GPU Implementation of Gaussian Processes
— Final Report —

Deng, Ying; Khon, Edward; Liang, Yuanruo; Lim, Terence; Ng, Jun Wei; Yin, Lixiaonan
{yd1713, yhk13, yl7813, wsl13, jwn09, ly1413}@doc.ic.ac.uk

Supervisor: Dr. Marc Deisenroth
Course: CO530/533, Imperial College London

16th May, 2014

Contents

1 Introduction

1.1 Project Description

Gaussian process models (henceforth Gaussian Processes) provide a probabilistic, non-parametric framework for inferring posterior distributions over functions from general prior information and observed noisy function values. This, however, comes with a computational burden of $O(N^3)$ for training and $O(N^2)$ for prediction, where N is the size of the training set [1]. Therefore, this method does not lend itself well to problems where N is large — a common occurrence in many modern machine learning or 'big data' problems. There are two routes to address this challenge, and they are (1) using approximations, or (2) the exploitation of modern processors, which we will explore in our project.

Modern-day graphics processing units (GPUs) have been shown to achieve performance improvements of up to two orders of magnitude in various applications by performing massively parallel computations on a large number of cores [2]. In this project, we will investigate whether the parallel processing power of these GPUs can be suitably exploited to scale Gaussian Processes to larger data sets. We also aim to develop a GPU - GP software package for exact GP regression.

1.2 Gaussian Processes

The set-up of a regression problem is as follows: given a set of training data $\mathbf{T} = \{\mathbf{x}_i, y_i\}_{i\in\{1,2,\cdots,n\}}$ consisting of inputs $\mathbf{x}_i \in \mathbb{R}^d$, and observations $y_i \in \mathbb{R}$, what is the best function $f(\cdot)$ that describes the relationship between each input-observation pair?

In Gaussian process regression, we model any finite number of observations y_i as a zero-mean multivariate Gaussian distribution with covariance matrix K, with $(K)_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$. Here $k(\cdot,\cdot)$ is the covariance or kernel function which, can be loosely interpreted as the similarity between two data points. The kernel function can take many forms, as long as it satisfies Mercer's condition (not discussed within the scope of this project [3]). Choosing the right kernel function is integral to the success of a Gaussian process. In our project, we use the Squared Exponential function with an isotropic distance measure (SEiso for short), where the process is depicted in Figure 2. It takes the form:

$$k(\mathbf{x}_i, \mathbf{x}_j) = \sigma_f^2 \exp\left(-\frac{||\mathbf{x}_i - \mathbf{x}_j||^2}{2l^2}\right) + \sigma_n^2 \delta(\mathbf{x}_i, \mathbf{x}_j)$$

Where $\delta(\mathbf{x}_i, \mathbf{x}_j)$ is the Kronecker delta function. σ_n^2 represents the observation variance in the regression problem.

Suppose we have a new input point $\mathbf{x}_*$ and we would like to predict y_*, the target observation associated with it. In Gaussian process regression, we would model our training data and prediction data as follows:

$$(y_1, y_2, \cdots, y_n, y_*)^T \sim \mathcal{N}(\mathbf{0}, \Sigma)$$

where

$$\Sigma = \begin{pmatrix} K & K_*^T \\ K_* & K_{**} \end{pmatrix}$$

$$K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j) \in \mathbb{R}, K_* = (k(\mathbf{x}_*, \mathbf{x}_1), k(\mathbf{x}_*, \mathbf{x}_2), \cdots, k(\mathbf{x}_*, \mathbf{x}_n)) \in \mathbb{R}^{1\times n}, K_{**} = k(\mathbf{x}_*, \mathbf{x}_*) \in \mathbb{R}$$

Given that $\mathbf{y} = y_1, \cdots, y_n$ and the entire covariance matrix are known, we can derive a conditional distribution for y_*

$$y_*|\mathbf{y}, \mathbf{X}, \mathbf{x}_* \sim \mathcal{N}(K_* K^{-1}\mathbf{y}, K_{**} - K_* K^{-1} K_*^T)$$

Our regression estimate for y_* is the mean of this distribution $K_*K^{-1}\mathbf{y}$ and the uncertainty in our prediction is expressed by the variance $K_{**} - K_*K^{-1}K_*^T$.

In addition to choosing an appropriate kernel function, we need to select a set of optimal kernel parameters in order to maximize the performance of the Gaussian process. In the case of the SEiso kernel, the hyperparameters are $\boldsymbol{\theta} = \{\sigma_f, l, \sigma_n\}$. In a Gaussian process, given a set of hyperparameters, we evaluate the effectiveness of the model using the negative marginal log likelihood nll, where

$$nll(\boldsymbol{\theta}; \mathbf{T}) = -\log p(\mathbf{y}|\mathbf{X}, \theta) = \frac{1}{2}\mathbf{y}^T K^{-1}\mathbf{y} + \frac{1}{2}\log|K| + \frac{n}{2}\log 2\pi$$

which is depicted by Figure 1.

The marginal likelihood is a probabilistic measure, which gives us an indication of how likely the observed data is, given a certain set of kernel parameters. The larger the likelihood value, the more effective our model is in explaining the observations. The marginal likelihood obeys Occams' Razor and trades off data fit and model complexity of the function, and hence is a robust metric for evaluation the model. Taking the log (monotonic increasing function) allows us to simplify the expression, and using the negative of this value allows us to combine this with any of the optimizers more naturally, most of which have been designed to find the minimum of functions. This process involves computing the derivative of the marginal likelihood, which is given by the expression

$$\frac{\partial nll}{\partial \theta} = \text{trace}\left(K^{-1}\frac{\partial K}{\partial \theta}\right) + \mathbf{y}^T K^{-1}\frac{\partial K}{\partial \theta}K^{-1}\mathbf{y}$$

for $\theta \in \boldsymbol{\theta}$.

1.2.1 Computational Costs

The computational cost in a Gaussian process regression can be broken down into a few major parts:

- Generating the covariance matrix
  - Computing each element of the $n \times n$ matrix requires a inner product of size d, where d is the dimension of the input space ($\mathbf{x}_i \in \mathbb{R}^d$). This part is $O(n^2 d)$.
- Decomposing the kernel matrix
  - Expressions involving K^{-1} can be computed using the Cholesky decomposition ($K = LL^T$). The derived Cholesky factor as an intermediate from K to K^{-1} can also be used to derive expressions such as $K^{-1}\mathbf{y}$ by solving $K\mathbf{z} = \mathbf{y}$ for $\mathbf{z}$ without having to compute K^{-1} and incurring the cost of the matrix-vector multiplication operation. The Cholesky decomposition, solving of linear systems and computing the inverse using the Cholesky factor are $O(n^3)$.
- Matrix-matrix/matrix-vector multiplication
  - Computing the predictive means for n training data points and m test inputs can be expressed as a multiplication between an $m \times n$ matrix and n length vector. This is $O(mn)$, given $K^{-1}\mathbf{y}$ has been precomputed from a previous routine.
  - Computing the predictive variances for n training data points and m prediction inputs requires m size n inner products with respect to K^{-1} (given K^{-1} has been precomputed from a previous routine). This is $O(mn^2)$.
  - In computing the derivatives of the negative log marginal likelihood, the trace term can be rewritten as the sum of the element-wise product of both matrices, which is an $O(n^2)$ operation instead of an $O(n^3)$ operation.

2 Specification

The software package consists of three parts: a Core Library, a Testing Framework and an Application Programming Interface (API) for portability. The three components work together and are interfaced using MATLAB. The structure is chosen to make the code intuitive from the point of view of a generic regression problem and can be augmented to incorporate additional features. This also provides group members with a basis for designing and working each of their parts in the initial stages, allowing the development of prototypes to be completed swiftly. We kept strictly to our proposed timeline (Refer to Section 4.2) and this is unchanged from the original requirements.

2.1 Core Library

The first part of the software package is the core library. This library provides all the necessary functionalities required for implementing Gaussian Processes using GPUs. In this library, every section of code that is written for the GPU will have a corresponding, suitably optimized CPU version to ensure a consistent and fair way of comparing the implementation of Gaussian Processes on both CPUs and GPUs. Since computational performance is the main focus of the project, this library will be written in C, a low-level language that gives us the ability to control every aspect of the process with much flexibility. The Nvidia CUDA [4] C interface will be used to access GPU functionalities. The library will also incorporate other optimized libraries such as the BLAS (Basic Linear Algebra Subprograms) library [5], LAPACK (Linear Algebra PACKage) [6] and MAGMA (Matrix Algebra on GPU and Multicore Architectures) [7].

2.2 Testing Framework

The testing framework will provide an environment in which testing and analysis of our implementation can take place. For fair comparisons of the results, both CPU and GPU implementations will use the same datasets and parameters. Our datasets will include both synthetically-generated data and real-world data. The use of synthetic data enables us to test our implementations over a large number of different dimensions and dataset sizes in a controlled environment, allowing us to demonstrate the performance implications of using GPUs over CPUs (or vice-versa). Data from real world problems would put into perspective the potential practical benefits of using our implementations in reality. The main benchmark for evaluating our implementation's performance will be computational time, subject to a given threshold for numerical accuracy.

2.3 MATLAB Toolkit, API and Documentation

While C is a powerful and efficient language, it may not provide a friendly environment for scientific computation as compared to other platforms such as MATLAB, which offer syntactical convenience and intuitive data types for mathematical analysis. For our library to be useful, it has to be portable to at least one of these environments. We aim to create a MATLAB toolkit. The choice of MATLAB is due to the existence of the GPML (Gaussian Processes for Machine Learning) toolkit for MATLAB [8], a comprehensive and well-documented package, which we will use as a benchmark for setting the scope and standard of our own implementation. Along with the MATLAB toolkit, we will provide an API and documentation for our library to allow other users to use our library in any other platform of their choice.

3 Design

3.1 Libraries Used

3.1.1 Basic Linear Algebra Subprograms (BLAS)

The BLAS library consists of a set of low-level subroutines that perform common linear algebra operations such as copying, inner products, matrix-vector/matrix-matrix multiplications. Although it originated as a set of FORTRAN subroutines, the BLAS interface has become the standard API for linear algebra computation across many other libraries and is a mainstay in high performance computing. Using BLAS gives our library portability since many implementations of BLAS exist (OpenBLAS and Intel's proprietary Math Kernel Library are two implementations that are known for superior performance), and the user is free to substitute this with any implementation optimised for the hardware he or she is using our library on.

3.1.2 Linear Algebra PACKage (LAPACK)

Built on the BLAS interface, LAPACK is a library consisting of linear algebra routines to solve systems of linear equations, least-squares equations, eigenvalue problems and singular value problems. Like BLAS, the LAPACK interface has become the standard for linear algebra routines in numerical computing. Including LAPACK in our library allows us to bring the same benefits to users as those mentioned above (BLAS section).

3.1.3 CUDA BLAS (cuBLAS)

cuBLAS is an implementation of the BLAS interface in the CUDA programming environment for Nvidia devices. In addition to being a well optimized library by the makers of the hardware themselves, the use of this library provides us with a natural translation for parts of our code from the CPU (BLAS) to the GPU (cuBLAS).

3.1.4 CUDA Linear Algebra (CULA)

CULA to cuBLAS is what BLAS is to LAPACK. CULA adopts the LAPACK interface for linear algebra routines for execution on the GPU. Like cuBLAS, it is developed by Nvidia and well optimized. It has been shown to have superior performance over CPU implementations of LAPACK.

3.1.5 Limited Memory Broyden-Fletcher-Goldfarb-Shanno (libLBFGS)

The BFGS algorithm is one of the many Quasi-Newton methods of doing numerical optimisation [9]. We have chosen to incorporate the libLBFGS which is an implementation of the limited memory BFGS algorithm [10] in our project as an example of how our library can be combined with a numerical optimiser. The libLBFGS library is written in C and allows us easily with the rest of the library which is also written in C. The user is not bound by this choice, and having compiled/wrapped the library into another programming environment (e.g, MATLAB/Python) can use any optimiser available in that environment.

3.2 Negative Log Marginal Likelihood and Derivative Marginal Log Likelihood Execution Path

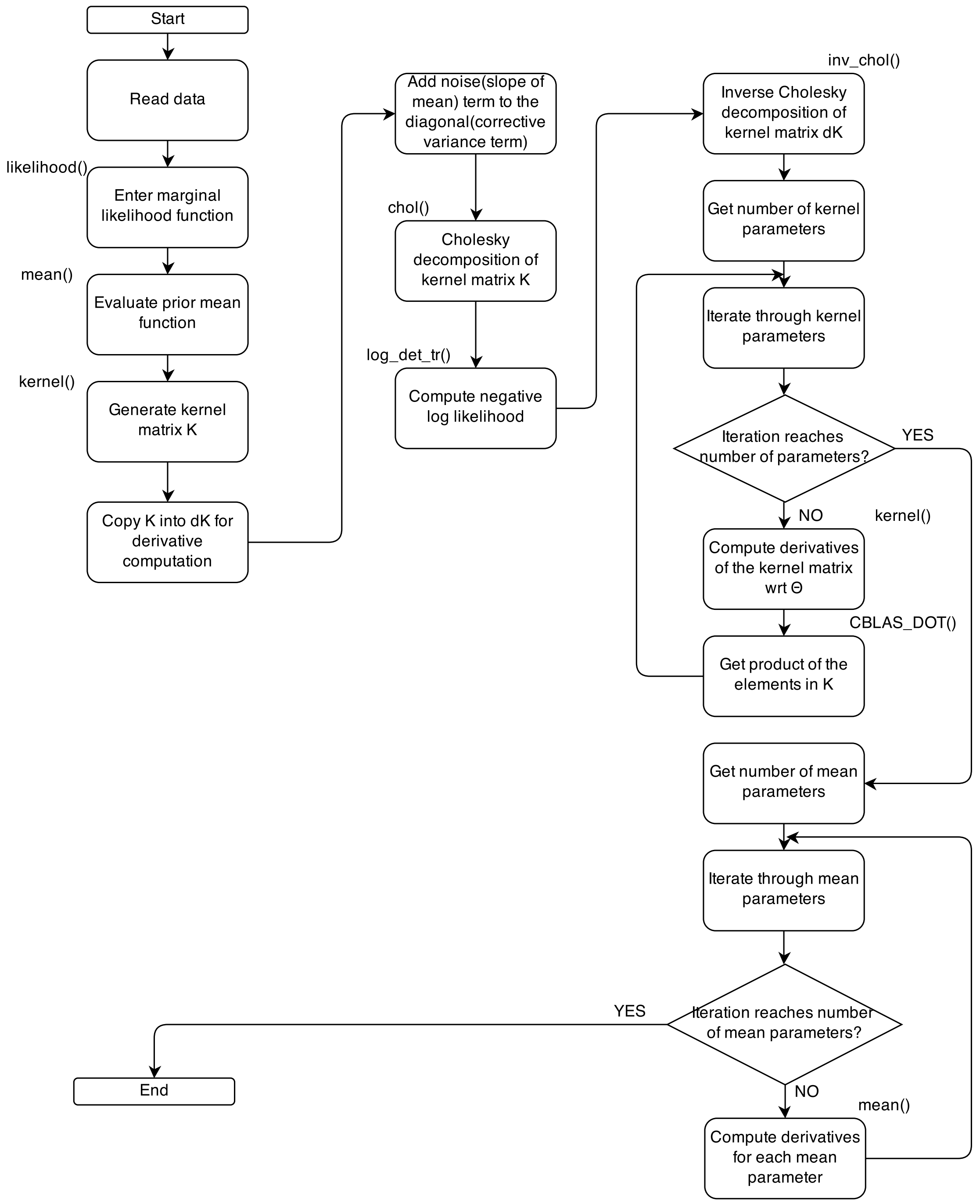

Start
Read data
likelihood()
Enter marginal likelihood function
mean()
Evaluate prior mean function
kernel()
Generate kernel matrix K
Copy K into dK for derivative computation
Add noise(slope of mean) term to the diagonal(corrective variance term)
chol()
Cholesky decomposition of kernel matrix K
log_det_tr()
Compute negative log likelihood
inv_chol()
Inverse Cholesky decomposition of kernel matrix dK
Get number of kernel parameters
Iterate through kernel parameters
Iteration reaches number of parameters?
YES
NO
kernel()
Compute derivatives of the kernel matrix wrt Θ
CBLAS_DOT()
Get product of the elements in K
Get number of mean parameters
Iterate through mean parameters
Iteration reaches number of mean parameters?
YES
NO
mean()
Compute derivatives for each mean parameter
End

