CHALMERS

Efficient GPU implementation of parameter estimation of a statistical model for online advertisement optimization

Master's Thesis

ALBIN LINDBÄCK

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Gothenburg, Sweden, June 2012

Efficient GPU implementation of parameter estimation of a statistical model for online advertisement optimization

ALBIN LINDBÄCK

Examiner: ULF ASSARSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2010

Abstract

The optimization problem of estimating parameters using a maximum a-posterior (MAP) [3] approach on a non-linear statistical model with a large data set can be solved using an L-BFGS [10] algorithm. When dealing with an ever changing reality, the evaluation need to be fast to capture the immediacy of the observations. This thesis will present the implementation of the problem objective function and its gradient being used in the numerical iterative optimization algorithm. In order to speed up the process of parameter estimation, an implementation is presented which utilizes the massively parallel computation power of a graphics processing unit (GPU). The implementations are done for both the CPU and the GPU, using C++ and NVIDIA's programming platform CUDA. Compared to the sequential CPU implementation, the result of the parallel GPU version is a speed up of between 20 and 50 for the objective function and around 4 for the gradient.

Keywords: Statistical model, CUDA, GPGPU, Concurrent Programming

Acknowledgments

First of all I would like to thank Torbjörn Westerlid for putting up with all my questions and the all help getting a better understanding of statistics. A special thanks to my supervisor at Admeta, Anders Sjögren, for all the support and expertise. I would also like to thank my examiner at Chalmers University of Technology, Ulf Assarsson, for his help and feedback. Last but not least, I would like to give my gratitude to all the employees of Admeta welcoming me into their inspiring work environment.

Contents

Abbreviations

API	Application Programming Interface
CPA	Cost Per Action
CPC	Cost Per Click
CPM	Cost Per Mille
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit
L-BFGS	Limited Memory Broyden–Fletcher–Goldfarb–Shanno
MAP	Maximum a-posterior
SIMT	Single-Instruction, Multiple-Threads

1 Introduction

1.1 Background

The company Admeta targets their product Admeta Whitebox Tango towards online publishers, where each publisher owns several ad placements. These ad placements are then used by advertisers, providing the publisher with ad materials which are displayed on the placements. An impression is made when a visitor to one of these websites is shown a material, the visitor may then (if interested) click on the material and even perform some type of action on the advertiser's web page. The online publishers get paid by the advertisers per thousand impressions (CPM), clicks (CPC) and/or actions (CPA). To maximize the revenue, showing the right material to the right viewer is crucial in order to increase the possibility for a click (and/or action). For each impression, the expected revenue for each available material given a number of covariates (variables in a statistical model) is evaluated.

A statistical model is utilized to predict the number of subsequent clicks and the number of each type of action. The expected revenue is formed by combining these numbers with the CPM, CPC and CPA bids presented by the advertiser. Given that there are dozens of covariates and millions of parameters where billions of observations are used in order to estimate the parameters, a lot of computation time is needed. But sudden dynamics needs to be captured and new models needs to be evaluated fast in order to give an accurate estimation in the right time-span.

1.2 Outline

The structure of the thesis is divided into to three main Sections: 'Problem Description', 'Implementation' and 'Results and Conclusion'.

The first Section, Problem Description, presents the problem and describes the domain of the statistical model provided by Admeta being implemented.

The second Section, Implementation, contains both the CPU and the GPU implementation of the statistical model described in Section one.

In the third Section, Results and Conclusion, the implementations are tested and evaluated. This Section also presents the conclusion of the thesis.

Finally, appendix A describes the CUDA architecture as well as optimization techniques considered when implementing the GPU version of the model.

1.3 Purpose

The purpose of this thesis is to explore the possible speed increase of implementing a parallel GPU implementation of a statistical model supplied by Admeta. This model includes the objective function (the logarithmic maximum a-posteriori function [3]) and its gradient, to be used in the optimization algorithm L-BFGS [8]. The goal is to find the best way of implementing the function and its gradient, taking advantage of the massive parallelization of the GPU. Aiming to increase the execution speed with a factor of at least ten compared to a reference CPU implementation.

1.4 Scope

The GPU implementation described in this thesis is using the NVIDIA Fermi architecture, with CUDA C/C++ v4.1 as the programming language. The implementations are executed on a NVIDIA GeForce GTX 570. The CPU version of the statistical model is implemented in C++ and both versions will be developed in Microsoft Visual Studio 2010 under Windows 7.

The choice to use CUDA instead of other GPU programming frameworks such as OpenCL was made by Admeta with the argument of better debug tools as well as C++ support.

The model is a simplified version of what Admeta uses and limited to only CPC actions (clicks). However, the design of the implementation should provide future proof of adding other actions. With the future intention of being able to fully execute an L-BFGS algorithm on the GPU, the complete set observations should be able to fit in the device memory.

1.5 Related work

The research and use of general purpose computing using GPUs has been increasingly popular over the recent years, mostly due to improved APIs and hardware support [17]. The speedup achieved by GPU parallelization has been applied to many different application areas. The book GPU Computing Gems [16] presents 50 articles written by researchers in ten different domains such as medical imaging, ray tracing and rendering as well as scientific simulation. NVIDIA also features over 1000 applications in different research areas on their CUDA Zone website [18]. In the field of general statistical modeling and data-intensive applications two articles was found. In Large-Scale Machine Learning, Hwu describes a GPU implementation of a maximum entropy learning algorithm with a large data set [16]. Suchard et al [15] provide an insight in statistical computation for large-scale data analysis in structured Bayesian mixture models using GPUs.

Several optimization techniques for general CUDA implementations have been found and evaluated.

In the presentation Better Performance at Lower Occupancy [9], Vasily Volkov presents the idea to hide memory latency using fewer threads. By using instruction level parallelism together with thread level parallelism, Volkov shows that high throughput can be achieved even at lower occupancy.

Techniques for higher instruction throughput are introduced by Wang in Fundamental Optimizations in CUDA [2]. Some of the techniques mentioned in the slides are reducing the number of instructions by using templates and reducing divergence and bank conflicts. Wang also presents some alternative instructions to replace expensive ones.

Several different approaches to parallel reduction are suggested in Optimizing Parallel Reduction in CUDA [11], where Mark Harris discusses different optimizations techniques to speed up the summarization of a large data set.

2 Problem Description

2.1 Core Technology

In order to maximize the revenue for the publisher, selecting the best material for each impression is important. This is achieved by evaluating the expected revenue of all participating materials, given a set of covariates of current interest. To calculate the expected revenue, a statistical model is used to predict the number of subsequent clicks for each material. The expected revenue is the result of combining the prediction with the CPC bids, given by the advertiser. The statistical model contains several covariates and millions of parameters, which are estimated by tens of billions of observations.

When evaluating the expected revenue, it is important that the parameters are close to optimal. Therefore, the estimation of said parameters must both be accurate and done fast to allow sudden dynamics to be captured. To estimate the parameters, the iterative limited-memory quasi-Newton optimization algorithm, L-BFGS [10] has been chosen. The algorithm and why it has been used, as well as performance evaluation can be found in "Application of L-BFGS to a Large-Scale Poisson MAP Estimation" [8]. The performance bottleneck of such an algorithm is the evaluation of the objective function and its gradient (when the number of observations is extremely large). Since the goal is to decrease the time taken per iteration, parallel implementations of the objective function as well as its gradient will be done on the GPU. For comparison, the model will also be implemented on the CPU.

2.2 The Statistical Model

Each impression a visitor makes when shown a material is stored as an observation. Each observation contains covariates representing different effects such as which material was used and the placement of the material. Along with the covariates, the observation also includes the number of clicks the impression resulted in. To predict the number of clicks, a large set of collected observations is used together with a set of parameters describing the contribution each covariate has for a click.

