp-content/uploads/hc_archives/hc27/HC27.25-Tuesday-Epub/HC27.25.50-GPU-Epub/HC27.25.531-Mali-T880-Bratt-ARM-2015_08_23.pdf.

[10] Manuel Blum. *Manuel Blum*, 2016 (accessed January 25, 2017). http://ml.informatik.unifreiburg.de/people/blum/info.

[11] CE. Rasmussen and CKI. Williams. *Gaussian Processes for Machine Learning*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, USA, January 2006. http://www.gaussianprocess.org/gpml/.

[12] Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000. http://ieeexplore.ieee.org/document/876288/.

[13] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[14] Kristen Boydstun. Introduction opencl. CalTech Lecture, http://www.tapir.caltech.edu/~kboyds/OpenCL/opencl.pdf.

[15] Simon McIntosh-Smith and Tom Deakin. Hands on opencl. https://www.slideshare.net/vladimirstarostenkov/hands-on-opencl. University of Bristol.

[16] Lloyd N Trefethen and David Bau III. *Numerical Linear Algebra*. Siam, 1997.

[17] G. Zielke. Horn, r. a.; johnson, c. r., matrix analysis. cambridge etc., cambridge university press 1985. xiii, 561 s., isbn 0-521-30586-1. *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift fr Angewandte Mathematik und Mechanik*, 67(3):212–212, 1987. http://dx.doi.org/10.1002/zamm.19870670330.

[18] GNU Binutils. *GNU gprof manual*. https://sourceware.org/binutils/docs/gprof/.

[19] markholland. A collection of parallel implementations of the cholesky decomposition. GitHub Repository, https://github.com/markholland/cholesky/tree/master/OpenCl, Accessed April 2016.

[20] Claudio Brunelli, Eero Aho, and Heikki Berg. Opencl implementation of cholesky matrix decomposition. In *System on Chip (SoC), 2011 International Symposium on*, pages 62–67. IEEE, 2011. http://ieeexplore.ieee.org/document/6089694/.

[21] Azzam Haidar, Ahmad Abdelfatah, Stanimire Tomov, and Jack Dongarra. High-performance cholesky factorization for gpu-only execution. http://dl.acm.org/citation.cfm?id=3038237.

[22] Rashid Mehmood and Jon Crowcroft. Parallel iterative solution method for large sparse linear equation systems. Technical report, University of Cambridge, Computer Laboratory, 2005. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-650.pdf.

[23] Zhangxin Chen, Hui Liu, and Bo Yang. Parallel triangular solvers on gpu. *arXiv preprint arXiv:1606.00541*, 2016. https://arxiv.org/pdf/1606.00541.pdf.

[24] Maxim Naumov. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the gpu. *NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011*, 1, 2011. https://research.nvidia.com/sites/default/files/publications/nvr-2011-001.pdf.

[25] JD Hogg. *A fast triangular solve on GPUs.* Citeseer, 2012. https://pdfs.semanticscholar.org/b87b/2b99d1e12f46fe6aa68641388972870fe15e.pdf.

[26] Li-Wen Chang and W Hwu Wen-mei. A guide for implementing tridiagonal solvers on gpus. In *Numerical Computations with GPUs*, pages 29–44. Springer, 2014.

[27] Girish Sharma, Abhishek Agarwala, and Baidurya Bhattacharya. A fast parallel gauss jordan algorithm for matrix inversion using {CUDA}. *Computers & Structures*, 128:31 – 37, 2013. www.sciencedirect.com/science/article/pii/S0045794913002095.

[28] Wolfgang Engel. *GPU Pro 5: Advanced Rendering Techniques.* AK Peters/CRC Press, 2014.

[29] Ivan Grasso, Petar Radojkovic, Nikola Rajovic, Isaac Gelado, and Alex Ramirez. Energy efficient hpc on embedded socs: Optimization techniques for mali gpu. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 123–132. IEEE, 2014. http://www.petarradojkovic.com/publications/IPDPS-2014_Grasso.pdf.

[30] ARM. *Mali T600 Series GPU OpenCL Developer Guide.* http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0538e/BJEIEBEF.html.

[31] ARM. *Mali OpenCL SDK v1.1.0 Documentation.* http://malideveloper.arm.com/downloads/deved/tutorial/SDK/opencl/.

[32] GNU. *GCC Command Options.* https://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_3.html.

[33] Ubuntu. libopencl documentation. http://manpages.ubuntu.com/manpages/wily/man7/libOpenCL.so.7.html, 2015.

[34] ARM. Mali t604 memory model. http://www.notebookcheck.net/uploads/tx_nbc2/Mali-T604_large.png, 2006.

[35] Intel. *Intel OpenCL Optimization Guide.* https://software.intel.com/sites/landingpage/opencl/optimization-guide/index.htm.

[36] Michal Mrozek and Zbigniew Zdanowicz. Gpu daemon, road to zero cost submission. Intel Corporation, http://www.iwocl.org/wp-content/uploads/iwocl-2016-gpu-daemon.pdf, 2016.

[37] Kirk D Mei W, Hwu W. Video lectures for ece 498al. http://www.nvidia.com/content/cudazone/cudacasts/CUDA%20Programming%20Model.m4v.

[38] Kamran Karimi, Neil G Dickson, and Firas Hamze. A performance comparison of cuda and opencl. *arXiv preprint arXiv:1005.2581*, 2010. https://www.researchgate.net/publication/45917701_A_Performance_Comparison_of_CUDA_and_OpenCL.

[39] NVIDIA. *CUDA Programming Guide.* http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[40] NVIDIA. Tegra mobile processors. http://www.nvidia.com/object/tegra.html, 2017.