Figure 1: Execution path of negative log marginal likelihood and derivative marginal log likelihood

3.3 Kernel Matrix Execution Path

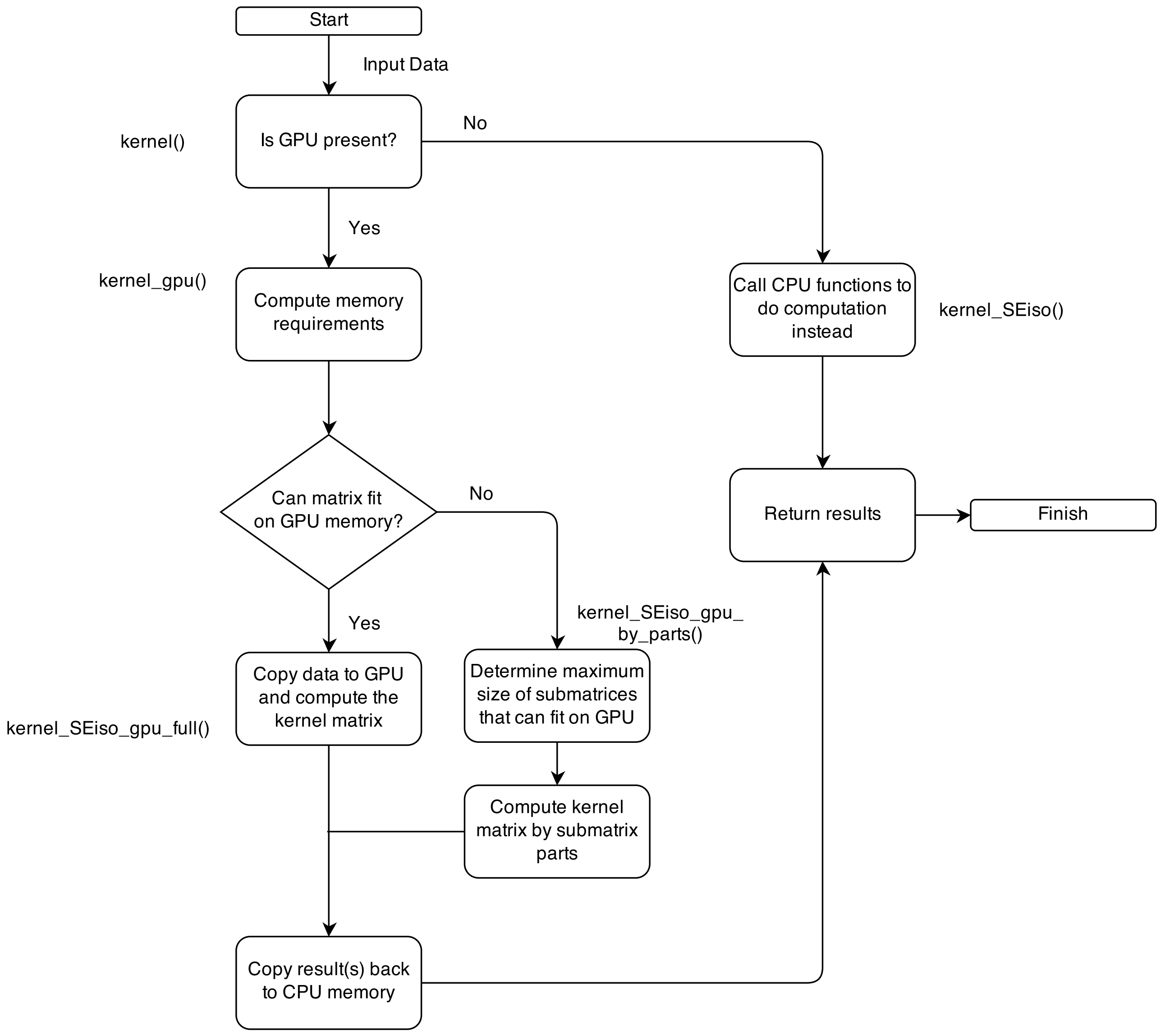

Start
Input Data
kernel()
Is GPU present?
No
Yes
kernel_gpu()
Compute memory requirements
Call CPU functions to do computation instead
kernel_SEiso()
Can matrix fit on GPU memory?
No
Return results
Finish
Yes
kernel_SEiso_gpu_by_parts()
Copy data to GPU and compute the kernel matrix
kernel_SEiso_gpu_full()
Determine maximum size of submatrices that can fit on GPU
Compute kernel matrix by submatrix parts
Copy result(s) back to CPU memory

Figure 2: Execution path of the kernel matrix

3.4 Layer Design Model

Our software package works with a layered design architecture as shown in Figure 1. It is similar to the Open Systems Interconnection (OSI) model [11], where each layer has specific responsibilities and knows nothing about the operation of the other layers. For instance, users interact with the top layer (MATLAB), where they call our library package within MATLAB, passing in the necessary parameters. These parameters are then handled by the MEX layer, which deals with the checking of inputs to ensure that the correct parameters are given. Within MEX, the C code is called, similarly passing the necessary parameters, and finally, the CUDA layer manages the GPU implementation. Eventually, the information is then passed back up the layers back to MATLAB to be returned to the user.

The advantage of having such a layered design architecture is that it breaks down our problem into different and smaller parts to be worked on by different groups, which reduces complexity. Moreover, members working on the MATLAB and MEX layer can work independently from those working on the lower level GPU implementation. Changes to code can also be made within each layer without breaking code of other layer provided that the interface is adhered to, and this facilities modular engineering.

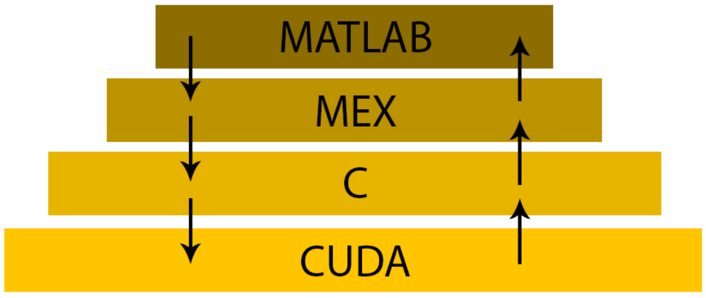


Figure 3: Layering design architecture of our software

3.5 Switching between CPU and GPU Computation

In order to achieve general-purpose computing on graphics processing units (GPGPU), it was decided to use the Complete Unified Device Architecture (CUDA) from Nvidia Corporation as a parallel computing platform. The reasons were twofold:

- Our allocated lab machine graphic01 comes a GeForce GTX 570 CUDA-enabled graphics card and CUDA library installed.
- It is known that CUDA could be integrated and compiled with both C and MATLAB. Using CUDA C programming, memory manipulation would be in the programmer's control, enabling the writing of memory-efficient code. This is a significant advantage when writing code for memory-intensive applications working with large data sets.

3.6 Compute Capability

The compute capability of a GPU is defined by a major revision number and minor revision number. These represents the core architecture of the GPU. For instance, the GeForce GTX 570 used in conjunction with our project is of compute capability 2.0, which indicates it is of the Fermi architecture [12].

The architecture of the GPU is significant because different architectures and revision numbers indicate different capabilities and features. Therefore, the success of kernel launches are ultimately tied with the hardware architecture, i.e., one cannot assume hardware abstraction when writing GPU code.

As the GPU architecture evolves, the design changes can affect backwards compatibility — code written for a specific architecture of GPUs may not work on computers with an older GPU architecture. For instance, the maximum number of threads per block is limited to 512 on a 1.x architecture device, whereas it is 1024 on a 2.x architecture device. Or, more radically, double-precision floating point numbers are not supported for devices with compute capability 1.2 and below.

In order to guarantee 2.x compatibility, we used the compilation flag `-arch=sm_20` with the `nvcc` compiler. A simplified explanation is that this allows the compiler to generate PTX code (pseudo-assembly bytecode) and utilise just-in-time compilation at program runtime, in order to select the most appropriate binary code to load and execute.

The CUDA code written for this project guarantees compatibility for the Fermi architecture v2.0 and upwards.

3.7 Column-Major Order Storage

MATLAB uses column-major storage for matrices, as part of their mxArray data structure. This means that a two dimensional matrix is stored column-by-column in a one dimensional array in memory, and it is accessed with only one index. We use the same convention as MATLAB for accessing matrices within our program. Following the convention of MATLAB is advantageous because we avoid the conversion between row-major and column-major and save unnecessary computation.

In column-major storage, the row i and column j indices of a matrix are related to the array index via the formula:

$$index = i + N \times j \tag{1}$$

where N is the stride of the matrix.

3.8 Symmetric Matrices

The computation of symmetric matrices implies that only half of the matrix need be computed. In our program, we can set a flag `SYMM` to indicate to the program that it needs only to compute half the matrix and copy its result to its symmetric counterpart element.

For a symmetric matrix, the identity

$$X[i + N \times j] \equiv X[j + N \times i] \tag{2}$$

is used to access the corresponding symmetric element for the array X.

4 Methodology

4.1 Development Strategy and Division of Work

Our group adopted the Agile framework, specifically the Scrum model [13], for our development strategy. As a group, we came to a consensus to operate on a centralized and flexible model. Jun Wei acted as the Scrum master/group coordinator, who facilitated our weekly updates and meetings and also challenged the team to improve.

The overall task was divided into three self-contained mini projects, where we gathered requirements, as well as implemented design, programming and testing. In summary, there were the self-contained teams within our project. They are namely, the core library, testing framework and the front end API. which are discussed under Section 6.

4.2 Implementing Scrum as our Software Development Technique

We adopted the scrum model for our software development technique, because scrum is an iterative model, which can enable our team to be flexible to changing requirements and challenges that cannot be easily addressed in a planned manner. For instance, we anticipated working close with our customer Dr. Marc Deisenroth, where we are constantly prepared for feedback and changes as we ship the final product.

Also, due to the fact that our group is relative small with six members, it is easy to manage using the Scrum model. The Scrum model has been tried and tested, and have been put to use by many successful organizations such as Google, Microsoft, Facebook, Adobe and Bank of America [14]. We feel that we will work well with the flexibility provided by this model. Also, this model will provide us with good practice and experience for software engineering in the real world.

One of the key practices of the Scrum model is to have short iterations called Sprints. We met up on a weekly basis every Tuesday afternoon for sprint meetings with our supervisor, Dr. Marc Deisenroth, which lasted approximately an hour. In the meetings, we had the opportunity to update one another on our progress, clarify doubts, discuss the possibilities of changing requirements, or request for assistance. We also met up as a group after our meeting with Dr. Marc Deisenroth, where we checked our progress against our timeline just to ensure that we are on track. The timeline is as follows:

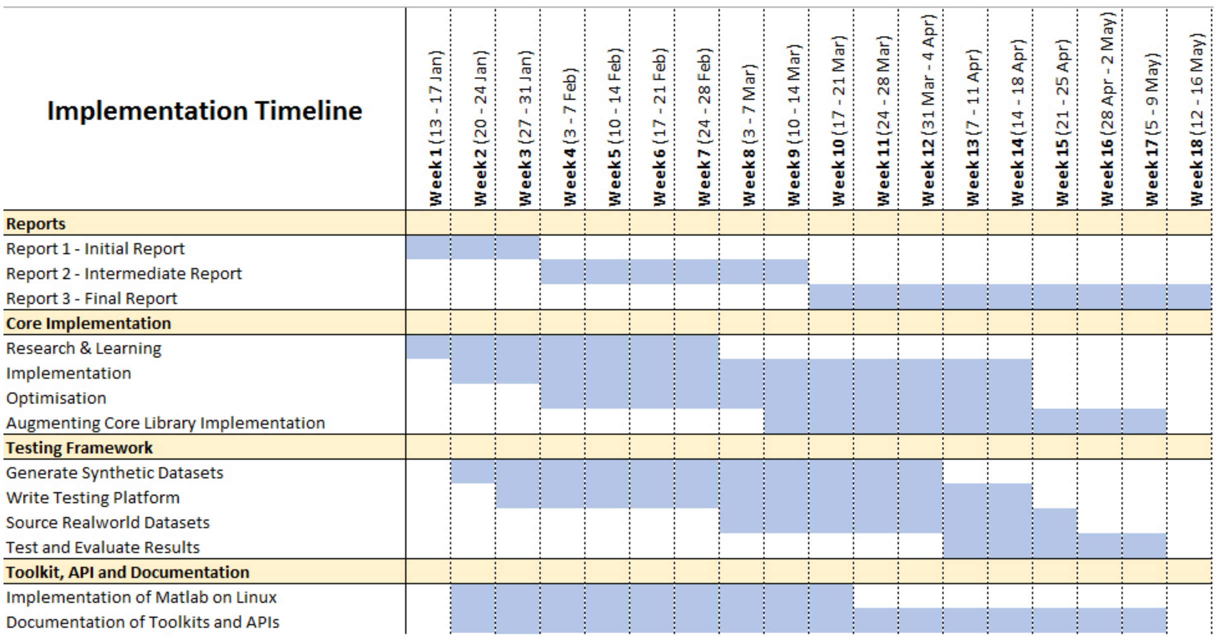


Figure 4: Gantt chart of the project

We also demonstrated our software at the end of each iteration to Dr. Marc Deisenroth, who also doubled up as the customer in a real software engineering project to ensure that our project is suitably adapted to the client's expectations. On top of this, we also met up for daily scrum meetings in between our common classes, where we briefly updated one another on issues that have surfaced. We also kept a product backlog which list the functionalities or use cases desired in the product. This backlog was shared via Dropbox, and easily updated by members.