Covariate	Parameters
(*intercept*)	θ^{lI}
Advertiser	θ_d^{lD}
Impression frequency	θ_f^{lF}
Material	θ_m^{lM}
Order	θ_r^{lR}
Placement	θ_p^{lP}
Material & Placement	$\boldsymbol{\theta}_m^{M:P}$,$\boldsymbol{\theta}_p^{P:M}$

Table 1: The parameters describing the effect of the different covariates

For an observation, let the outcome (number of clicks) be denoted as Y, covariates as X and the set of all parameters as $\boldsymbol{\theta}$. Then the expected number of clicks will be $\mathbb{E}[Y|\boldsymbol{\theta}, X]$, for convenience denoted as μ.
A log-additive model is assumed for the mean:

$$log(\mu) = \theta^{lI} + \theta_d^{lD} + \theta_f^{lF} + \theta_m^{lM} + \theta_r^{lR} + \theta_p^{lP} + \boldsymbol{\theta}_m^{M:P} \cdot \boldsymbol{\theta}_p^{P:M} \tag{1}$$

The outcome is assumed to be Poisson distributed:

$$f(Y = y|\boldsymbol{\theta}, X) = \frac{(\mu)^y e^{-\mu}}{y!}$$

In order to prevent over fitting the model, regularization of the parameter estimates are required. This is done by assuming prior distribution, which represents the uncertainty about the parameters before the observations have been taken into consideration [3].

$$f(\theta^\star) \sim \mathcal{N}(0, \psi^\star)$$

The hyperparameter, ψ, used as the variance of the normal distribution, is fixed and given for each class, $\star$, of parameters (table 1).

$$\psi^\star \in (0, \infty) \tag{2}$$

To find the a-posterior density of the parameters, the probability density of the parameters given the observations, Bayes theorem is used:

$$f(\boldsymbol{\theta}|\boldsymbol{X}, Y) = \frac{f(Y|\boldsymbol{\theta}, \boldsymbol{X}) \cdot f(\boldsymbol{\theta}|\boldsymbol{X})}{f(Y|\boldsymbol{\theta})}$$

Assuming $f(\boldsymbol{\theta}) = f(\boldsymbol{\theta}|\boldsymbol{X})$, the proportionality with respect to $\boldsymbol{\theta}$:

$$f(\boldsymbol{\theta}|\boldsymbol{X}, Y) \propto f(Y|\boldsymbol{\theta}, \boldsymbol{X}) \cdot f(\boldsymbol{\theta})$$

To estimate the expectation of the outcome (number of clicks) for future impressions, the appropriate parameters must be found. This is achieved by finding the $\boldsymbol{\theta}$ that maximizes the posterior distribution. Thus the optimization problem is to find said $\boldsymbol{\theta}$.

By assuming conditional independence between the outcomes, given the parameters, the posterior distribution can be expressed as:

$$f(Y|\boldsymbol{\theta}, \boldsymbol{X}) \cdot f(\boldsymbol{\theta}) = \prod f(Y|\boldsymbol{\theta}, \boldsymbol{X}) \cdot \prod f(\boldsymbol{\theta})$$

Since the maximization is not affected by using a logarithm approach, the products can be reduced to sums. Thus the posterior distribution can be expressed as:

$$\sum log(f(Y|\boldsymbol{\theta}, \boldsymbol{X})) + \sum log(f(\boldsymbol{\theta}))$$

Since the amount of observations that contains 0 clicks is significantly larger than the amount that contains more than 0, importance sampling is used. The idea is that one observation can represent a larger amount by adding a weight to it. The observations of the maximization problem looks like: $(\boldsymbol{Y}_i, \boldsymbol{X}_i, w_i)$, $i \in 1, ..., N$ where w_i denote the weight from the importance sampling. Each parameter index is denoted $j \in 1, ..., \boldsymbol{\theta}$ for each class, $\star$, of parameters (table 1). Thus the log-posterior density of the statistical model is expressed as:

$$\sum_{i=1}^{N} w_i \cdot log(f(Y_i|\boldsymbol{\theta}, X_i)) + \sum_{\star} \sum_{j=1}^{\boldsymbol{\theta}} log(f(\boldsymbol{\theta}_j^{\star}))$$

2.2.1 The Objective Function

The objective function of the optimization problem, the log-posterior density, is divided into two parts: the data part and the prior part further explained below.

Prior part The second part of the log-posterior density, $\sum_{\star} \sum_{j=1}^{\boldsymbol{\theta}} log(f(\boldsymbol{\theta}_j^{\star}))$, the prior part, where $f(\boldsymbol{\theta}_j^{\star})$ is the normal distribution, thus $log(f(\boldsymbol{\theta}_j^{\star})) = -\frac{log(2\pi)}{2} - \frac{log(\psi^{\star})}{2} - \frac{(\boldsymbol{\theta}_j^{\star})^2}{2\psi^{\star}}$.
This can be further expressed as:

$$log(f(\boldsymbol{\theta}_j^{\star})) = \sum_{\star} \sum_{j=1}^{\boldsymbol{\theta}} -\frac{log(2\pi)}{2} - \frac{log(\psi^{\star})}{2} - \frac{(\boldsymbol{\theta}_j^{\star})^2}{2\psi^{\star}} \tag{3}$$

Data part The first part of the log-posterior density, $\sum_{i=1}^{N} w_i \cdot log(f(Y_i|\boldsymbol{\theta}, X_i))$, the data part, depends on the observations (the data) where $log(f(Y_i|\boldsymbol{\theta}, X_i)) = Y_i \cdot log(\mu) - \mu - log(Y_i!)$ can be expressed as:

$$log(f(Y_i|\boldsymbol{\theta}, X_i)) = \sum_{i=1}^{N} w_i \cdot (Y_i \cdot log(\mu) - \mu - log(Y_i!)) \tag{4}$$

2.2.2 The Gradient

To perform the maximization, the gradient of the log posterior density with respect to $\boldsymbol{\theta}$ is necessary in order to use the L-BFGS algorithm [10].

Prior part

$$\frac{\delta log(f(\boldsymbol{\theta}))}{\delta \theta_j^{\star}} = -\frac{\theta_j^{\star}}{\psi^{\star}} \tag{5}$$

Data part

$$\frac{\delta log(f(Y_i = y_i|\boldsymbol{\theta}, X_i))}{\delta \theta_j^\star} = \frac{\delta log(\mu_i)}{\delta \theta_j^\star} \cdot (y_i - \mu_i) \tag{6}$$

For all simple effects (all but the material-placement matrix factorization effect):

$$\frac{\delta log(\mu_i)}{\delta \theta_j^\star} = \begin{cases} 1 & \text{, if } x_i^\star = j \\ 0 & \text{, otherwise} \end{cases} \tag{7}$$

where $x_i^\star$ is the index of the class $\star$ of the covariate X_i.
For the material-placement matrix factorization effect, (for $k \in 1, ..., |\boldsymbol{\theta}_m^{M:P}|$):

$$\begin{array}{rcl} \frac{\delta log(\mu_i)}{\delta \theta_{mk}^{M:P}} & = & \begin{cases} \theta_{x_i^P k}^{P:M} & \text{, if } x_i^M = m \\ 0 & \text{, otherwise} \end{cases} \\ \frac{\delta log(\mu_i)}{\delta \theta_{pk}^{P:M}} & = & \begin{cases} \theta_{x_i^M k}^{M:P} & \text{, if } x_i^P = p \\ 0 & \text{, otherwise} \end{cases} \end{array} \tag{8}$$

where $x_i^M$is the index of the material class and $x_i^P$is the index of the placement class of the covariate X_i.

3 Implementation

When designing the model, the problem of storing different parameters of different effects in a simple way became obvious. An approach to solving this is the use of a binary tree of effects represented by templates. Two effects were implemented, `IndexedEffects` for simple effects and `IndexedMatrixFactorizationEffects` for the material-placement matrix factorization effect. These objects were instantiated by using templates to represent each effect that should be taken into consideration. Each instantiated effect were added together by ta binary operator class `AddedEffects`.

By using templates, the model could take any number of effects in any independent order. When a method of the model is called, it will pattern match on the `AddedEffects` class, which will iteratively evaluate each effect in the tree.
The parameter values for each effect are stored within each instanced `Effects` class, one dimensional vector for simple effects and two dimensional vectors for the material-placement effect.