4.3 Version Control

A technical problem faced was the need to work on similar code at the same time, and to manage the repository such that we have one working product available at any point in time. To solve this problem, we had to choose a form of Version Control, one that the group members were familiar with. Clearly between the choices of Git and Mercurial, we chose the former. The choice of Git was a natural one — its branching and merging functionalities allowed us to appropriately develop working prototypes and recombine further work on different parts as progress was made. Furthermore, the projects repository was hosted on the Imperial College Department of Computing's GitLab server, and all members were familiar with it. Also, only Jun Wei, our coordinator, had full access to the main master branch which contained a working product. Changes by members were pushed to separate branches, which are tested for errors and scrutinized by Jun Wei before changes were merged into the master branch.

4.4 Problems Faced

4.4.1 Linking of Libraries with MATLAB

A major problem we had was the linking of existing libraries using available libraries such as FORTRAN, Linear Algebra Package (LAPACK), Basic Linear Algebra Subprograms (BLAS), cuBLAS (Nvidia CUDA Basic Linear Algebra Subprograms), Broyden-Feltcher-Goldfarb-Shanno (BFGS) and MAGMA with MATLAB. This was because MATLAB came with its own preinstalled libraries, and we had to find ways to compile our C code into a MATLAB executable using our own version of these libraries.

4.4.2 Threads, Blocks, Memory and Race Conditions

When launching a kernel to do parallel computation, we had to ensure the following:

- The number of threads launched must be sufficient to justify the overhead of copying data and program instructions to the GPU device. Launching insufficient threads than the problem size leads to incomplete computation, while launching too many leads to insufficient occupancy.
- The number of threads within each block must be monitored. This is important because the GPU issues 32 instructions at a time (called warps) and any unused threads are wasted.
- Synchronisation and atomicity must be used across threads. These are necessary when utilising shared memory in order to avoid garbage reads.

4.4.3 Memory Management

1. GPU Memory Capacity and Limitations

   For a large data set, the kernel matrix K will exceed GPU memory. For instance, the GTX 570 GPU has 1279MBytes of global memory, which allows approximately 31975000 32-bit `float` type elements or 159875000 64-bit `double` type elements. Furthermore, the input matrices to compute K (we shall term these X_1 and X_2) must also be accommodated on the device memory.

   To circumvent the problem, we derived a framework to compute the kernel matrix section by section, one square submatrix at a time. This approach allows us to safely and effectively compute the kernel matrix whilst computing with the GPU at maximum efficiency.

Figure 5 presents the computation sequence of the kernel matrix. As a symmetric matrix, the elements located in the top-right triangle and corresponding elements located in the bottom-left triangle, excluding the diagonal, will have the same values, so the required part of the matrix to hold full information is either triangle plus the diagonal elements. In this situation, we select the bottom-left triangle to compute for ease of programming. The order of computation is to compute top to down, left to right, so if the result matrix is split into 9 parts from K_{11} to K_{33}, where the first digit shows vertical coordinate, and the second digit shows horizontal coordinate, the parts being computed are: K_{11}, K_{21}, K_{22}, K_{31}, K_{32}, K_{33}, in stated order.

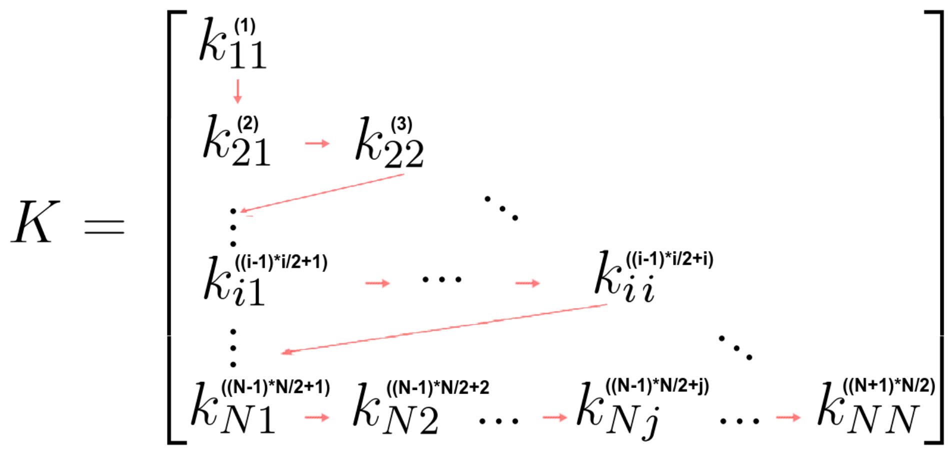


Figure 5: Computing the kernel matrix, one square submatrix(K_{ij}) at a time

2. Computing Submatrices

In order to determine the size of the kernel submatrix that may be allocated on GPU memory, we may query the GPU for its available memory M. When input matrices X_1 and X_2 are both too large, the optimal computation speed can be obtained when the input submatrices are of the same length, thus each computation produces a squared part of the result matrix, which is the maximum entries being calculated with the same memory size. M is related to the GPU memory limitation of

$$M \geq r \times c \times 2 + r^2 \tag{3}$$

where c and r correspond to input matrix's dimension and stride (the number of rows of the matrix) respectively. Also considering that one of the input matrices might be too small that it will not reach the input limit, the other input matrix will have a looser limit of:

$$M = r_{X_1} \times c + r_{X_2} \times c + r_{X_1} \times r_{X_2} \tag{4}$$

where r_{X_1} and r_{X_2} correspond to the stride of the input matrix X_1 and X_2 respectively. By dividing the computation into blocks, it might be possible that the last few rows (and/or columns) of the result matrix are smaller than the calculated limit. In this case they are computed in the same way of the other submatrices but with a smaller stride parameter (less rows of input matrix).

As a result of dividing the computation of the resultant matrix K into submatrices, the originally symmetric result matrix will only have symmetric submatrices where they reside along the diagonal, but will be asymmetric otherwise. Where possible, the submatrix computation will ignore upper-triangular elements encountered when computing each square submatrix along the diagonal.

After the GPU computes the lower triangular result matrix, the elements are simply copied to their symmetric counterpart in order to obtain the full matrix K.

The algorithm we implemented will utilise the GPU to compute the lower triangular result matrix K, so that half the computation time can be stored; then the option of `FILL` and `PACKED`, which are mutually exclusive, will result in three ways of copying the submatrix back to the result matrix: (a) copy only the computed values back to their corresponding positions; or (b) copy the computed values back to their corresponding positions as well as the symmetric counterpart. This is done via flag checking after the computations.

The basic structure of the algorithm is as follows:

```
1     Query available memory from GPU
2     Compute size limit for input matrices according to available memory
3     Decide whether data set size (length) of input X1 is too large
4     If input X1 does not exceed size limit
5          Compute size limit for input X2 according to total size of input X1
6     Else
7          If input X2 does not exceed size limit
8              Recompute size limit for input X1 according to size of input X2
9     Split inputs X1 and X2 according to their size limit into submatrices
10    Loop when there exists submatrices not computed
12         If the sub matrix at the bottom-left triangle of the kernel
13             Compute results of the sub matrix
14             Combine submatrix into the kernel matrix
15         Else
16                 Skip the computation of this submatrix
17         Else
18             Compute results of the sub matrix
19             Combine submatrix into the kernel matrix
```

3. Determining Parameters for Kernel Launch

   The compulsory parameters required when launching a GPU kernel computation are the number of threads `N_THRDS` and the number of blocks `N_BLKS`, from which a third value, the number of threads per block, can be determined `THRDS_PER_BLK`, where

   `THRDS_PER_BLK = N_THRDS / N_BLKS`

   On a GPU, a warp of 32 threads are launched and handled by each multiprocessor — which means that in order to optimise for occupancy (number of active threads), the number of threads per block should be a multiple of 32, and yet not exceed the device limit (1024 in the case of the `graphics01` lab machine).

   The following equations are used to determine kernel launch parameters for an arbitrary problem of size N and dimensionality of 1 (i.e. vectors and column-major upper triangular matrices):

   `THRDS_PER_BLK = 1024`

   `NUM_THRDS = N`

   `NUM_BLKS = N / 1024 + 1`

   Using CUDA's built-in indexing code, the unique ID for each thread can be determined with

```
int i = threadIdx.x + blockIdx.x * blockDim.x;
int j = threadIdx.y + blockIdx.y * blockDim.y
```

For problems of size $M * N$ and dimensionality of 2:

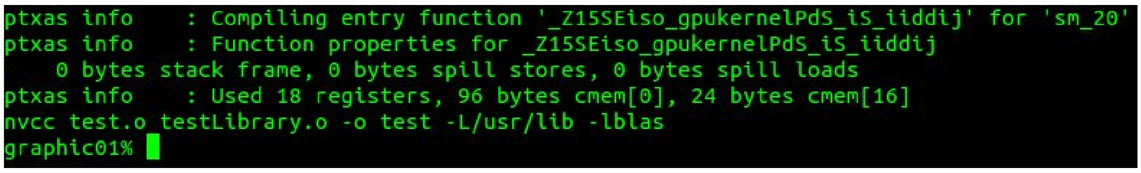
```
THRDS_PER_BLK = (32,32)
NUM_THRDS = (M,N)
NUM_BLKS = (M/32 + 1, N/32 + 1)
```

Using CUDA's built-in indexing code, the unique ID for each thread can be determined with

```
int i = threadIdx.x + blockIdx.x * blockDim;
```

4. Improving parameter estimation with CUDA occupancy calculator

   The CUDA Occupancy Calculator is a tool provided by Nvidia to [15] estimate and improve the kernel launchtime parameters. Using the `-ptxas` compiler flag in NVCC, we can obtain the following information about the kernel launch as shown in Figure 6.

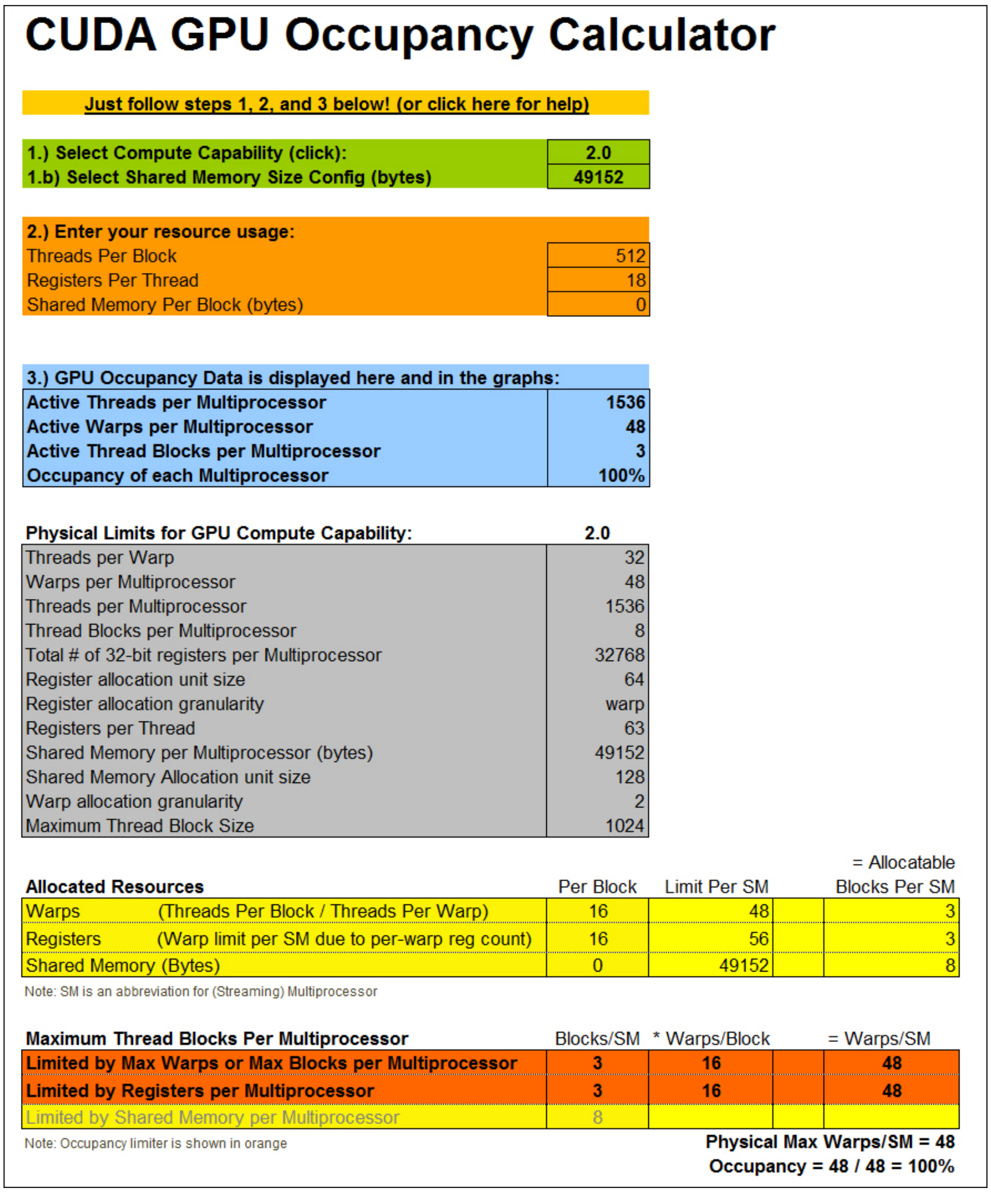


```
ptxas info    : Compiling entry function '_Z15SEiso_gpukernelPdS_iS_iiddij' for 'sm_20'
ptxas info    : Function properties for _Z15SEiso_gpukernelPdS_iS_iiddij
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 18 registers, 96 bytes cmem[0], 24 bytes cmem[16]
nvcc test.o testLibrary.o -o test -L/usr/lib -lblas
graphic01% 
```

Figure 6: Information obtained from the NVCC compiler can be used to improve GPU occupancy

   The information from Figure 6 (number of registers, shared memory) can be used as input to the CUDA Occupancy Calculator, as shown in Figure 7. The results are in Figures 8 and 9.

   From Figures 8 and 9, we see that our program maximises GPU occupancy without causing register spillage. This indicates that our kernel launches are at near-maximum efficiency.



CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	2.0
1.b) Select Shared Memory Size Config (bytes)	49152

2.) Enter your resource usage:	
Threads Per Block	512
Registers Per Thread	18
Shared Memory Per Block (bytes)	0

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	1536
Active Warps per Multiprocessor	48
Active Thread Blocks per Multiprocessor	3
Occupancy of each Multiprocessor	100%

Physical Limits for GPU Compute Capability:	2.0
Threads per Warp	32
Warps per Multiprocessor	48
Threads per Multiprocessor	1536
Thread Blocks per Multiprocessor	8
Total # of 32-bit registers per Multiprocessor	32768
Register allocation unit size	64
Register allocation granularity	warp
Registers per Thread	63
Shared Memory per Multiprocessor (bytes)	49152
Shared Memory Allocation unit size	128
Warp allocation granularity	2
Maximum Thread Block Size	1024

Allocated Resources		Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps	(Threads Per Block / Threads Per Warp)	16	48	3
Registers	(Warp limit per SM due to per-warp reg count)	16	56	3
Shared Memory (Bytes)		0	49152	8

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	3	16	48
Limited by Registers per Multiprocessor	3	16	48
Limited by Shared Memory per Multiprocessor	8		

Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 48
Occupancy = 48 / 48 = 100%

Figure 7: The CUDA Occupancy Calculator

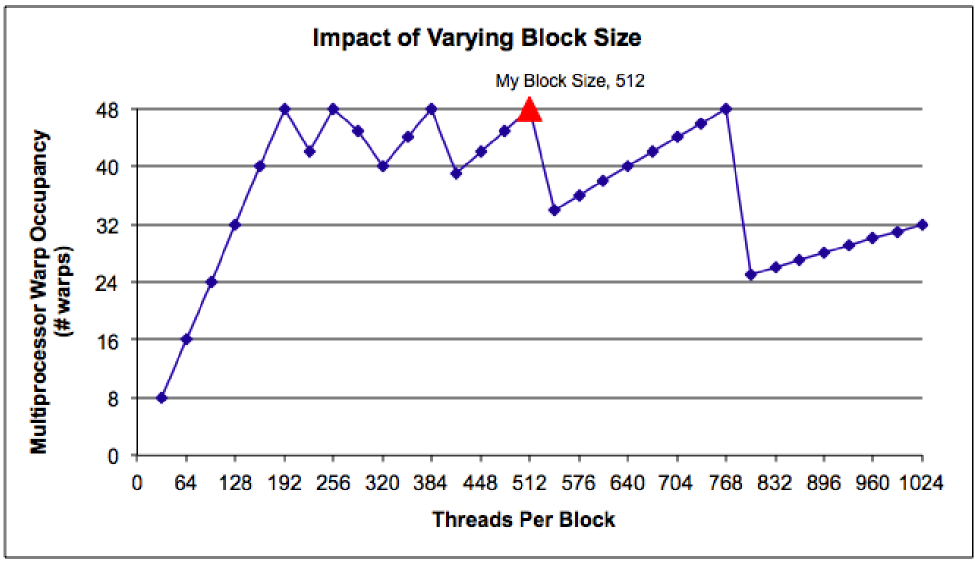


Figure 8: Estimated occupancy as a function of threads per block

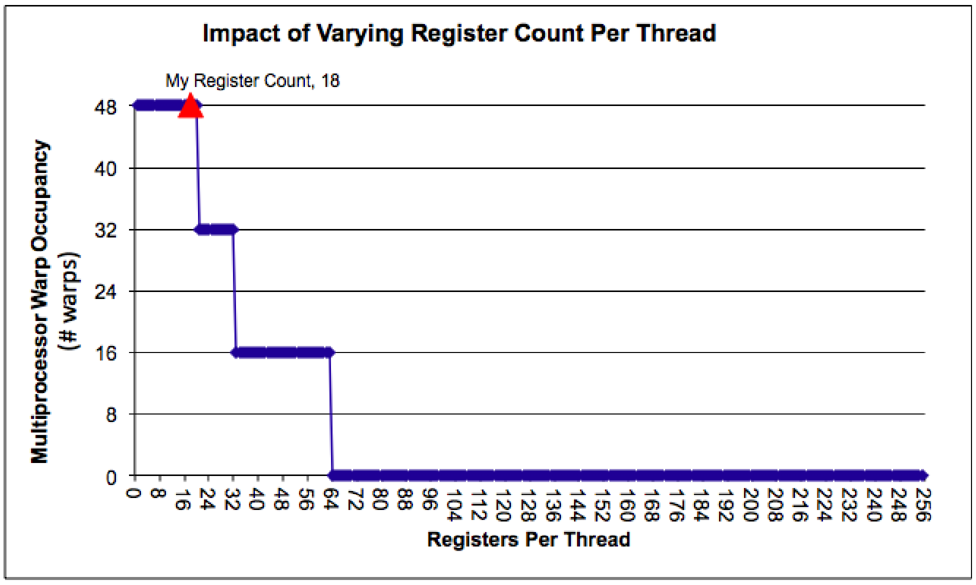


Figure 9: Estimated occupancy as a function of registers per thread

4.4.4 Performance Optimisation Strategies

1. Maximising Utilisation

   The application should be structured in a manner such that iterative and repetitive computations are parallelised as much as possible, and that hardware bottlenecks are minimised.

   At the device level, the GPU should maximise parallel computation between multiprocessors of a device. We are currently looking into improving this by using more advanced techniques such as asynchronous concurrent execution, which is a technique to enable kernel computation and memory transfers concurrently.

2. Maximising Memory Throughput

   Maximising memory throughput means to reduce small bandwidth transfers between the GPU and the CPU. By maximising memory throughput, we can reduce the overhead on each call to the operating system.

   Where possible, we use paged (pinned) memory for device computation. This would mean that the GPU would be able to bypass the host when retrieving data (DMA access), so that read/writes by the GPU would be faster because there would be no need for the CPU to copy the output to buffer. This results in an effective bandwidth increase of $\times 2$.

3. Maximising Instruction Throughput

   Instruction throughput is defined as the number of instructions that can be executed in a unit of time. Maximising instruction throughput can be accomplished by having few divergent paths of execution in the kernel execution (usually only an `if()` statement guarding out-of-bounds array access). This is because different execution paths can cause threads of the same warp (job batch) to diverge. The different paths would have to be serialised, increasing the number of executions needed.

   Instruction throughput is also maximised by trading precision for speed (i.e., using `float` instead of `double` where possible). By using `float` format instead of `double`, the size of a matrix copied into the GPU memory can be 4 times (twice the size of each dimension of the result matrix) because the `float` data requires half of memory compared to `double`. For this reason, the number of threads being launched each time can be 4 times larger, hence reducing the computation time for the same size matrix to be 1/4 of the double precision.

   However, being a scientific computation project, precision is left to the user of the library to decide via the macro `FLOAT` which is determined at install time by the user. Using `double` instead of `float` in GPU device code on a Fermi architecture GPU results in at least $\times 2$ and as much as $\times 8$ slower performance, due to increased computational cost and an underlying hardware implementation bias towards single-precision floating point numbers.

   Atomic addition of double precision numbers is not supported natively in CUDA and must be externally implemented by the programmer, which will further compound the time taken.

4.5 Testing Methodology

Testing is an important aspect of software development, as enures that the code written is doing what is expected, and seeks to identify errors and bugs in a program. The subject of testing will be the GPU Gaussian Processes toolbox for MATLAB.

Features to be tested: We test our program against 5 main features, which are partly drawn from the characteristics under the ISO 9126 Standard "Software Product Evaluation – Quality Characteristics and Guidelines" [16]. These are:

1. Functionality: Do valid inputs result in valid outputs? Are invalid inputs caught and error messages thrown?
2. Accuracy: Are the outputs correct?
3. Efficiency: How fast does the program run? How much resources does it take? Where are the bottlenecks?
4. Usability: Is the program user friendly?
5. Portability: Can the program be ported to various platforms easily? Is it easily installable?

For the most part, we take a largely scripted testing approach. Our program, being of a scientific nature, has well-defined behaviour and limited variability in use cases, which lends itself to such a testing approach where test cases are designed a priori.

4.5.1 Functionality and Accuracy

In terms of functionality and accuracy, a partition testing approach is used. This means that the universe of possible inputs to our program are partitioned into equivalence classes, and a test case is created for each equivalence class using a representative input. For some equivalence classes, we also add test cases for boundary values between classes (See Appendix B.1).

The choice of equivalence classes is best described as white-box. For one, from the perspective of an end-user, there should be no class difference between a very large (e.g, 50×50000 training inputs) and a small (e.g, 2×100 training inputs) real-numbered training dataset. A black-box approach would hence only require one test case for the domain of such valid datasets. On the other hand, we have decided to adopt a 'white-box' testing approach precisely because we know that the size of the dataset has practical implications on the algorithm used (e.g, matrix subpartitioning). In addition to the size of the input, the numerical algorithm is sensitive to several other input types as well. As a result, we divide the input space more finely for more robust testing.

4.5.2 Efficiency

Efficiency is of a slightly different nature compared to functionality and accuracy. Here, we are more interested in profiling the program — finding out the bottlenecks in the algorithm in order to optimize it. This is particularly important to the project since it is directly related to our thesis, which is to find out at which point the performance of a GPU implementation might exceed that of a CPU implementation of the Gaussian process methodology. To do this, we benchmark our program against a CPU implementation for various sizes of data. We will run the benchmark tests based on the same conditions, so as to produce a control range of results that can be compared against.

4.5.3 Usability and Portability

In terms of usability and portability, we classify them as a different type of testing, where the users are largely involved. These are 'tested' by consultation with the client. We performed user testing, where we conducted sessions and involve participants to perform a certain task, such as passing in a set of training inputs and outputs and performed an inference test to obtain the negative log marginal likelihood and the partial derivatives with respect to the hyperparameters. We also provide the user with a set of documentation and instructions for him/her to perform the task. Subsequently, we noted any feedback from the user, in terms of ease of use, as well as points for improvement, and implemented them towards the shipping of the final product.

4.6 Continuous Integration and System Level Testing

In terms of our testing philosophy, we modeled our tests using a form of continuous integration. The test cases for functionality, accuracy and efficiency together have been consolidated into an automated test suite that checks the output of each test case against its expected output. We adopted a regression testing approach in that the developers may only submit their code (merge request) if and only if their updated program passes all the tests that have previously been passed. This ensures that changes do not break previously-functioning code. By identifying integration bugs at an early stage, we save time and effort on debugging over the lifespan of the project. This also ensures that we have an almost 100% availability of a most-updated build at any point in time for demonstration purposes.

Thus far, the testing described falls into the unit testing category. However, on a more 'meta' level, although the scientific nature of our program implies that there is little in the way of different use cases, we do our best to make sure that our program is as robust as possible on the system level. As such, we also performed system testing. This requires a list of objectives for the project, which we will compare against when testing the system. Through these series of tests, we aim to find any misalignment between our objectives and specifications of the system. System-level robustness is also rigorously tested using software fault injection, to make sure that our program handles environmentally-induced situations such as CPU memory overloading, Matlab errors etc. in a predictable manner, terminating gracefully if appropriate.

4.6.1 Evaluation Criteria

With respect to functionality and accuracy, the invalid cases should have an informative error message printed, while the valid cases should return results that are numerically similar to those of the GPML toolbox within an appropriate error bound. That is, given the same set of inputs, the outputs generated by the GPML toolbox and those from our implementation should not differ by more than a given percentage (0.1%). A test log will be generated for each test run, which will contain an entry for every test case, indicating whether it has passed or failed, with a digest of the output/error messages recorded.

While there is no distinct pass/fail criteria to efficiency, we should expect our GPU implementation to at least be more efficient than the CPU implementation for some range of inputs. We currently expect that the CPU implementation would be superior to the GPU implementation for small data sets and past a certain level, the GPU would overtake the performance of the CPU. For usability and portability, this would once again be subject to the client's comments, which will be documented.

We aim to achieve 100% statements and branch coverage using our automated code coverage tool `gcov/lcov`.

5 Group Work

We divided the project into three closely-related ‘mini projects’. This consisted of the Core Library, the Testing Framework and the API. Each member’s focus was then spread across these mini projects, as seen in the table below.

Core Library	**Testing Framework**	**API**
Jun Wei	Edward	Terence
Yuan Ruo	Lixiaonan	
Deng Ying		

The Core Library team led by Jun Wei worked on the implementation of the Gaussian Process library. A great deal of the implementation was modularized into GPU functions (kernels), which were the domain of Yuanruo and Deng Ying. The upshot of this modularization was that there was some tricky interleaving of CPU and GPU code. As such, Jun Wei oversaw the overall integration of the various components of the algorithm.

The Testing Framework was led by Edward, who also served as the lead tester of the team. Following a test-driven development style, following an initial brainstorming of project requirements by the full group, the Testing Framework team then translated those requirements into a test suite, and began an iterative process of expanding the suite to cover incremental paths resulting from changes in the Core Library and API implementations. Lixiaonan was in charge of preparing the datasets used in testing, both by organizing real-life datasets and synthesizing them in a Monte Carlo fashion.

Finally, the API was led by Terence. This required the integration of the C-based core library with the MATLAB front-end, using the MEX framework. The need to make our Gaussian Process toolbox portable across operating systems added a dimension of complexity to this task. Terence also had to work very closely with both other teams, since the task of marshalling user input from the MATLAB front-end into the core library and handling relevant exceptions required understanding of both user cases as well as the implementation of the core library.

Refer to Appendix D for detailed work breakdown, and priority of tasks.

6 Final Product

6.1 Product Description

Our final product consists of a software package that consists of a core library, a testing framework and a documented API, interfaced with MATLAB. The use of our software package is similar to the GPML toolbox as mentioned earlier.

Users may load their training inputs, test inputs into MATLAB using a 2-dimensional array respectively, together with the mean, covariance, and likelihood parameters as a struct, and call our library using `GP()`.

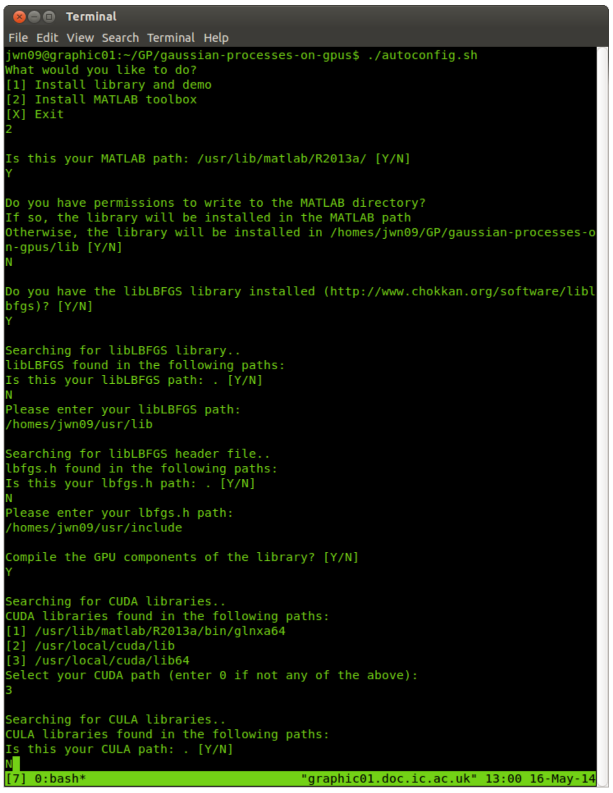
Users may choose to use between two covarience methods, namely the Isotropic Distance Measure or the Automatic Relevance Determination. Users may also choose to use between three prior mean methods, namely Zero Mean, One Mean or Linear Mean.