3.1 Design

The model should be defined by which effects needed to be taken into consideration. It should also be somewhat expandable for more and different effects not used within the scope of this thesis. To fully satisfy the needs previously described, the implementation utilizes templates to describe which effects the model should affect. This provides a type-safe approach as well as the ability to perform some computations at compile time instead of at run time (also known as templated metaprogramming) [6].

```
1 template <class TEffect>
2 class Model { ... };
```

To add effects to the model, the class `AddedEffects` is used as a binary operator, combining two effects by using partial template specialization [1]. To do this, each effect is represented by a type. By using this strategy, any number of effects might be added in any order.

```
1 template <class TEffect1, class TEffect2>
2 class AddedEffects { ... };
```

Given that there are major differences in evaluating the one dimensional parameters of simple effects compared to evaluating the two dimensional parameters of the matrix factorization effect, two different classes are used: the `IndexedEffect` and the `IndexedMatrixFactorizationEffect`. Both of these classes utilize templates for which kind of effect it should evaluate. The effect is represented by Indexer classes, one for each effect, corresponding to the relevant index given by the covariate.

```
1  class MaterialIndexer
2  {
3  public:
4      inline int GetIndex (const Covariates &cov) const
5      {
6          return cov.GetMaterialId();
7      }
8  };
```

For a syntactically cleaner representation, each effect is represented by its own class to instantiate the correct `IndexedEffect` class with corresponding Indexer class.

```
1  class MaterialEffect : public IndexedEffect<MaterialIndexer>{};
2
3  template<int N>
4  class MaterialPlacementEffect :
5      public IndexedMatrixFactorizationEffect<MaterialIndexer, PlacementIndexer, N>{};
```

The `IndexedEffect` class does all the contributing mathematical computations for each instance of each effect. The parameters for each model are represented by a `ParameterVector` class, one for each effect. This class holds the one and two dimensional arrays representing the parameter values. By using the `ParameterVector` class as an interface, the real data type can be changed without modifying the whole structure of the model. For the matrix factorization effect, two `innerParameterVector` classes are used to represent both matrices in that effect, the templated integer N is the length of the second dimension.

```
1  template<class TIndexer>
2  class IndexedEffect
3  {
4  public:
5      class ParameterVector { ... };
6      ...
7  };
```

```
1  template<class TIndexer1, class TIndexer2, int N>
2  class IndexedMatrixFactorizationEffect
3  {
4  public:
5      class ParameterVector { ... };
6      class innerParameterVector { ... };
7      ...
8  };
```

Two methods were implemented for extracting the `ParameterVector` class from the model given an observation sample. The first one was to get some starting parameters, either with a default value, or randomized. The second method to gain a parameter vector filled with zeroes, for gradient calculation purposes. The most important part of these methods is to make sure that the representing array will be large enough to hold the largest index in the observation sample.

In the implementation of the `Covariate` class, each effect identification value is stored as a private

integer, and is retrieved from get methods. Since the only time the variables of the covariates need to be modified is when created, no get methods are needed.

The `ObservationSample` class was implemented to represent the set of observations. Each observation within the class is represented by a `Covariate` object along with a value for clicks and a value for weight.

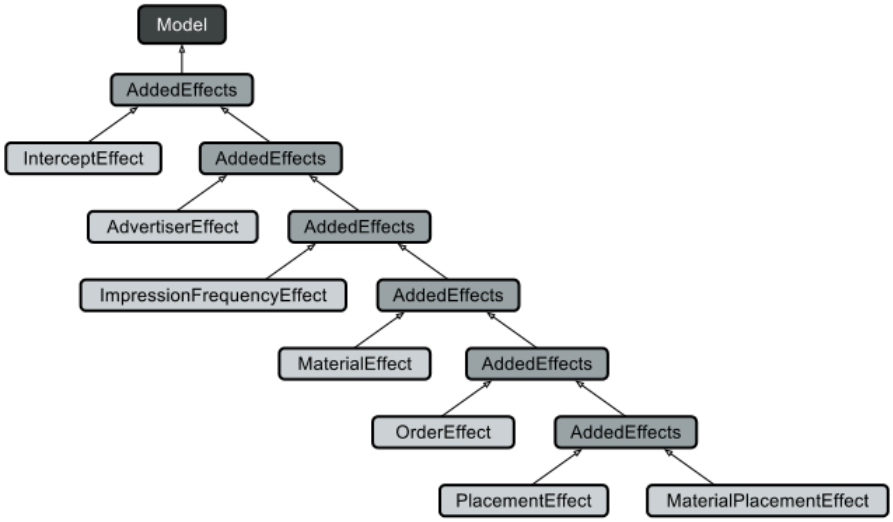

Figure 1: The model and its effects

3.2 CPU implementation

To represent the parameter arrays, `std::vector` has been used instead of a raw heap-allocated array, given that in most cases, the extra overhead incurred is insignificant [1].
In order to fully utilize the `ParameterVector` class in different algorithms, certain arithmetic operators have been overloaded. The operators that have been implemented are multiplication, addition and subtraction, together with their assignment counterpart. Important to notice is that the multiplication operator is actually implemented as two versions, one for the scalar times vector product and one for the dot product. Due to most operations of the `ParameterVector` class involved reading from or writing to every position of the represented array, a general operation method was introduced. This method would loop over all elements in the array, taking different custom operators as arguments, along with single values and/or other parameter vectors.

```
1  template<typename Operator>
2  GeneralVectorOperation(const ParameterVector &parameterVector,
3                         const double &value, Operator f)
4  {
5      for(int i=0;i<size();i++)
6          f(parameterResult[i], parameterVector[i], value);
7  }
```

3.2.1 The Objective Function

Prior part The prior part described in equation 3 under Section 2.2 is completely independent of the observation sample, and only affects the parameter values. Formula (9) loops over all parameter values of all parameter classes. The first sum of all classes is done in the `AddedEffect` class, where the second, of all parameters, is implemented in the `IndexedEffect` and `IndexedMatrixFactorizationEffect` class.

$$\sum_{\star}\sum_{j=1}^{M} -\frac{log(2\pi)}{2} - \frac{log(\psi^{\star})}{2} - \frac{(\boldsymbol{\theta}_j^{\star})^2}{2\psi^{\star}} \tag{9}$$

The sum over all parameter values are utilizing the general operation method introduced in Section 3.2, together with a custom operator representing the inner calculations of the formula. The operator is shown in the program listing below where a is representing the parameter value for a particular index, and c represent the result.

```
1  struct priorfunc
2  {
3      double _psi;
4      priorfunc(double psi) : _psi(psi){};
5      void inline operator() (const double &a, double &c) const
6      {
7          c += -(log(2.0*PI)/2.0) - (log(_psi)/2.0) - (a*a)/(2.0*_psi);
8      }
9  };
```

For performance enhancements, major parts of this formula can be pre-calculated, once for all

parameters ($\frac{log(2\pi)}{2}$), and once for each class ($\frac{log(\psi^\star)}{2}$).

Data part To calculate the data part of the objective function, each observation in the set of observation samples needs to be taken into account. The formula is described in equation (4) under Section 2.2 and can also be seen below (10).

$$\sum_{i=1}^{N} w_i \cdot (Y_i \cdot log(\mu) - e^{log(\mu)} - log(Y_i!)) \tag{10}$$

As described in Section 3.2, each observation in the observation sample contains a covariate, a value for Y (clicks) and a value for w (weight). The formula is implemented as a loop over all observations, where $log(\mu)$ (log of expected number of clicks) is evaluated in the effect classes. To calculate the log of expected number of clicks, referring to the formula (1) in Section 2.2, is done by sum up the contribution from each effects. For the `IndexedEffect` class, the contribution is simply the parameter value given the index provided by the covariate. The contribution from the `IndexedMatrixFactorizationEffect` class is a sum of all N parameter values represented by the first covariate index multiplied with N parameter values represented by the second covariate index.

```
1  auto observation = o;
2  for (int i=0;i<size();i++)
3  {
4      logExpVal = CalculateLogExpectedValue(o[i].getCov(),parameters);
5      weight = o[i].getWeight();
6      y = o[i].getSuccessCount();
7      result += weight * (y * logExpVal - exp(logExpVal) - logFac(y));
8  }
```

The performance can be improved by storing constant results for $log(Y_i!)$, since in the scope of this thesis $Y \in 0, 1, 2, 3$.

3.2.2 The Gradient

The gradient calculation implemented does not return any value, instead it will modify the values of an existing gradient vector. The initial gradient vector is just an instance of the `ParameterVector` class with all elements set to zero. This method, also mentioned in Section 3.2, sets the maximum array size for each array to the maximum identity of the covariates found in the observation sample.

Prior part For the prior part, a lot of similarities can be drawn to the implementation of the prior part of the objective function. The equation (5) found in Section 2.2, is completely independent of the set of observation samples. Instead the formula (11) is applied to all parameter values of the model.

$$\frac{\delta log(f(\boldsymbol{\theta}))}{\delta \theta_j^\star} = -\frac{\theta_j^\star}{\psi^\star} \tag{11}$$

The implementation take advantage of the previously mentioned general operation method (Section 3.2), applied with a custom operator seen implemented below. The operator set the value of `gradient` representing a gradient vector value given an individual index, using the content `parameter` representing a parameter vector value given the same index.

```
1  struct priorgradientcontribution
2  {
3      double _psi;
4      priorgradient(double psi) : _psi(psi){};
5      void inline operator() (double &gradient, const double &parameter)
6      {
7          gradient = -parameter/_psi;
8      }
9  };
```

Data part The calculations regarding the data part of the gradient calculations, equation (6) under Section 2.2, can be seen as two parts. The first part is to gather $(y_i - \mu)$ also known as the prediction error, while the second part is to write the prediction error to the gradient vector. Formula (12) is implemented in the model, looping over all observations in the set of observation samples.

$$\frac{\delta log(f(Y_i = y_i|\boldsymbol{\theta}, X_i))}{\delta \theta_j^\star} = \frac{\delta log(\mu)}{\delta \theta_j^\star} \cdot (y_i - \mu_i) \tag{12}$$

For each observation, the prediction error is first calculated, and used as an argument for the final gradient calculation, done in the effect classes. As seen in equation (7) and (8) under Section 2.2, reintroduced below as (13) and (14), the derivative is calculated differently for simple effects compared to matrix factorization effects.

$$\frac{\delta log(\mu_i)}{\delta \theta_j^\star} = \begin{cases} 1 & \text{, if } x_i^\star = j \\ 0 & \text{, otherwise} \end{cases} \tag{13}$$

$$\begin{aligned} \frac{\delta log(\mu_i)}{\delta \theta^{M:P}_{mk}} &= \begin{cases} \theta^{P:M}_{x_i^P k} & \text{, if } x_i^M = m \\ 0 & \text{, otherwise} \end{cases} \\ \frac{\delta log(\mu_i)}{\delta \theta^{P:M}_{pk}} &= \begin{cases} \theta^{M:P}_{x_i^M k} & \text{, if } x_i^P = p \\ 0 & \text{, otherwise} \end{cases} \end{aligned} \tag{14}$$

For the `IndexedEffect` class, the implementation is simply adding the prediction error to the gradient vector given the covariate index. Given that the gradient vector is instantiated with zeroes, there is no need to loop over all parameters in the effect, only the given indexes of the covariates will be modified.

```
1  void AddGradientContribution(&covariates, predictionError, &parameters, &gradient)
2  {
3      int index = _indexer.GetIndex(covariates);
4      gradient[index] += predictionError;
5  }
```

The implementation in the `IndexedMatrixFactorizationEffect` class loops over N elements of the two indexes provided by the covariates, adding the prediction error multiplied with the parameter value represented by the other index.

```
1  void AddGradientContribution(&covariates, predictionError, &parameters, &gradient)
2  {
3   int index1 = _indexer1.GetIndex(covariates);
4   int index2 = _indexer2.GetIndex(covariates);
5   for (int i=0;i < N; i++)
6   {
7    gradient.getMatrix1(index1)[i] += parameters.getMatrix2(index2)[i]*predictionError;
8    gradient.getMatrix2(index2)[i] += parameters.getMatrix1(index1)[i]*predictionError;
9   }
10 }
```

3.3 GPU implementation

In order to add the GPU co-processor computability, some changes was made to the model. In order be able to copy the data (parameters) from and to the device, the data type of the parameters were changed from `std::vector` to native `double []`. With that change, the `ParameterVector` class was given constructors, assign operators and the quite newly added (C++11) move semantics, allowing the compiler to move ownership of the memory from one object to another, thus avoiding redundant memory allocation and copying [6]. To allow the `ParameterVector` to be copied between the device and the host, set and get methods was added for the pointers of the allocated parameter arrays.

In order to be able to use the model from the GPU, the `__device__` qualifier is required for executing the methods [4]. For convenience, a device manageable version of the model was created with this in mind, only having the necessary methods needed for computing the objective function and gradient. When copying the parameters from the host to the device, memory is allocated on the device where the parameters are transferred to. To reconstruct the `ParameterVector` class on the device, each instance of the parameters is supplied with a pointer to the device memory where the newly copied parameters are stored. When the device has made its gradient calculations, the parameters are transferred back to the host.

3.3.1 Parallel reduction

A common sub problem of the GPU implementation, primarily apparent in the objective function calculation, is how to sum the result provided by each thread without sacrificing the parallel strength. One solution that is applied here is what NVIDIA calls Parallel Reduction, a binary tree-based approach for summarizing the content of an array. By letting each thread represent an index of the array, every other thread will sum up its content with its neighbors, and then synchronize between all threads before repeating the procedure. Mark Harris presents seven different versions of this method in the presentation Optimizing Parallel Reduction in CUDA [11] each with an incremental optimization. Because it uses a tree-like structure, 2^D threads are required for a complete binary tree, where the number of elements $N \leq 2^D$.

3.3.2 Observation sample

To improve the memory access, the technique described in appendix A.3.2 are applied to the `ObservationSample` class, redesigning it into a structure of arrays. This design allows the SIMT architecture (see appendix A.2.1) of the multiprocessor to access multiple variables from the global memory in a single transaction, making full use of the memory bandwidth. This is implemented by removing the `Covariate` class and instead stores its content as separate arrays in a struct. When all threads of a warp are accessing an element in the struct, the aligned elements of that array can be transferred in up to 128-byte transactions [12].

```
1  struct ObservationSampleSoA{
2
3      struct Covariate{
4          int _materialArray[N];
5          int _placementArray[N];
6          int _orderArray[N];
7          int _impressionfrequencyArray[N];
8          int _advertiserArray[N];
9      }
10
11     char _successCount[N];
12     short _weight[N];
13     int _size;
14 };
15
16 ObservationSampleSoA obs;
```

In order of improving L1 and L2 cache hits when accessing the parameters represented by the covariates of the observation sample, sorting the observations could potentially improve the bandwidth (see appendix A.2.6). The sorting is done on the CPU and in regard to the most frequently used indexes.

3.3.3 The Objective Function

When summarizing the result from calculating the prior part and the data part of the objective function, two distinct versions of the parallel reduction technique (Section 3.3.1) was considered. The first method was to let each thread in a thread block store its contribution in the shared memory. When all threads have done their calculations, they will synchronize and start reducing the result. The advantages with this approach is that it require less global memory and less global memory access, due to each thread block only storing their summarized result once. The drawbacks are that the number of threads that are required for running the calculations need to be 2^D, making occupancy optimizations more difficult.

The second method was to let each thread store its result in global memory and separating the reduction from the actual computations. This would result in one extra kernel launch (see appendix A.1.1) but the requirement of 2^D threads are no longer needed. Of course the disadvantage of this approach is the need for more global memory and more global memory access, as well as the extra overhead from launching an extra kernel. The results from both of these methods still require at least one more summarization, supposing more than one block was launched.

Since the second method requires a lot more global memory than the first one, and one of the

limitations with this thesis is that all observations need to be able to fit in the global memory, the first method was chosen to be implemented.

```
1  __global__ void Reduction(double *temp_result)
2  {
3      double *sdata = SharedMemory();
4      unsigned int tid = threadIdx.x;
5      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
6      sdata[threadIdx.x] = temp_result[i];
7      __syncthreads();
8      for(int s = blockDim.x/2; s>0; s/=2)
9      {
10         if(tid < s)
11         {
12             sdata[tid] += sdata[tid + s];
13         }
14         __syncthreads();
15     }
16     if (tid == 0)
17         temp_result[blockIdx.x] = sdata[0];
18 }
```

Prior part The parallelization of the prior part is achieved by removing the general operation method introduced in Section 3.2 and instead letting each thread ID represent the parameter index. Every thread will calculate the contribution from the index for each effect, summarize it and store it in shared memory. Each thread is performing the calculation, $-\frac{log(2\pi)}{2} - \frac{log(\psi)}{2} - \frac{(\boldsymbol{\theta})^2}{2\psi}$, for each parameter value represented by each effect. As mentioned as a performance enhancement in Section 3.2.1, the advantages with pre-calculation of certain parts has been implemented here. Only two variables are used in the calculations, ψ and the parameter value $\boldsymbol{\theta}$, in which ψ at most differ once per effect and can therefore be stored in the fast access constant memory (see Section A.2.5). By reducing the constant values of these calculations to $c_1 = -\frac{log(2\pi)}{2}$, $c_2 = -\frac{log(\psi)}{2}$, and $c_3 = -\frac{1}{2\psi}$, the arithmetic operations performed for each parameter value are $c_1 + c_2 + \boldsymbol{\theta}^2 \cdot c_3$.

```
1  __device__ inline double PriorFunction() const
2  {
3      auto value = __values[threadIdx.x];
4      return (c1 + c2 - value*value * c3);
5  }
```

Data part In the implementation of the data part of the objective function, each observation is represented by a thread. No major changes were required for the effect classes given that calculating $log(\mu)$ is sequential performed by each thread, instead the for-loop of the model where all the observations were iterated through has been removed.

```
1  __device__ const inline double CalcDataPart(observationSample, parameters)
2  {
3      index = threadId.x;
4      logExpVal = CalcLogExpVal(observationSample.getCov(), parameters);
5      y = observationSample.getSuccessCount(index);
6      weight = observationSample.getWeight(index);
7      return  (w * (y * logExpVal - exp(logExpVal)) - logFactorial(y));
8  }
```

When the expected number of clicks is evaluated, the rest of the formula $w \cdot (Y \cdot log(\mu) - e^{log(\mu)} - log(Y!))$ is calculated. As mentioned in data part of Section 3.2.1, the performance can be increased by pre-calculating the values for $log(Y!)$ by the fact that $Y \in 0, 1, 2, 3$. This was implemented by using a switch method matching on the value of Y, return constant results for any of the eligible values.