A wiki page is also available for users as a platform to learn how to use our library, as well as to learn more about our project. Users who are familiar in the field may also share and contribute their knowledge. Refer to 6.3 for full details.

6.2 Using our Software in MATLAB

Our software requires some configuration to be set, and this can be easily done with a simple script as shown in Figure 10.

6.2.1 Setting up the Configuration Files

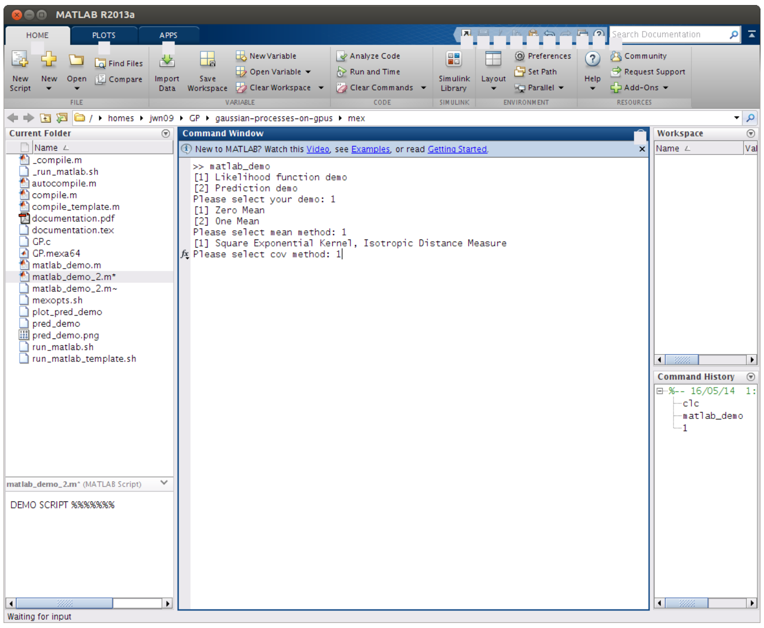


```
jwn09@graphic01:~/GP/gaussian-processes-on-gpus$ ./autoconfig.sh
What would you like to do?
[1] Install library and demo
[2] Install MATLAB toolbox
[X] Exit
2

Is this your MATLAB path: /usr/lib/matlab/R2013a/ [Y/N]
Y

Do you have permissions to write to the MATLAB directory?
If so, the library will be installed in the MATLAB path
Otherwise, the library will be installed in /homes/jwn09/GP/gaussian-processes-o
n-gpus/lib [Y/N]
N

Do you have the libLBFGS library installed (http://www.chokkan.org/software/libl
bfgs)? [Y/N]
Y

Searching for libLBFGS library..
libLBFGS found in the following paths:
Is this your libLBFGS path: . [Y/N]
N
Please enter your libLBFGS path:
/homes/jwn09/usr/lib

Searching for libLBFGS header file..
lbfgs.h found in the following paths:
Is this your lbfgs.h path: . [Y/N]
N
Please enter your lbfgs.h path:
/homes/jwn09/usr/include

Compile the GPU components of the library? [Y/N]
Y

Searching for CUDA libraries..
CUDA libraries found in the following paths:
[1] /usr/lib/matlab/R2013a/bin/glnxa64
[2] /usr/local/cuda/lib
[3] /usr/local/cuda/lib64
Select your CUDA path (enter 0 if not any of the above):
3

Searching for CULA libraries..
CULA libraries found in the following paths:
Is this your CULA path: . [Y/N]
N
[7] 0:bash*                               "graphic01.doc.ic.ac.uk" 13:00 16-May-14
```

Figure 10: Setting up the configuration files

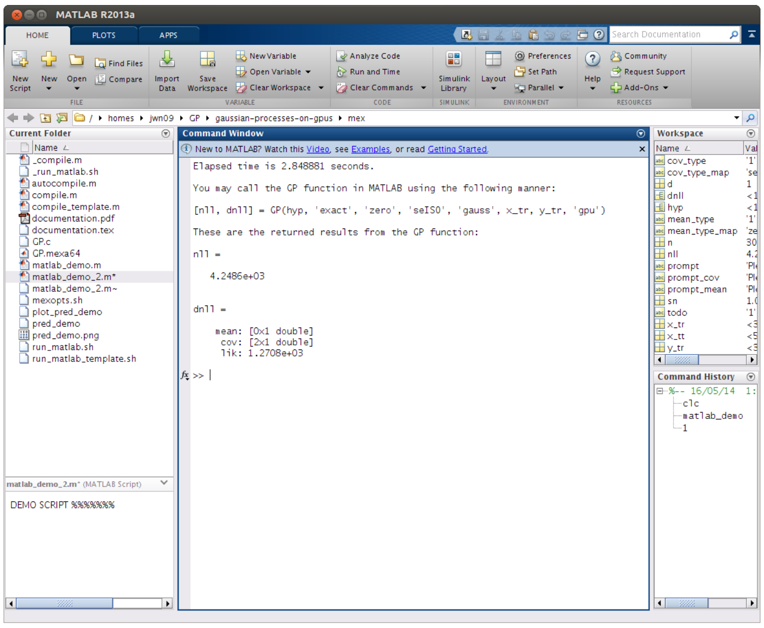
6.2.2 A MATLAB Demo

After the configurations have been set using our script, the user may use our software package natively in MATLAB. Shown in Figures 11 and 12 is a demo for the likelihood function using zero mean and square exponential kernel, isotrophic distance measure.



```
>> matlab_demo
[1] Likelihood function demo
[2] Prediction demo
Please select your demo: 1
[1] Zero Mean
[2] One Mean
Please select mean method: 1
[1] Square Exponential Kernel, Isotropic Distance Measure
Please select cov method: 1
```

Figure 11: Executing a simple MATLAB demo using our software package



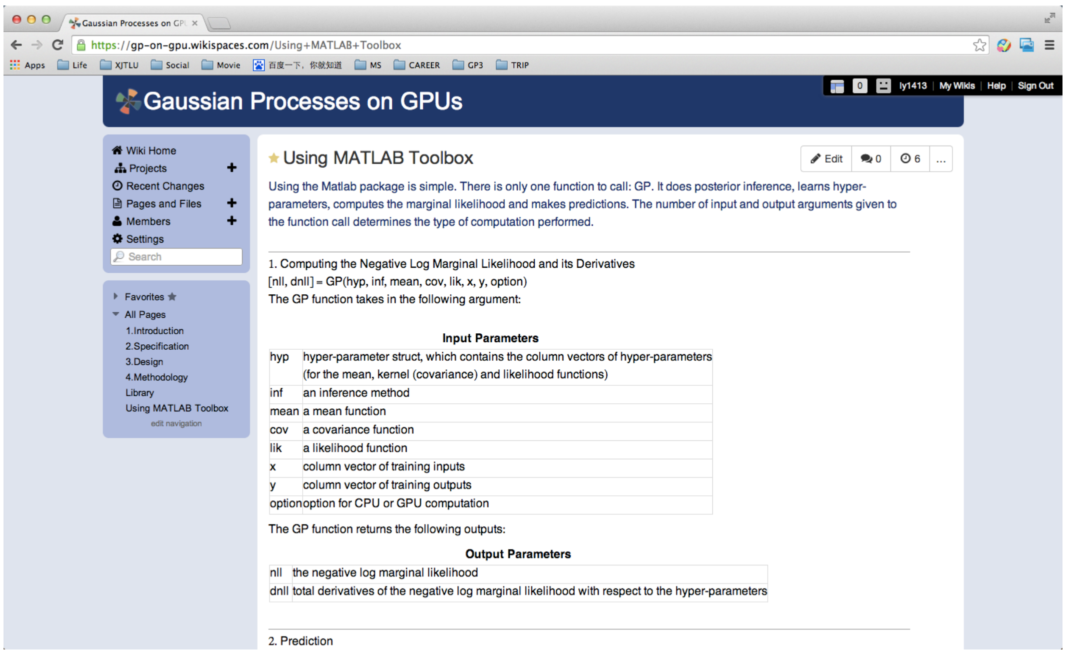
```
Elapsed time is 2.848881 seconds.

You may call the GP function in MATLAB using the following manner:

[nll, dnll] = GP(hyp, 'exact', 'zero', 'seISO', 'gauss', x_tr, y_tr, 'gpu')

These are the returned results from the GP function:

nll =

   4.2486e+03


dnll =

    mean: [0x1 double]
     cov: [2x1 double]
     lik: 1.2708e+03

>>
```

Figure 12: Results from the MATLAB demo

Refer to Appendix A for full documentation on the use of our GP library in MATLAB.

6.3 Wiki Page

The wiki page is hosted on wikispaces[17], and it can be accessed via the URL: `https://gp-on-gpu.wikispaces.com/`. A sample page of the wiki can be seen in Figure 13.



Using MATLAB Toolbox

Using the Matlab package is simple. There is only one function to call: GP. It does posterior inference, learns hyper-parameters, computes the marginal likelihood and makes predictions. The number of input and output arguments given to the function call determines the type of computation performed.

1. Computing the Negative Log Marginal Likelihood and its Derivatives
[nll, dnll] = GP(hyp, inf, mean, cov, lik, x, y, option)
The GP function takes in the following argument:

Input Parameters

hyp	hyper-parameter struct, which contains the column vectors of hyper-parameters (for the mean, kernel (covariance) and likelihood functions)
inf	an inference method
mean	a mean function
cov	a covariance function
lik	a likelihood function
x	column vector of training inputs
y	column vector of training outputs
option	option for CPU or GPU computation

The GP function returns the following outputs:

Output Parameters

nll	the negative log marginal likelihood
dnll	total derivatives of the negative log marginal likelihood with respect to the hyper-parameters

2. Prediction

Figure 13: A sample page of the wiki

It requires the user to have an account in order to view and contribute content on our wiki page. A guest account has been created for convenience with the following credentials:

```
Username: guest_gpgpu
Password: gpgpu
```

6.4 Product Evaluation

6.4.1 Benchmarking

Figure 14 shows that for the marginal likelihood function, the CPU implementation runs faster for smaller input sizes ($N \leq 2000$), after which the GPU implementation fares significantly better. This is expected because while the GPU implementation incurs some overhead, the computation has effectively been distributed across the multiple parallel cores of the GPU, making the time required grow less quickly than the CPU implementation while spreading the overhead over a larger input size.

Figure 15 shows that for the prediction function, the CPU implementation has an advantage over the GPU implementation. This is because as compared to the marginal likelihood function, the prediction function is more memory intensive and requires more host-to-device copying.