3.3.4 The Gradient

The gradient is calculated in the same way as the CPU implementation in the sense that an instance of the `ParameterVector` class filled with zeroes is modified in the positions given by the covariates. The class is created and filled with zeroes on the CPU, and then the arrays representing the parameters are copied to the GPU, where the calculations are done.

Prior part The implementation of the prior part of the gradient calculation is accomplished by removing the general operator method described in Section 3.2 as well as the custom operator performing the calculation. Instead, each parameter index is represented by a thread ID and one calculation is performed sequentially per effect. The gradient calculation, $\frac{\delta log(f(\boldsymbol{\theta}))}{\delta \theta} = -\frac{\theta}{\psi}$, given that $c = -\frac{1}{\psi}$ can be pre-calculated and stored in the fast access constant memory, reduced to $\theta' = \theta \cdot c$.

```
1  __device__ inline void PriorGradient(parameters)
2  {
3      gradient[threadId.x] = parameters[threadId.x] * psi_minus_div;
4  }
```

Data part - naïve version Just like in the CPU implementation (Section 3.2), the data part is divided into two major blocks, calculating the prediction error, and writing it to the gradient vector. Each thread represents one observation in the set of observation samples. The prediction error, $y - e^{log(\mu)}$, is calculated once per thread, where the $log(\mu)$ contribution is using the same implementation described in the data part of Section 3.3.3. The second part, adding the prediction error to the gradient vector, is very similarly implemented as the CPU version. The difference is that since indexes could be the same, writing to the same location in global memory leads to undefined results. To counter this, the use of atomic addition has been applied in the form of the `atomicAdd` method, guaranteeing that both results will be correctly added to the memory location.

```
1  __device__ const inline void CalcDataGradPart(observationSample, parameters, gradient)
2  {
3      cov = observationSample.getCov();
4      expVal = exp(CalcLogExpVal(cov), parameters));
5      y = observationSample.getSuccessCount(threadId.x);
6      weight = observationSample.getWeight(threadId.x)
7      predictionError = (y - expVal)*weight;
8      _effect.AddGradientContribution(cov,predictionError,parameters,gradient);
9  }
```

```
1  __device__ inline void AddGradientContribution(&cov, predictionError,
2                                                 &parameters, &gradient)
3  {
4      atomicAdd(&parameterGradient[_indexer.GetIndex(cov)],predictionError);
5  }
```

The major setback with this implementation is that the intercept effect (θ^{lI}) is implemented as a vector of only one element. This results in that every thread must write atomically to the same global memory location, essentially performing the complete implementation sequential. To adjust this, the fact that the intercept effect basically is a sum of all prediction errors can be exploited. By removing the intercept from the model and storing the prediction error in shared memory, the intercept effect can fully utilize the reduction technique described in Section 3.3.1. The advantage of this implementation is the elimination of atomic operations when calculating the gradient of the intercept effect, increasing performance and parallelism. The drawbacks however is that the cost of using the parallel reduction is applied, as well as the cost of launching sequential kernels when dealing with a large amount of threads.

```
1  __device__ const inline void CalcDataGradPart(observationSample, parameters, gradient)
2  {
3      cov = observationSample.getCov();
4      expVal = exp(CalcLogExpVal(cov), parameters));
5      y = observationSample.getSuccessCount(threadId.x);
6      weight = observationSample.getWeight(threadId.x)
7      sharedMemory[threadIdx.x] = (y - expVal)*weight;
8      __syncthreads();
9      _effect.AddGradientContribution(cov,parameters,gradient);
10 }
```

Data part - improved version To further reduce the amount of atomic operations. A strategy is to sum up all the parameters of a block in the shared memory, before writing it to the global memory. An approach to this is to let each thread of a block represent one index for each effect. Then loop over the block size, and sum up the contribution each time that index is found. When done, the result is written atomically to the global memory. This method allows for a minimum of 25 atomic operations per block instead of a static 25 per thread, but at the cost of sequentially looping over each effect.

```
1  __device__ inline void AddGradientContribution(&cov, predictionError,
2                                                  &parameters, &gradient)
3  {
4    __shared__ indexes[blockDim];
5    index = indexes[threadIdx.x]
6    result = 0.0;
7    mem = 0;
8    first = 0;
9    for(i < blockDim.x)
10   {
11     if(indexes[i] == index)
12     {
13          if(mem == 0)
14          {
15          mem++;
16          first = i;
17          }
18          result += predictionError[i];
19     }
20   }
21   if(first == threadIdx.x)
22     atomicAdd(&parameterGradient[index],result)
23 }
```

4 Results and Conclusion

A set of benchmarks have been used to compare the GPU implementation to the CPU implementation as well as the impacts of the optimizations and the changes described in Section 3.3.

The system configuration used to execute the implementations is presented in table 2.

CPU	Intel Core i7-3930K
RAM	8x4GB DDR3 1600 MHz
Hard Drive	Corsair Force 3 SSD 120GB
Operating System	Microsoft Windows 7 Professional 64-bit
IDE	Microsoft Visual Studio 2010 v 10.0.30319.1
GPU	NVIDIA GeForce GTX 570 1280 MB GDDR5
GPU Driver	NVIDIA GeForce Driver v296.10
GPU SDK	NVIDIA CUDA Toolkit v4.1 64-bit

Table 2: The system used for evaluating the performance of the implementations

To capture the time taken for the CPU implementation to execute, the `clock()` method found in the `<ctime>` library is used to return the number of clock ticks elapsed since the program was launched [13]. To measure the time, two calls to the `clock()` method are done, one before the execution and one after. The resulting processing time for the implementation is the difference between these calls. When measuring the GPU implementation, execution time from NVIDIA NSight as well as the CUDA events is used [4]. With future intentions to execute the complete L-BFGS algorithm on the GPU, storing all the data in global memory, the time captured will only be measured for the execution of the computation and not the time taken to transfer the parameters or the observations to and from the global memory. Each benchmark was executed ten times and an average was calculated. The benchmark will contain the time for the execution (ms) as well as number of observations per millisecond (obs/ms).

4.1 The Test Data

Two sets of test data are used to evaluate the performance of the implementations. One with real test data used by Admeta, and one simulated. The simulated data is created from randomizing covariates uniformly, followed by randomizing parameter values according to prior distribution [3]. For each covariate, μ is calculated from the parameter values. Since the distribution for clicks is completely defined given μ, this is used to simulate the number of clicks for each covariate. To provide the same test suit for different implementations, the random value will be generated from a static seed, providing the same result for each test. The major difference between the two sets of test data is the fact that in the real data supplied by Admeta, the covariate indexes are not uniformly distributed. For example, only a limited amount of material and placement combinations exist.

Test Data	Observations	Parameter Indexes	Active Parameter Indexes
Real Data	13 619 739	1 482 799	32 587
Simulated Data	33 554 432	125 000	125 000

Table 3: The configurations of the test data

In the scope of this thesis and previously described, the number of observations should be able to fit in the device memory due to the future intention of executing the complete L-BFGS algorithm on the GPU. The graphics card that is used in this test (table 2) has 1280 MB device memory. To fully utilize the implementation, a set of 2^N observations is used for the simulated data to reduce branch divergence [4]. Denoting s as the size of an observation in bytes, d as the device memory in Megabytes (MB) the following formula can be used to calculate the maximum size of the observation sample.

$$\lfloor \log_2(\frac{2^{20}}{s} \cdot d) \rfloor$$

With an observation size of 23 bytes, $2^{25} = 33554432$ observations can be used at the size of 734 MB. Important to notice is that at least two sets of parameters also needs to fit in the device memory. The size of the parameters is however significantly smaller than that of the set of observations. When using double precision floating units (8 byte) one million parameters only have the size of $\frac{8 \cdot 10^6}{2^{20}} = 7.63$MB.

The real data contains several parameter values that are no longer used, for example materials that have been removed, and don't have an index in the covariates. The negative impact with this is that the parameter values that are no longer active still needs to be allocated and evaluated, wasting computation and memory. However, the simulated data only contains active parameter values.

The model tested contains all the implemented effects holding the ψ value of 0.05 (see eq 2 in Section 2.2).

4.2 The CPU implementation

The CPU implementation of the statistical model (Section 3.2) is used as a reference to measure the possible speedups provided by the GPU implementation. The implementation is quite naïve and there is without doubt possibilities of improvement. CPU optimization is however outside of the scope of this thesis. The benchmark is presented in table 4.

Test Data	Floating Point Format	Objective Function		Gradient	
		(ms)	(obs/ms)	(ms)	(obs/ms)
Real Data	Double Precision	478	28 493	999	13 633
	Single Precision	384	35 468	840	16 213
Simulated Data	Double Precision	1 584	21 183	2 676	12 539
	Single Precision	1 322	25 382	2 283	14 698

Table 4: The result of the benchmark running the model on the CPU

4.3 The GPU implementation of the objective function

4.3.1 Occupancy

The first step in running the GPU implementation of the objective function is to decide the block and grid size. To maximize the occupancy mentioned in appendix A.3.1, the NVIDIA Occupancy Calculator [14] was used to find the most effective configuration given the number of registers and the number of threads per block (the size of the shared memory is also a matter, but not a limitation in the objective function implementations). The number of registers currently in use was extracted by using NVIDIA's Visual Profiler and NSight. Important to mention is that there is no guarantee that higher occupancy results in higher performance, specifically if there is no latency or bandwidth limitation [14].

Sum Reduction The sum reduction kernel is used by both GPU implementations to sum up vectors containing double or single precision elements. The implementation (see Section 3.3.1) sums up all elements in a block and stores it in global memory given a block size of 2^N. From profiling, using single precision results in 8 registers compared to 10 for double precision. Since the register usage is well under the limit of 20 (See appendix A.2.5) 100% occupancy can be achieved. The block sizes resulting in highest theoretical occupancy, calculated using NVIDIA Occupancy Calculator [14], is using 256 or 512 threads per block. However, given the nature of the reduction technique, this does not scale very well with grid sizes of one. This is due to the fact that half of the threads halts for every iteration, and when there is only one block for the warp scheduler to handle, some Streaming Multiprocessors will idle (see appendix A.3.1). To be able to assign an appropriate grid and block size to the sum reduction kernel, a benchmark was done to evaluate the execution time for a set of applicable sizes. The test involves summing up an array of double precision values (with the same size as the number of threads) measured with NVIDIA NSight profiling. Table 5 includes the grid

configurations with fastest execution speed given the array size.

Array Size	Dimensions	Execution Time (ms)
2^{20}	1024×1024	0.3300
2^{19}	1024×512	0.1230
2^{18}	1024×256	0.0590
2^{17}	512×256	0.0330
2^{16}	256×256	0.0200
2^{15}	256×128	0.0140
2^{14}	64×256	0.0101
2^{13}	64×128	0.0078
2^{12}	64×64	0.0071
2^{11}	64×32	0.0068
2^{10}	1×1024	0.0065
2^{9}	1×512	0.0044
2^{8}	1×256	0.0040
2^{7}	1×128	0.0037
2^{6}	1×64	0.0034
2^{5}	1×32	0.0033

Table 5: The fastest grid dimensions given the array size and the sum reduction test data

Prior Part The block (and grid size) of the prior part is calculated from the number of threads that needs to be executed, namely the number of parameter values, at least 125000 for the simulated data, and at least 1482799 for the real data (see table 3). By the design of the implementation, the number of threads to be launched must also be a power of two greater or equal to the number of parameter values, resulting in 2^{17} for simulated data and 2^{21} for the real data. The implementation is designed with support of one dimensional block sizes and two dimensional grid sizes where the calculations are first done on the block followed by a kernel call for each grid dimension to be reduced (see Section 3.3.1 and Section 3.3.3). Using the Occupancy Calculator [14] with double precision, the power of two configurations for the block size that gave highest theoretical occupancy (83%) was using 256 threads per block. The benchmark used a CUDA event to record the time of the execution of the calculation followed by the reduction. Since only the prior part is tested in this Section, the grid and block size of the data part is fixed.

The result for testing different grid and block sizes for the prior part is presented in table 6. By using the following formula, 2^{M-B}, where 2^M is the number of parameter values and 2^B is the desired block size of the prior kernel. The grid size is determined by looking up the result from the formula in table 5 .

Data Part The data part of the objective function need to launch more threads than the prior part due to one thread representing one observation. The simulated data was choose to be a power of two

Test Data	Grid Size	Block Size	Occupancy	Time (ms)
Real Data	64×256	128	67%	**0.66**
	64×128	256	83%	0.75
	64×64	512	67%	0.79
	64×32	1024	67%	0.92
Simulated Data	1×1024	128	67%	0.28
	1×512	256	83%	0.29
	1×256	512	67%	**0.26**
	1×128	1024	67%	0.29

Table 6: The benchmark of the prior part of the objective function

thus requires 2^{25} threads to launch. The real data requires 2^N threads, where $N = \lceil \log_2(13619739) \rceil$, resulting in 2^{24}threads. The number of registers used by the data part resulted in 26, when using NVIDIA's Visual Profiler and NSight. When using the NVIDIA Occupancy calculator [14] for all power of two block sizes results in the same theoretical occupancy of 67% , due to the high register usage. The result of the benchmark with different block sizes is presented in table 7, the grid size is determined the same way as in the prior part.

Test Data	Grid Size	Block Size	Occupancy	Time ms	Throughput obs/s
Real Data	512×256	128	67%	70	193 875
	256×256	256	67%	67	204 562
	256×128	512	67%	**61**	**223 751**
	64×256	1024	67%	62	219 284
Simulated Data	1024×256	128	67%	409	81 968
	512×256	256	67%	405	82 769
	256×256	512	67%	401	83 621
	256×128	1024	67%	**400**	**83 957**

Table 7: The benchmark of the data part of the objective function

Conclusion The result shows that there is no huge difference in the execution time in either the prior part or the data part from changing the block size. The occupancy is not represented by the performance for these implementations mainly due to the fact that there is no major latency or bandwidth limitation. As with any optimization, experimentation is advised, and should be tested continuously when adding new optimizations [14].

4.3.2 Cache Optimization

Accessing the parameter values from global memory provides latency up to $400 - 800$ clock cycles (see appendix A.3.1), this has a large negative impact on the overall throughput of the implementation. To improve upon this, utilizing the L1 and L2 cache described in appendix A.2.6 can reduce the need for global memory access. When L1 caching is activated all global memory access are cached [2], to maximize the usage of this, parameter values accessed frequently should be stored close to each other. The design of the access of the parameter values depends on the indexes represented by the covariates, thus, sorting the covariate indexes on the CPU before calling the GPU implementation should improve the L1 cache hits. The sorting is with respect to the material index followed by the placement index since they are most frequently used given that they also appear in the matrix factorization effect. The test is done with the fastest grid and block configurations from Section 4.3.1.

The results presented in table 8 involve using activated and disabled L1 cache as well as shuffled and sorted covariates. The cache hit is measured using NVIDIA NSight.

Test Data	L1 Cache	Covariates	L1 Cache Hit	L2 Cache Hit	Time (ms)	Throughput (obs/ms)
Real Data	Disabled	Shuffled	0%	96%	70	194 568
$(256 \times 128) \times 512$	Disabled	Sorted	0%	70%	16	851 234
	Enabled	Shuffled	58%	98%	62	219 673
	Enabled	Sorted	88%	17%	**12**	**1 134 978**
Simulated Data	Disabled	Shuffled	0%	70%	289	116 105
$(256 \times 256) \times 512$	Disabled	Sorted	0%	88%	**83**	**404 270**
	Enabled	Shuffled	9%	66%	430	78 034
	Enabled	Sorted	27%	95%	111	302 292

Table 8: The benchmark of the cache optimization of the objective function

Conclusion As presented in appendix A.3.2 using L1 cache when having scattered memory locations can reduced the performance. This can be seen in the benchmark of the simulated data where the execution time increased with 50% by activating the L1 cache. However, in the real data test, that is not the case. The execution time is actually improved by using the cache, but as presented, the cache hit is also significantly better in the real data versus the simulated data. When benchmarking with sorted data, the use of L1 cache improved the execution time for the real data, but not for the simulated data. The conclusion that can be drawn from the benchmark is that L1 cache should be disabled when cache hits are low (lower than 58% from this benchmark).

4.3.3 Single Precision

The model has been designed with double precision floating point units to represent parameter values. Loosing accuracy, single precision floating point would provide higher throughput given that, compared to the double precision, uses half the space as well as arithmetic instructions uses less clock cycles (see appendix A.3.3). Using single precision instruction also provide the use of dual dispatch in the warp scheduler (see appendix A.2.4) as well as less register usage (see appendix A.2.5). To compare the double and single precision floating unit, a modified version of the model is implemented utilizing single precision floating point functions described in [2, 4].