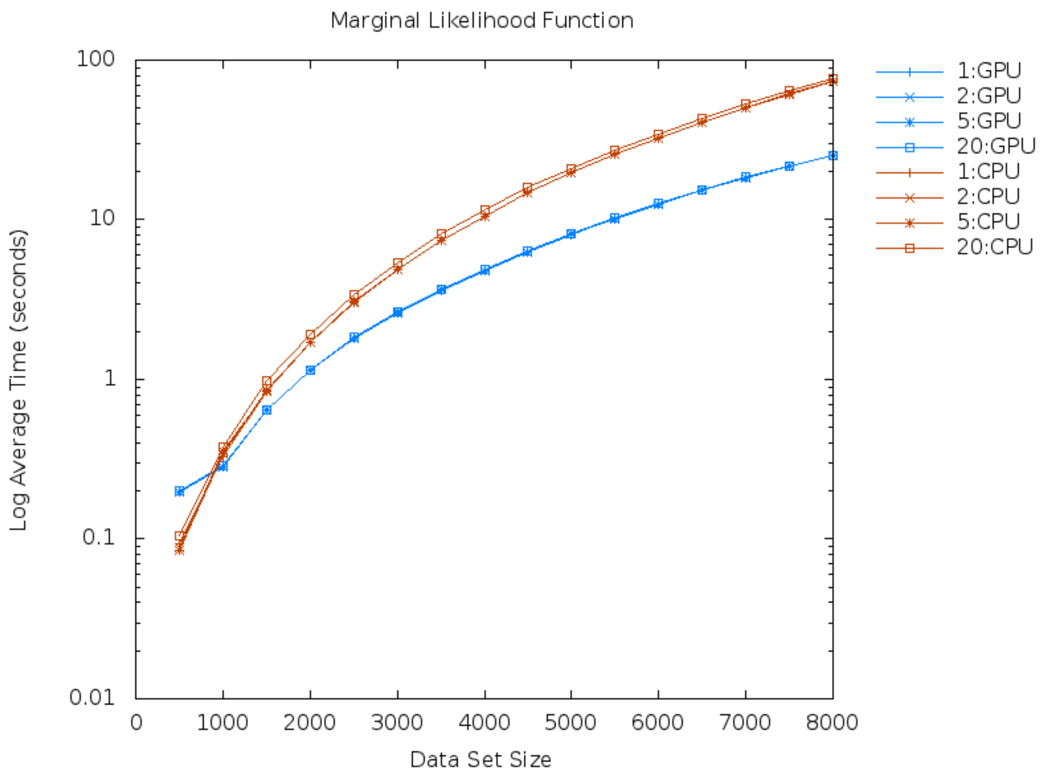


Figure 14: Running time of the likelihood function

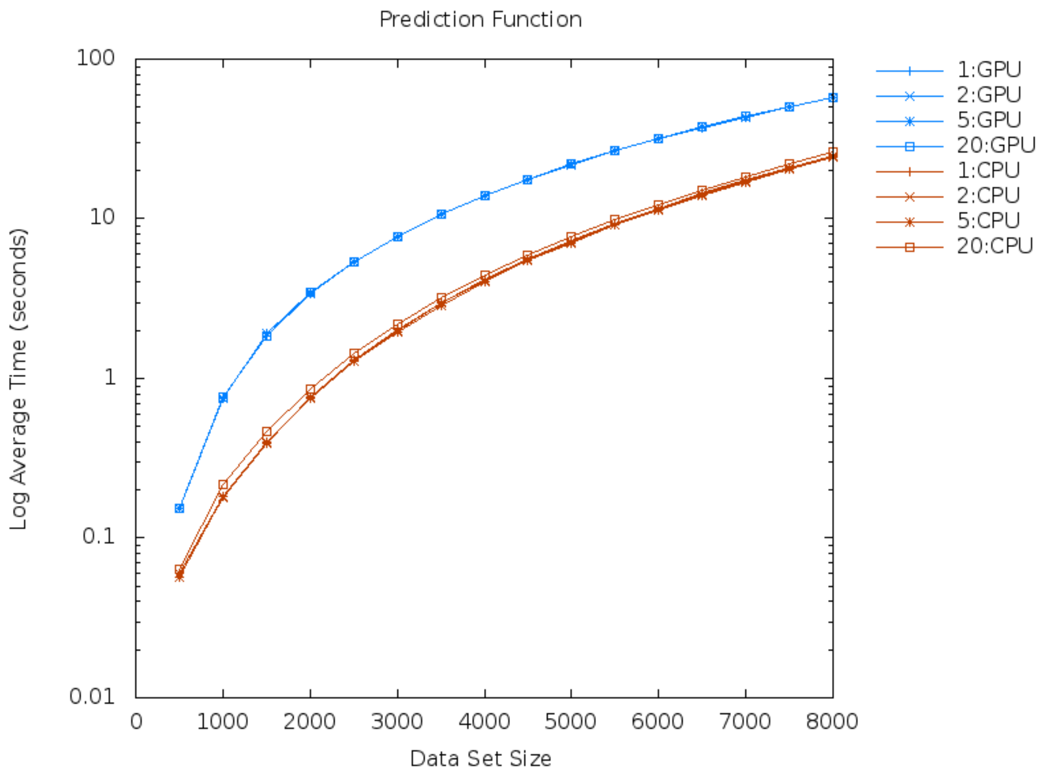
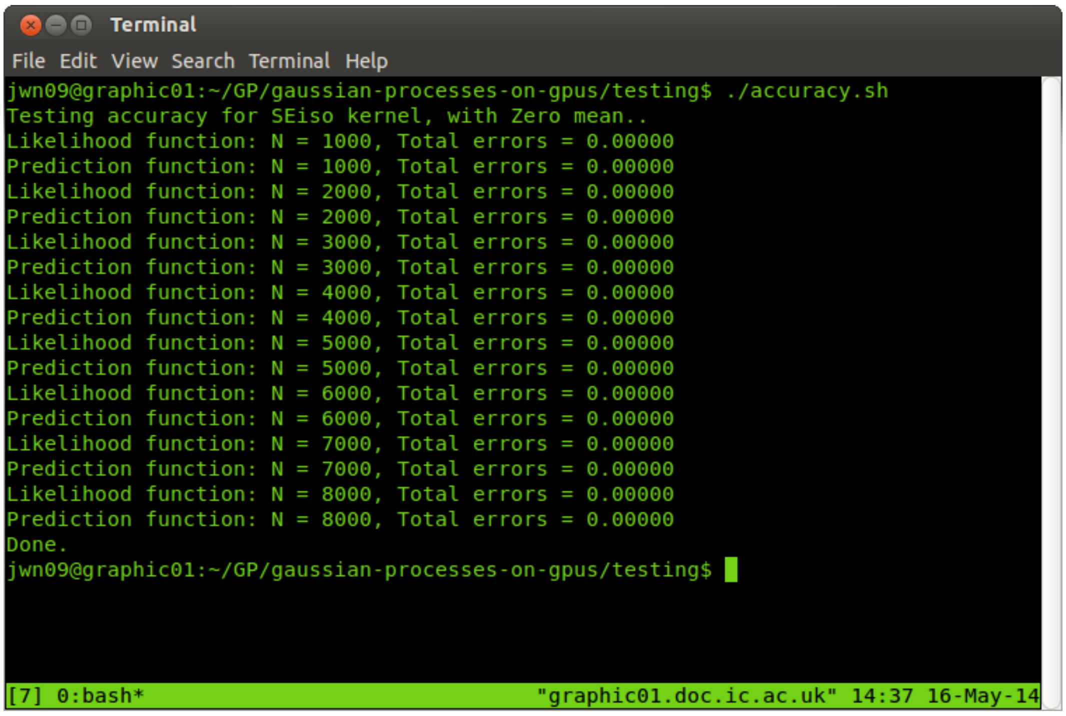


Figure 15: Running timeof the prediction function

6.4.2 Testing and Results



```
jwn09@graphic01:~/GP/gaussian-processes-on-gpus/testing$ ./accuracy.sh
Testing accuracy for SEiso kernel, with Zero mean..
Likelihood function: N = 1000, Total errors = 0.00000
Prediction function: N = 1000, Total errors = 0.00000
Likelihood function: N = 2000, Total errors = 0.00000
Prediction function: N = 2000, Total errors = 0.00000
Likelihood function: N = 3000, Total errors = 0.00000
Prediction function: N = 3000, Total errors = 0.00000
Likelihood function: N = 4000, Total errors = 0.00000
Prediction function: N = 4000, Total errors = 0.00000
Likelihood function: N = 5000, Total errors = 0.00000
Prediction function: N = 5000, Total errors = 0.00000
Likelihood function: N = 6000, Total errors = 0.00000
Prediction function: N = 6000, Total errors = 0.00000
Likelihood function: N = 7000, Total errors = 0.00000
Prediction function: N = 7000, Total errors = 0.00000
Likelihood function: N = 8000, Total errors = 0.00000
Prediction function: N = 8000, Total errors = 0.00000
Done.
jwn09@graphic01:~/GP/gaussian-processes-on-gpus/testing$ 

[7] 0:bash*                              "graphic01.doc.ic.ac.uk" 14:37 16-May-14
```

Figure 16: Accuracy of results

The numerical outputs of both the CPU and GPU versions of the likelihood and predict functions were compared. The absolute value of the differences were summed and displayed. There is no significant discrepancy between the two implementations, which is not surprising since the exact numerical routines were used. In addition, using double precision floating point numbers allows for approximately 15 decimal digits of numerical accuracy, which is a large enough safety of margin for computations involving noisy data.

6.5 Features for Future Developments

Our project is very scalable, and there are rooms for future developments. Here are some possible extensions:

- Porting onto other operating system platforms such as Microsoft Windows
- Wrapping it with other programming modules such as Python or R
- Taking advantage of multiple-GPU machines, where the script can detect the available GPUs on a given machine to exploit them accordingly
- Implementing different Kernel, Mean, Covariance, Likelihood functions and Inference methods.

We attempted to implement the above-mentioned features, but due to a lack of time and also the nature of the complexity of the task, we were unable to do so. Nevertheless, we have documented our project on our wiki, which will enable crowd-sourcing, where interested developers can contribute to building the library.

A Using the Matlab toolbox

Using the Matlab package is simple. There is only one function to call: GP. It does posterior inference, learns hyperparameters, computes the marginal likelihood and makes predictions. The number of input and output arguments given to the function call determines the type of computation performed.

A.1 Computing the Negative Marginal Log Likelihood and its Derivatives

```
[nll, dnll] = GP(hyp, inf, mean, cov, lik, x, y, option)
```

The GP function takes in the following arguments:

Input Parameters	
`hyp`	hyperparameter struct, which contains the column vectors of hyperparameters (for the mean, kernel (covariance) and likelihood functions)
`inf`	an inference method
`mean`	a mean function
`cov`	a covariance function
`lik`	a likelihood function
`x`	column vector of training inputs
`y`	column vector of training outputs
`option`	option for CPU or GPU computation

The GP function returns the following outputs:

Output Parameters	
`nll`	the negative log marginal likelihood
`dnll`	total derivatives of the negative log marginal likelihood with respect to the hyperparameters

A.2 Prediction

$$[\texttt{ymu}, \texttt{ys2}] = \texttt{GP}(\texttt{hyp}, \texttt{inf}, \texttt{mean}, \texttt{cov}, \texttt{lik}, \texttt{x}, \texttt{y}, \texttt{xs}, \texttt{option})$$

The GP function takes in the following arguments.

Input Parameters	
`hyp`	hyperparameter struct, which contains the column vectors of hyperparameters (for the mean, kernel (covariance) and likelihood functions)
`inf`	an inference method
`mean`	a mean function
`cov`	a covariance function
`lik`	a likelihood function
`x`	column vector of training inputs
`y`	column vector of training outputs
`xs`	column vector of test set inputs
`option`	option for CPU or GPU computation

The GP function will then compute the test set predictive probabilities, and return the following outputs.

Output Parameters	
`ymu`	column vector of predictive output means
`ys2`	column vector of predictive output variances

A.3 Parameters

A.3.1 Hyperparameter Struct

The hyperparameter struct contains three fields: mean, cov, lik (some of which may be empty). When specifying hyperparameters, it is important that the number of elements in each of these struct fields, precisely match the number of parameters expected by the mean function, the covariance function and the likelihood functions respectively (otherwise an error will result). Hyperparameters whose natural domain is positive are represented by their logarithms.

Pass in the struct into the GP function, where `hyp` is taken as an argument.

For example, you may create the hyp struct in the following manner in Matlab:

```
hyp.mean = []
hyp.cov = log([0.5 2])
hyp.lik = log(sqrt(var(ys))
```

A.3.2 Inference Methods

The only inference method available is the exact inference method. Pass in the respective string values into the GP function, where `inf` is taken as an argument.

Inference Methods Inputs	
`'exact'`	Exact Inference Method

A.3.3 Mean Functions

You may choose to use the zero, one, or linear functions for the mean functions. Pass in the respective string values into the GP function, where `mean` is taken as an argument.

Mean Functions Inputs	
`'zero'`	Zero Mean
`'one'`	One Mean

A.3.4 Covariance Functions

Currently the only choice of kernel function is the Squared Exponential function with an isotropic distance measure Pass in the respective string values into the GP function, where `cov` is taken as an argument.

Mean Functions Inputs	
`'seISO'`	Squared Exponential Kernel with Isotropic Distance Measure

A.3.5 Likelihood Functions

The only likelihood function available is the Gaussian likelihood function. Pass in the respective string values into the GP function, where `lik` is taken as an argument.

Likelihood Functions Inputs	
`'gauss'`	Gaussian Likelihood Function

A.3.6 Options

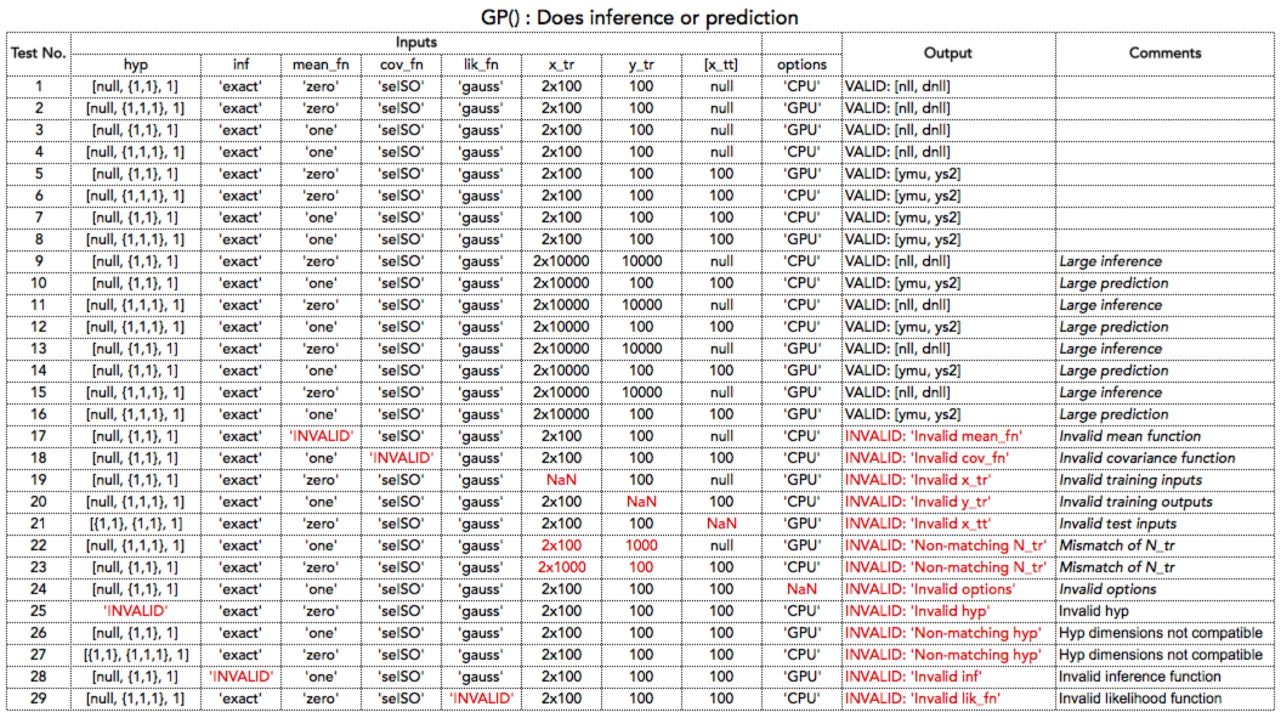
You may choose to use the CPU or GPU computation. Pass in the respective string values into the GP function, where `option` is taken as an argument.

Option Inputs	
`'cpu'`	CPU Implementation
`'gpu'`	GPU Implementation

B Testing

B.1 Testing Matrix

The table below lists the various permutations of inputs to the program and the expected output that should be obtained.