The results presented in table 9 only compares the time of the execution, not the accuracy of the result.

Test Data	L1 Cache	Covariates	Single precision		Double precision	
			Time (ms)	Throughput (obs/ms)	Time (ms)	Throughput (obs/ms)
Real Data	Disabled	Shuffled	49	277 954	70	194 568
$(256 \times 128) \times 512$	Disabled	Sorted	9	1 513 304	16	851 234
	Enabled	Shuffled	51	267 053	62	219 673
	Enabled	Sorted	**7**	**1 945 677**	**12**	**1 134 978**
Simulated Data	Disabled	Shuffled	185	181 375	289	116 105
$(256 \times 256) \times 512$	Disabled	Sorted	**50**	**671 089**	**83**	**404 270**
	Enabled	Shuffled	373	89 958	430	78 034
	Enabled	Sorted	62	541 200	111	302 292

Table 9: The benchmark of the objective function using single precision

Conclusion While the result will not be as accurate as when double precision is used, the benchmark shows that execution time is reduced up to around a factor of 2 by switching to single precision floating point units.

4.4 The GPU implementation of the gradient

4.4.1 Occupancy

Just as in Section 4.3.1, NVIDIA Occupancy Calculator [14] was used to find the most effective grid and block size configuration of the gradient.

Prior Part The number of threads that needs to be executed for the prior part of the gradient is the same as for the prior part of the objective function (Section 4.3.1), 2^{17} for the simulated data and 2^{21} for the real data. NVIDIA Visual Profiler and NSight show that the prior part only uses 12 registers per thread. The result is that block sizes of 256 and 512 provide occupancy of 100% [14]. Each configuration is tested together with the corresponding grid size calculated from 2^{M-B}, where 2^M is the number of parameter values and 2^B is the desired block size of the prior kernel. The grid size is determined by looking up the result from the formula in table 5 . The result from the benchmark is presented in table 10.

Test Data	Grid Size	Block Size	Occupancy	Time (ms)
Real Data	64×256	128	67%	0.299
	64×128	256	100%	0.306
	64×64	512	100%	**0.297**
	64×32	1024	67%	0.319
Simulated Data	1×1024	128	67%	0.070
	1×512	256	100%	0.067
	1×256	512	100%	0.074
	1×128	1024	67%	**0.066**

Table 10: The benchmark of the prior part of the gradient

Data Part As presented in Section 3.3.4, two versions of the data part of the GPU implementation have been implemented. The number of threads required for the implementations is just as many as the data part of the objective function, namely 2^{25} for the simulated test data and 2^{24} for the real test data. The naïve implementation uses 35 registers per thread while the improved version uses as many as 61. Since the improved version uses

```
blockSize · (nrOfSimpleEffects-1 · sizeof(unsigned int) + sizeof(double))
```

bytes of shared data per block, the occupancy will be significantly lower. Just as in the prior part, the grid size is taken from table 5. The results from the naïve implementation is presented in table 11 while the results from the improved implementation is presented in table 12.

Test Data	Grid Size	Block Size	Occupancy	Time (ms)	Throughput (obs/ms)
Real Data	512×256	128	58%	31 848	428
	256×256	256	50%	32 627	417
	256×128	512	33%	**24 082**	**566**
Simulated Data	1024×256	128	58%	2 349	14 285
	512×256	256	50%	**2 309**	**14 532**
	256×256	512	33%	2 150	15 607

Table 11: The benchmark of the data part of the naïve gradient implementation

Test Data	Grid Size	Block Size	Occupancy	Time (ms)	Throughput (obs/ms)
Real Data	1024×512	32	17%	2 434	5 596
	1024×256	64	33%	**1 797**	**7 579**
	512×256	128	33%	2 147	6 344
	256×256	256	33%	3 606	3 777
	256×128	512	33%	6 713	2 029
Simulated Data	1024×1024	32	17%	**3 109**	**10 793**
	1024×512	64	33%	4 596	4 596
	1024×256	128	33%	4 410	7 609
	512×256	256	33%	3 938	8 521
	256×256	512	33%	4 501	7 455

Table 12: The benchmark of the data part of the improved gradient implementation

Conclusion As presented, the occupancy of either version does not seem to have any correlation with the execution time. Just as mentioned in the conclusion part of Section 4.3.1, when no major latency or bandwidth limitation is present, occupancy has little effect. The results also hint of the distribution of the data between the real and the simulated tests. The naïve implementation is limited by the atomic additions, if several parameter values need to be written at the same time, the global memory writes will be serialized.

4.4.2 Cache Optimization

As described in appendix A.3.1 accessing parameter values from global memory is an expensive operation given the number of clock cycles required. Just like in Section 4.3.2, the two versions of the gradient implementation is tested with sorted and shuffled covariates, along with activated and disabled L1 cache. The result of the naïve implementation of the gradient is presented in table 13. For the improved gradient implementation, results are presented in table 14. The benchmarks have been done with the fastest grid and block configurations from table 11 and 12. The improved version is tested with block size of both 32 and 64 for real data.

Test Data	L1 Cache	Covariates	L1 Cache Hit	L2 Cache Hit	Time (ms)	Throughput (obs/ms)
Real Data	Disabled	Shuffled	0%	30%	**23 908**	**570**
$(256 \times 128) \times 512$	Disabled	Sorted	0%	10%	160 966	85
	Enabled	Shuffled	49%	41%	24 098	565
	Enabled	Sorted	76%	11%	161 424	84
Simulated Data	Disabled	Shuffled	0%	18%	**2 196**	**15 280**
$(512 \times 256) \times 256$	Disabled	Sorted	0%	10%	104 694	321
	Enabled	Shuffled	7%	31%	2 309	14 532
	Enabled	Sorted	33%	14%	104 421	321

Table 13: The benchmark of the cache optimization of the naïve gradient implementation

Test Data	L1 Cache	Covariates	L1 Cache Hit	L2 Cache Hit	Time (ms)	Throughput (obs/ms)
Real Data	Disabled	Shuffled	0%	44%	1 661	8 200
$(1024 \times 256) \times 64$	Disabled	Sorted	0%	20%	1 166	11 681
	Enabled	Shuffled	42%	56%	1 814	7 508
	Enabled	Sorted	49%	6%	**1 085**	**12 553**
Real Data	Disabled	Shuffled	0%	57%	2 339	5 823
$(1024 \times 512) \times 32$	Disabled	Sorted	0%	25%	858	15 874
	Enabled	Shuffled	60%	63%	2 435	5 593
	Enabled	Sorted	58%	13%	**519**	**26 242**
Simulated Data	Disabled	Shuffled	0%	39%	3 370	9 957
$(1024 \times 1024) \times 32$	Disabled	Sorted	0%	41%	2 419	13 871
	Enabled	Shuffled	25%	44%	3 110	10 789
	Enabled	Sorted	45%	42%	**1 716**	**19 554**

Table 14: The benchmark of the cache optimization of the improved gradient implementation

Conclusion As seen in table 13, sorting the covariates have significantly negative effect on the execution time. The reason is that when the indexes are sorted, the same location in global memory needs to be written to atomically, serializing the process. Using the L1 cache has little to no effect on the execution time for the naïve implementation. The improved version however benefits from the coalesced memory reads provided by sorting the covariates.

4.4.3 Single Precision

The test of replacing double precision to single precision is performed the same way as in Section 4.3.3. The results from the naïve implementation is presented in table 15 and the improved implementation is presented in table 16, only time is measured, not accuracy.