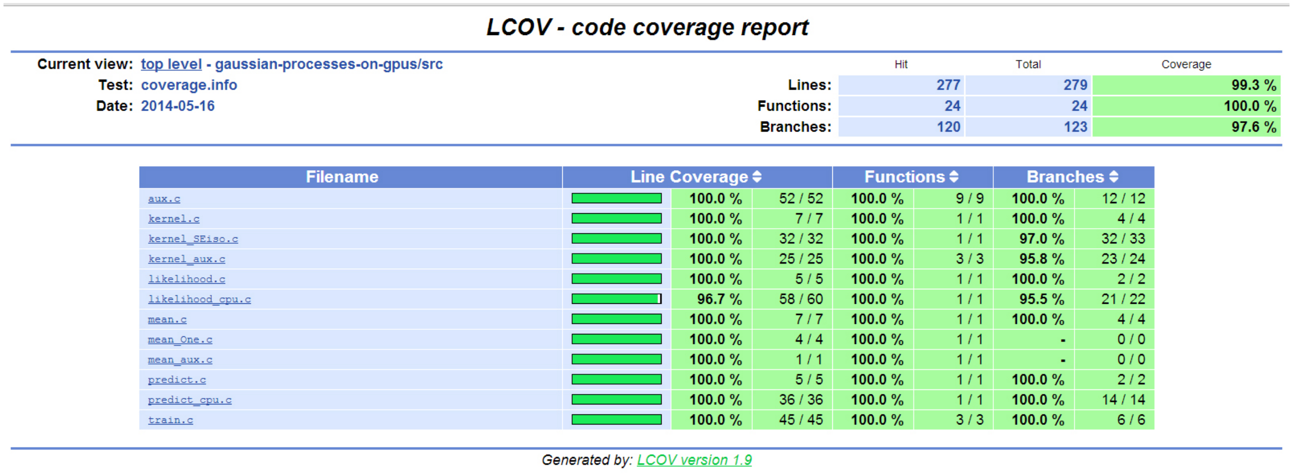


GP() : Does inference or prediction

Test No.	Inputs									Output	Comments
	hyp	inf	mean_fn	cov_fn	lik_fn	x_tr	y_tr	[x_tt]	options		
1	[null, {1,1}, 1]	'exact'	'zero'	'selSO'	'gauss'	2x100	100	null	'CPU'	VALID: [nll, dnll]	
2	[null, {1,1,1}, 1]	'exact'	'zero'	'selSO'	'gauss'	2x100	100	null	'GPU'	VALID: [nll, dnll]	
3	[null, {1,1}, 1]	'exact'	'one'	'selSO'	'gauss'	2x100	100	null	'GPU'	VALID: [nll, dnll]	
4	[null, {1,1,1}, 1]	'exact'	'one'	'selSO'	'gauss'	2x100	100	null	'CPU'	VALID: [nll, dnll]	
5	[null, {1,1}, 1]	'exact'	'zero'	'selSO'	'gauss'	2x100	100	100	'GPU'	VALID: [ymu, ys2]	
6	[null, {1,1,1}, 1]	'exact'	'zero'	'selSO'	'gauss'	2x100	100	100	'CPU'	VALID: [ymu, ys2]	
7	[null, {1,1}, 1]	'exact'	'one'	'selSO'	'gauss'	2x100	100	100	'CPU'	VALID: [ymu, ys2]	
8	[null, {1,1,1}, 1]	'exact'	'one'	'selSO'	'gauss'	2x100	100	100	'GPU'	VALID: [ymu, ys2]	
9	[null, {1,1}, 1]	'exact'	'zero'	'selSO'	'gauss'	2x10000	10000	null	'CPU'	VALID: [nll, dnll]	*Large inference*
10	[null, {1,1}, 1]	'exact'	'one'	'selSO'	'gauss'	2x10000	100	100	'CPU'	VALID: [ymu, ys2]	*Large prediction*
11	[null, {1,1,1}, 1]	'exact'	'zero'	'selSO'	'gauss'	2x10000	10000	null	'CPU'	VALID: [nll, dnll]	*Large inference*
12	[null, {1,1,1}, 1]	'exact'	'one'	'selSO'	'gauss'	2x10000	100	100	'CPU'	VALID: [ymu, ys2]	*Large prediction*
13	[null, {1,1}, 1]	'exact'	'zero'	'selSO'	'gauss'	2x10000	10000	null	'GPU'	VALID: [nll, dnll]	*Large inference*
14	[null, {1,1}, 1]	'exact'	'one'	'selSO'	'gauss'	2x10000	100	100	'GPU'	VALID: [ymu, ys2]	*Large prediction*
15	[null, {1,1,1}, 1]	'exact'	'zero'	'selSO'	'gauss'	2x10000	10000	null	'GPU'	VALID: [nll, dnll]	*Large inference*
16	[null, {1,1,1}, 1]	'exact'	'one'	'selSO'	'gauss'	2x10000	100	100	'GPU'	VALID: [ymu, ys2]	*Large prediction*
17	[null, {1,1}, 1]	'exact'	'INVALID'	'selSO'	'gauss'	2x100	100	null	'CPU'	INVALID: 'Invalid mean_fn'	*Invalid mean function*
18	[null, {1,1}, 1]	'exact'	'one'	'INVALID'	'gauss'	2x100	100	100	'CPU'	INVALID: 'Invalid cov_fn'	*Invalid covariance function*
19	[null, {1,1}, 1]	'exact'	'zero'	'selSO'	'gauss'	NaN	100	null	'GPU'	INVALID: 'Invalid x_tr'	*Invalid training inputs*
20	[null, {1,1,1}, 1]	'exact'	'one'	'selSO'	'gauss'	2x100	NaN	100	'CPU'	INVALID: 'Invalid y_tr'	*Invalid training outputs*
21	[{1,1}, {1,1}, 1]	'exact'	'zero'	'selSO'	'gauss'	2x100	100	NaN	'GPU'	INVALID: 'Invalid x_tt'	*Invalid test inputs*
22	[null, {1,1,1}, 1]	'exact'	'one'	'selSO'	'gauss'	2x100	1000	null	'GPU'	INVALID: 'Non-matching N_tr'	*Mismatch of N_tr*
23	[null, {1,1}, 1]	'exact'	'zero'	'selSO'	'gauss'	2x1000	100	100	'CPU'	INVALID: 'Non-matching N_tr'	*Mismatch of N_tr*
24	[null, {1,1}, 1]	'exact'	'one'	'selSO'	'gauss'	2x100	100	100	NaN	INVALID: 'Invalid options'	*Invalid options*
25	'INVALID'	'exact'	'zero'	'selSO'	'gauss'	2x100	100	100	'CPU'	INVALID: 'Invalid hyp'	Invalid hyp
26	[null, {1,1}, 1]	'exact'	'one'	'selSO'	'gauss'	2x100	100	100	'GPU'	INVALID: 'Non-matching hyp'	Hyp dimensions not compatible
27	[{1,1}, {1,1,1}, 1]	'exact'	'zero'	'selSO'	'gauss'	2x100	100	100	'CPU'	INVALID: 'Non-matching hyp'	Hyp dimensions not compatible
28	[null, {1,1}, 1]	'INVALID'	'one'	'selSO'	'gauss'	2x100	100	100	'GPU'	INVALID: 'Invalid inf'	Invalid inference function
29	[null, {1,1,1}, 1]	'exact'	'zero'	'selSO'	'INVALID'	2x100	100	100	'CPU'	INVALID: 'Invalid lik_fn'	Invalid likelihood function

B.2 Coverage Report

This is the code coverage report generated by `lcov`



LCOV - code coverage report

Current view: top level - gaussian-processes-on-gpus/src
Test: coverage.info
Date: 2014-05-16

	Hit	Total	Coverage
Lines:	277	279	99.3 %
Functions:	24	24	100.0 %
Branches:	120	123	97.6 %

Filename	Line Coverage			Functions		Branches	
aux.c	🟩	100.0 %	52 / 52	100.0 %	9 / 9	100.0 %	12 / 12
kernel.c	🟩	100.0 %	7 / 7	100.0 %	1 / 1	100.0 %	4 / 4
kernel_SEiso.c	🟩	100.0 %	32 / 32	100.0 %	1 / 1	97.0 %	32 / 33
kernel_aux.c	🟩	100.0 %	25 / 25	100.0 %	3 / 3	95.8 %	23 / 24
likelihood.c	🟩	100.0 %	5 / 5	100.0 %	1 / 1	100.0 %	2 / 2
likelihood_cpu.c	🟩	96.7 %	58 / 60	100.0 %	1 / 1	95.5 %	21 / 22
mean.c	🟩	100.0 %	7 / 7	100.0 %	1 / 1	100.0 %	4 / 4
mean_One.c	🟩	100.0 %	4 / 4	100.0 %	1 / 1	-	0 / 0
mean_aux.c	🟩	100.0 %	1 / 1	100.0 %	1 / 1	-	0 / 0
predict.c	🟩	100.0 %	5 / 5	100.0 %	1 / 1	100.0 %	2 / 2
predict_cpu.c	🟩	100.0 %	36 / 36	100.0 %	1 / 1	100.0 %	14 / 14
train.c	🟩	100.0 %	45 / 45	100.0 %	3 / 3	100.0 %	6 / 6

Generated by: LCOV version 1.9

C Logbook

Week 1, 06 Jan 2014

1. Introduction to GP and key difficulties
2. Introduction to CUDA and explain how it works/communicates with the CPU - GPU structure, memory allocation, value copy etc.
3. Discuss criteria of testing: memory and computational throughput
4. Discuss structure of report:
   - Software development approach
   - Scope of project: basic, bonus, testing framework and wrapping into MATLAB
   - Core: mathematical breakdown and GPU implementation, find inverse or matrices, implement parallel programming, combine all the parallel outcomes
   - Testing/benchmarking: test cases, data, formulating of the questions, driver functions, CPU implementation
   - Wrapping into MATLAB
5. Job allocation:

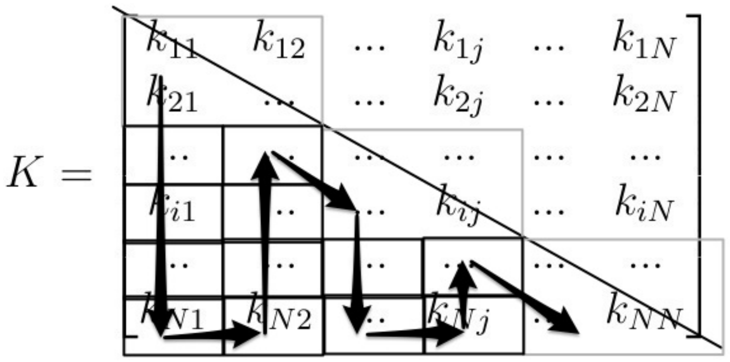
	Core		Testing			Wrapping	Styling&Drafting
	CPU	GPU	Test Cases	Driver functions	CPU		
Jun Wei	✓						
Yuanruo		✓				✓	
Deng Ying		✓					
Edward				✓	✓		✓
Xiaonan			✓				
Terence						✓	

Week 2, 13 Jan 2014

1. Discuss library dependencies, find all the dependencies and document them
2. Grasp a common understanding of regression
3. Develop idea for the platform, measure statistics for testing, discuss possible datasets
4. Different parts co-working together: create a software package could be difficult, agreed on interfaces
5. Complete demo scripts in C and MATLAB

Week 3, 20 Jan 2014

1. CUDA library memory management: `http://developer.download.nvidia.com/compute/cuda/4_1/rel/toolkit/docs/online/group__CUDART__MEMORY.html`
2. Dealing with memory management in cases where datasets are too large for the GPU to process as a whole. In the case of both extreme sizes and dimensions, one key concern in this memory management is that data copying from CPU to GPU will take up an excessive amount of time and slow down the execution to a great extent. One possible solution would be to copy resed data into GPU multiple times to form different combinations of data sets. Based on this re-copying situation, it might be better to implement the computation with a Z-shaped structure, which would mean maintaining half of the data unchanged and replacing the other half each loop, and complete the computation for a whole of the part column/row of the kernel matrix before moving to other parts of columns/rows. The basic structure of the solution graphically will be as:



$K =$
k_{11} k_{12} $\cdots$ k_{1j} $\cdots$ k_{1N}
k_{21} $\cdots$ $\cdots$ k_{2j} $\cdots$ k_{2N}
k_{i1} k_{ij} k_{iN}
k_{N1} k_{N2} k_{Nj} k_{NN}

Figure 17: Sub-matrix computation by part

- Check input data according to size limitation
- Decide whether data set size (length) is too large
- Decide whether data dimension is too large

3. Draft coding style guideline:
   - Naming conventions
   - Comments
   - Formatting
4. Compile CUDA code and wrap that into MATLAB.
5. Review documentation for GPML MATLAB Code version 3.4:
   `http://www.gaussianprocess.org/gpml/code/MATLAB/doc/`

Week 4, 27 Jan 2014
Note: 31 Jan (Fri), Submission of Report 1

1. Discuss the development approach part, according to
   `http://www3.imperial.ac.uk/computing/internal/students/noticeboards/msc/grp-proj`, requires:
   - Development strategy chosen by your group
   - How we're using it in our context
   - Division of work
2. Update the current draft to include references and the changes Dr. Marc suggested.
3. Update the gantt chart (extend research and learning, create a distinction between current progress and rest of time line), add more references.

Week 5, 03 Feb 2014

1. Set up Gitlab repository:
   - Linear folder is complete (i.e. it has everything needed to compile into a library and the scripts needed to compile for MATLAB) and we'll use this as a reference for the Gaussian processes version
   - In the main repository directory:

  - src directory (contains the code we have so far for the core library) for Yuanruo and Ying. This will be where we will be adding stuff to (probably will create a GPU subdirectory)
  - testing directory (not created yet) for Edward and Lixiaonan
  - mex directory (not created yet, but similar to the mex directory in the Linear example) for Terence. GPML toolbox will give a good idea of what the end product should look like

Week 6, 10 Feb 2014

1. Clarify about derivatives, and showed Dr. Marc the CPU implementations in C
2. Linking of libraries was a little tricky
3. Report graphics machine resources issue, constantly in used by others
4. Discuss performance, GPU is 20% faster than CPU, due to overheads in MATLAB
5. Find a way to optimse based on different GPU/available memory

Week 7, 17 Feb 2014

1. Look at the Broyden Fletcher Goldfarb Shanno library although it is quite old
2. Add kernel mean functions, and implement GPU part
3. Suggest possible training datasets: `www.gaussianprocess.org/gpml/data/`
4. Use two kinds of data for testing: Real World Data and Syntactic generated data. Suggested data sets: `www.esrl.noaa.gov/gmd/ccgg/trends/`
5. Set up a Wikipage/website for documentation.

Week 8, 24 Feb 2014
Note: Exam week

Week 9, 03 Mar 2014

1. Draft report 2:
   - Discuss project progress and overall testing strategy
   - Use coverage measurement tool and report the statement and branch coverage achieved in code, summary of coverage results

Week 10, 10 Mar 2014
Note: 14 Mar (Fri), Submission of Report 2

Week 11, 17 Mar 2014

1. Discuss BFGS library, except that linking with MATLAB might still be a problem.
2. For symmetric matrix, packed storage, which increases the limits of problem size by 2x.
3. Working on Kernel matrix, currently still buggy.

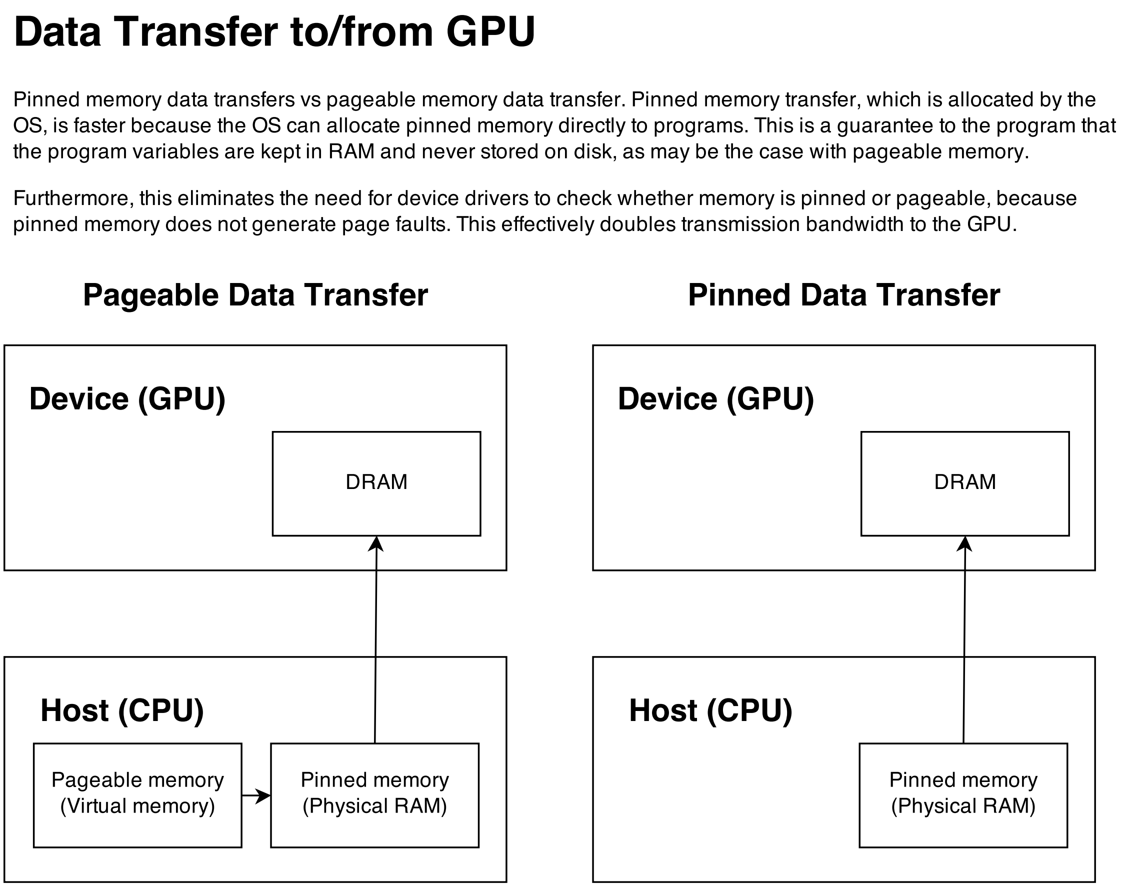
Week 12, 24 Mar 2014
Note: Exam week

Week 13, 31 Mar 2014

1. Use graphic01: add the following to .cshrc to setup nvcc automatically graphic01 is logged on
```
if ( 'hostname' == "graphic01.doc.ic.ac.uk" ) then
source <PATH>/setup.csh
endif
where PATH is wherever you store setup.csh (get the modified version from our dropbox/comm
for this to work)
```

Week 14, 07 Apr 2014

1. Draft final report:
   - Section 4: Methodology. Combine the parts about the software engineering approach and methodology from the previous two reports (Terence)
   - Section 5: Group Work. (Lixiaonan and Edward)
   - Section 6: Final Product. Generate some graphs describing the performance issues of Gaussian processes (Junwei)
   - GPU performance. A section describing the issues relating to the GPU implementation (Yuanruo and Ying)



Data Transfer to/from GPU

Pinned memory data transfers vs pageable memory data transfer. Pinned memory transfer, which is allocated by the OS, is faster because the OS can allocate pinned memory directly to programs. This is a guarantee to the program that the program variables are kept in RAM and never stored on disk, as may be the case with pageable memory.

Furthermore, this eliminates the need for device drivers to check whether memory is pinned or pageable, because pinned memory does not generate page faults. This effectively doubles transmission bandwidth to the GPU.

Figure 18: Pinned host m emory

Week 15, 14 Apr

1. Consider how to optimize data transfers in CUDA C/C++:
   `http://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/`
2. Testing for 2000 datasets
3. Compute sub-matrix, store the full matrix in CPU

4. Discuss Parallel computing and storing

5. Test functionality: training and predicting, MATLAB interface

6. Future software plan: public available, 4 pages abstract + documentation + well tested code (for machine learning journal), can be done after the project.

7. Review second report feedback: details to show the portability, block diagram to visualise the process.

Week 16, 21 Apr 2014
Note: Exam week

Week 17, 28 Apr 2014
Note: Exam week
Setting Wikipage: `https://gp-on-gpu.wikispaces.com/`

Week 18, 05 May 2014
Note: Exam week

Week 19, 12 May 2014
Note: 16 May (Fri), Submission of Report 3

1. Review reference about MAGMA: `http://icl.cs.utk.edu/magma/`

2. Show testing results:

    - Exhibit A:

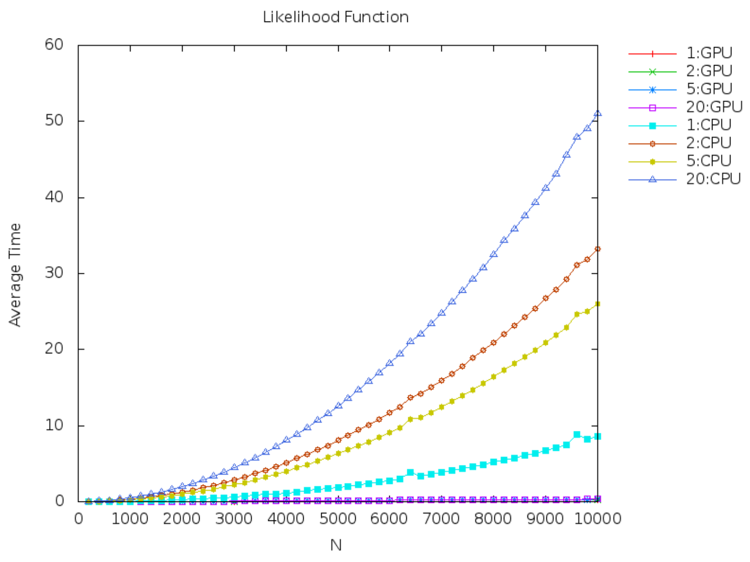


Figure 19: Likelihood function for kernel

Shows how the GPU kernel does really well compared to the CPU kernel. This is only run to 10000, we will run to 20000 later

- Exhibit B:

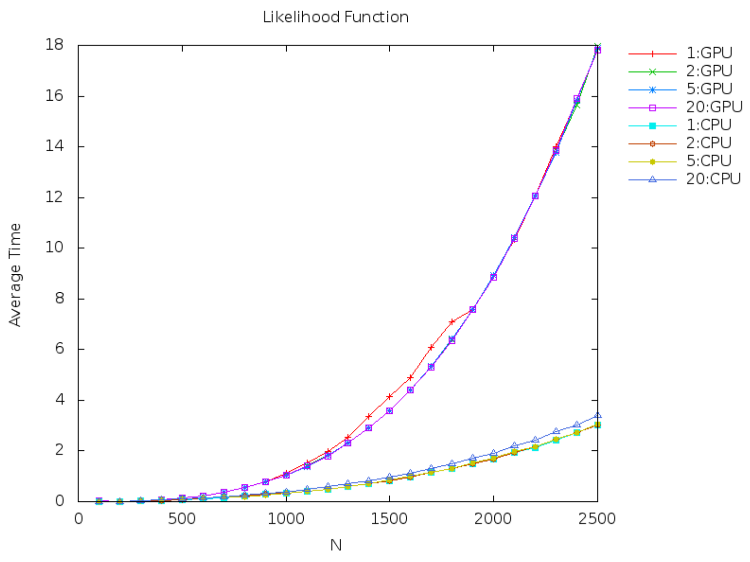
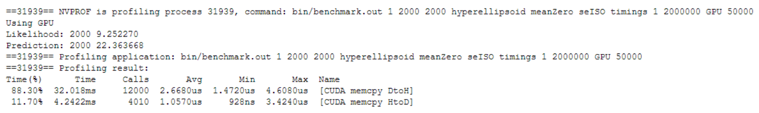


Figure 20: Full computation results likelihood function

Try switching out MAGMA for just LAPACK, or hand-coding the linear algebra

3. Profiling of the likelihood and prediction functions:



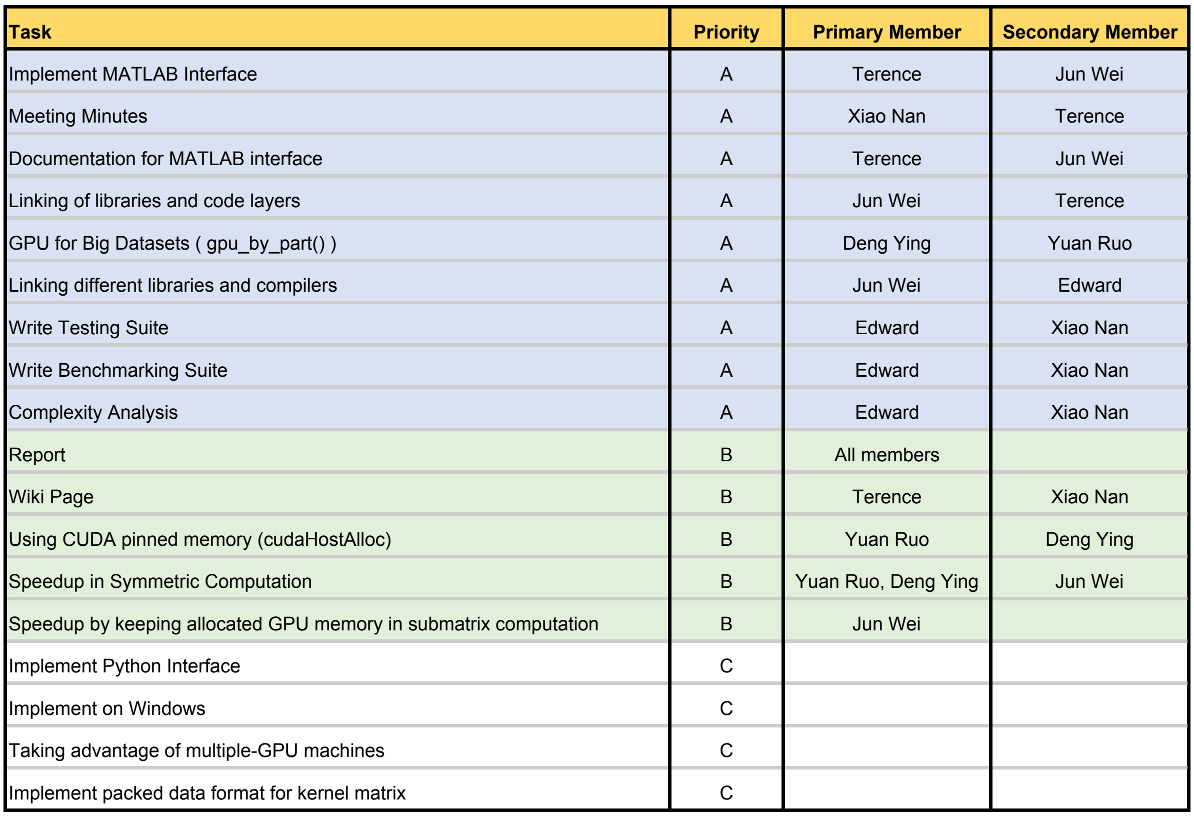
```
==31939== NVPROF is profiling process 31939, command: bin/benchmark.out 1 2000 2000 hyperellipsoid meanZero seISO timings 1 2000000 GPU 50000
Using GPU
Likelihood: 2000 9.252270
Prediction: 2000 22.363668
==31939== Profiling application: bin/benchmark.out 1 2000 2000 hyperellipsoid meanZero seISO timings 1 2000000 GPU 50000
==31939== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 88.30%  32.018ms     12000  2.6680us  1.4720us  4.6080us  [CUDA memcpy DtoH]
 11.70%  4.2422ms      4010  1.0570us     928ns  3.4240us  [CUDA memcpy HtoD]
```

Figure 21: Profiling likelihood and prediction functions

4. Implement a full CPU profile using gprof.

Week 20, 19 May 2014
Note: 20 May (Tue), Group Project Presentation, Session Three

D Detailed Work Breakdown



Task	Priority	Primary Member	Secondary Member
Implement MATLAB Interface	A	Terence	Jun Wei
Meeting Minutes	A	Xiao Nan	Terence
Documentation for MATLAB interface	A	Terence	Jun Wei
Linking of libraries and code layers	A	Jun Wei	Terence
GPU for Big Datasets (gpu_by_part())	A	Deng Ying	Yuan Ruo
Linking different libraries and compilers	A	Jun Wei	Edward
Write Testing Suite	A	Edward	Xiao Nan
Write Benchmarking Suite	A	Edward	Xiao Nan
Complexity Analysis	A	Edward	Xiao Nan
Report	B	All members	
Wiki Page	B	Terence	Xiao Nan
Using CUDA pinned memory (cudaHostAlloc)	B	Yuan Ruo	Deng Ying
Speedup in Symmetric Computation	B	Yuan Ruo, Deng Ying	Jun Wei
Speedup by keeping allocated GPU memory in submatrix computation	B	Jun Wei	
Implement Python Interface	C		
Implement on Windows	C		
Taking advantage of multiple-GPU machines	C		
Implement packed data format for kernel matrix	C		

Figure 22: Designed Work Breakdown with Priority

Priority A takes highest precedence in our task list, and Priority C takes lowest precedence, which will be implemented should time allow.

References

[1] C. E. Rasmussen and C. K. I. Williams. "Gaussian Processes for Machine Learning". In: (2006).

[2] J. D. Owens. "GPU Computing". In: *Proceedings of the IEEE* 96 (5 2008), pp. 879–899.

[3] By Bernhard Schlkopf and Alexander J. Smola. "Learning with Kernels: Support Vector Machines, Regularization, Optimization and Beyond". In: (2001).

[4] *CUDA Parallel Computing—GPU Computing on the CUDA Architecure.* URL: `http://www.nvidia.co.uk/object/cuda-parallel-computing-uk.html` (visited on 01/21/2014).

[5] *BLAS (Basic Linear Algebra Subprograms).* URL: `http://netlib.org/blas/` (visited on 01/21/2014).

[6] *LAPACK(Linear Algebra PACKage).* URL: `http://netlib/org/LAPACK/` (visited on 01/21/2014).

[7] *MAGMA (Matrix Algebra on GPU and Multicore Architectures).* URL: `http://icl.utk.edu/magma/` (visited on 01/21/2014).

[8] Carl Edward Rasmussen and Hannes Nickisch. *Documentation for GPML Matlab Code version 4.3.* URL: `http://www.gaussianprocess.org/gpml/code/matlab/doc/` (visited on 01/21/2014).

[9] *libLBFGS: a library of Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS).* URL: `http://www.chokkan.org/software/liblbfgs/` (visited on 05/10/2014).

[10] Dong C. Liu and Jorge Nocedal. "On the Limited Memory BFGS Method for Large Scale Optimization". In: *Mathematical Programming 45 (1989)* (1989), pp. 503–528.

[11] *Internetworking Basics.* URL: `http://www.cisco.com/cpress/cc/td/cpress/fund/ith/ith01gb.htm` (visited on 01/21/2014).

[12] *CUDA GPUs — Nvidia Developer Zone.* URL: `https://developer.nvidia.com/cuda-gpus` (visited on 05/15/2014).

[13] Jeff Sutherland and Ken Schwaber. "The Scrum Papers: Nuts, Bolts, and Origins of an Agile Process". In: *Library, Scrum Foundation* (1993).

[14] *Scrum Foundation: Jeff Sutherand.* URL: `http://scrumfoundation.com/about/jeff-sutherland` (visited on 01/21/2014).

[15] *CUDA Occupancy Calculator.* URL: `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls` (visited on 05/12/2014).

[16] ISO/IEC. "Software Product Evaluation - Quality Characteristics and Guidelines". In: (1991).

[17] *About Wiki spaces.* (Visited on 01/21/2014).