Test Data	L1 Cache	Covariates	Single precision		Double precision	
			Time (ms)	Throughput (obs/ms)	Time (ms)	Throughput (obs/ms)
Real Data	Disabled	Shuffled	1315	10 357	**23 908**	**570**
$(256 \times 128) \times 512$	Disabled	Sorted	**994**	**13 702**	160 966	85
	Enabled	Shuffled	1342	10 149	24 098	565
	Enabled	Sorted	**994**	**13 702**	161 424	84
Simulated Data	Disabled	Shuffled	1441	23 286	**2 196**	**15 280**
$(512 \times 256) \times 256$	Disabled	Sorted	2008	16 710	104 694	321
	Enabled	Shuffled	**1337**	**25 097**	2 309	14 532
	Enabled	Sorted	2002	16 760	104 421	321

Table 15: The benchmark of the naïve gradient implementation using single precision

Test Data	L1 Cache	Covariates	Single precision		Double precision	
			Time (ms)	Throughput (obs/ms)	Time (ms)	Throughput (obs/ms)
Real Data $(1024 \times 512) \times 32$	Disabled	Shuffled	854	15 948	2 339	5 823
	Disabled	Sorted	301	45 248	858	15 874
	Enabled	Shuffled	860	15 837	2 435	5 593
	Enabled	Sorted	**199**	**68 441**	**519**	**26 242**
Real Data $(1024 \times 256) \times 64$	Disabled	Shuffled	620	21 967	1661	8 200
	Disabled	Sorted	360	37 833	1166	11 681
	Enabled	Shuffled	576	23 645	1814	7 508
	Enabled	Sorted	**228**	**59 736**	**1085**	**12 553**
Simulated Data $(1024 \times 1024) \times 32$	Disabled	Shuffled	1 910	17 568	3 370	9 957
	Disabled	Sorted	809	41 476	2 419	13 871
	Enabled	Shuffled	1586	21 157	3 110	10 789
	Enabled	Sorted	**670**	**50 081**	**1 716**	**19 554**
Simulated Data $(1024 \times 512) \times 64$	Disabled	Shuffled	1560	21 509	4264	7 869
	Disabled	Sorted	657	51 072	3206	10 466
	Enabled	Shuffled	1596	21 024	4612	7 275
	Enabled	Sorted	**577**	**58 153**	**3039**	**11 041**

Table 16: The benchmark of the improved gradient implementation using single precision

Conclusion The use of single precision instead of double precision floating point unit had a significant effect of the naïve implementation. One explanation to this could be that since there is no built in atomic addition method for double precision floating point units, a special implementation was required to split the data into two parts that needs to be written one after another locking up resources. For floating point units however, atomic operations are supplied in the CUDA language. Using single precision also had a positive effect on the improved implementation.

4.5 Summary of results

The results show that in all cases, using single precision over double precision reduces the execution time. The same applies to sorting the covariates before computations. Having the L1 cache enabled for sorted data seems to improve the performance for the gradient. However, the objective function seems to be better off having it disabled. The figures below shows the speedup of the GPU implementation compared to the CPU implementation plotted on a logarithmic scale, where 1 is the speed of the reference CPU implementation. Figure 2 shows the speedup of the real data while Figure 3 shows the speedup of the simulated data.

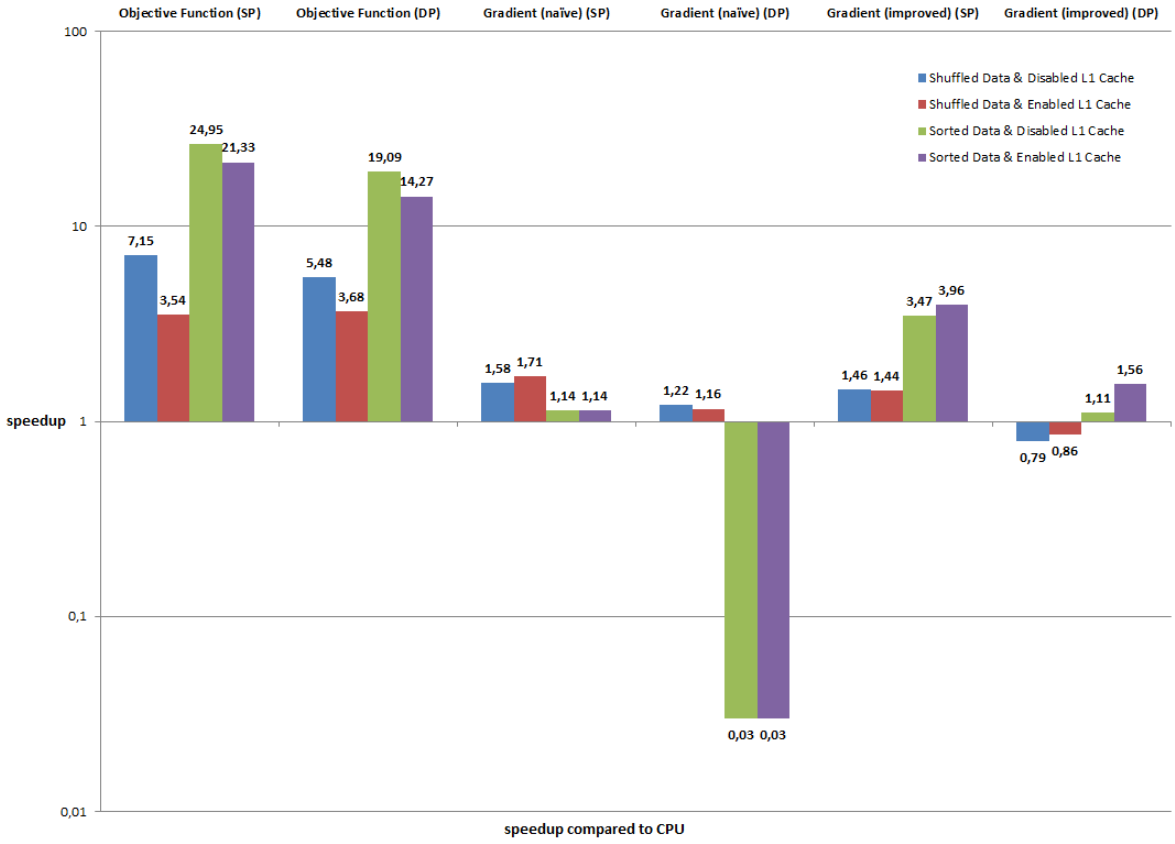

Figure 2: Graph of Real Data

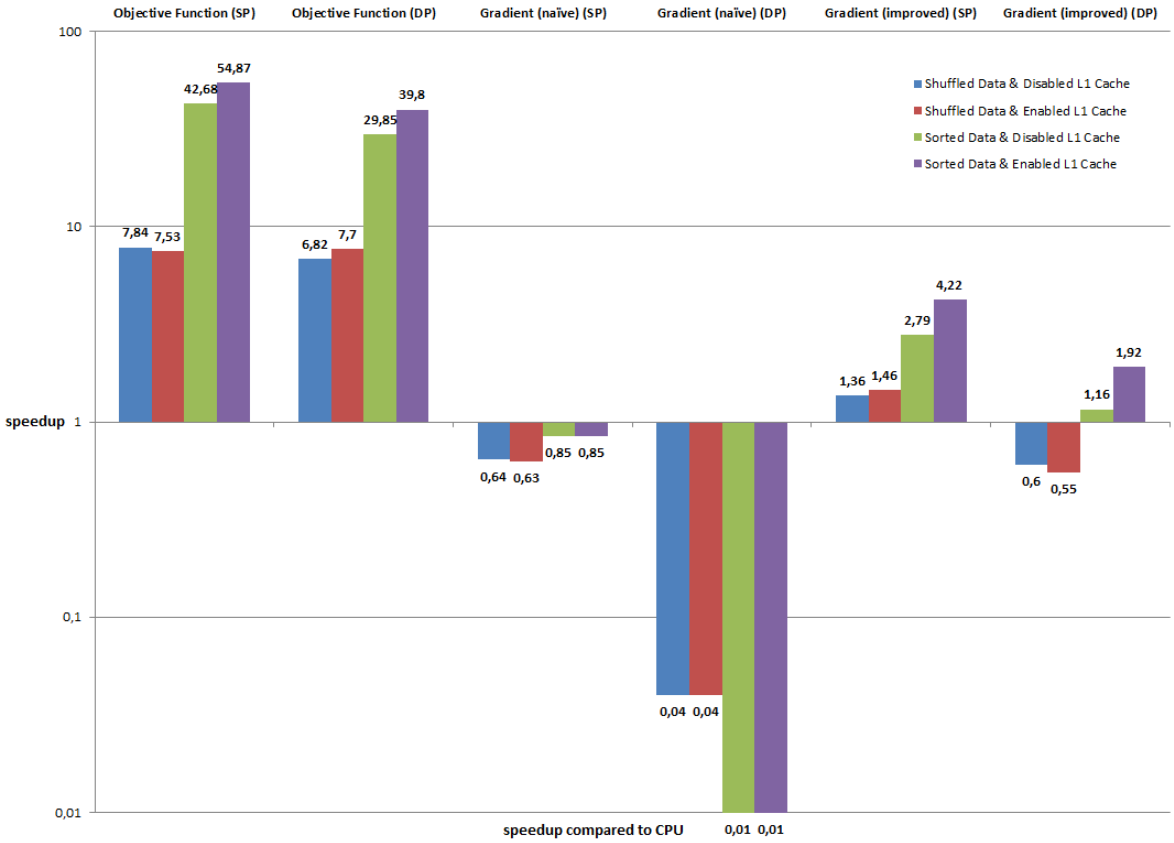

Figure 3: Graph of Simulated Data

To get a sense of the combined results from both the objective function and the gradient, Fig 4 shows the speedup of using both the implementations iteratively compared to the CPU implementation.

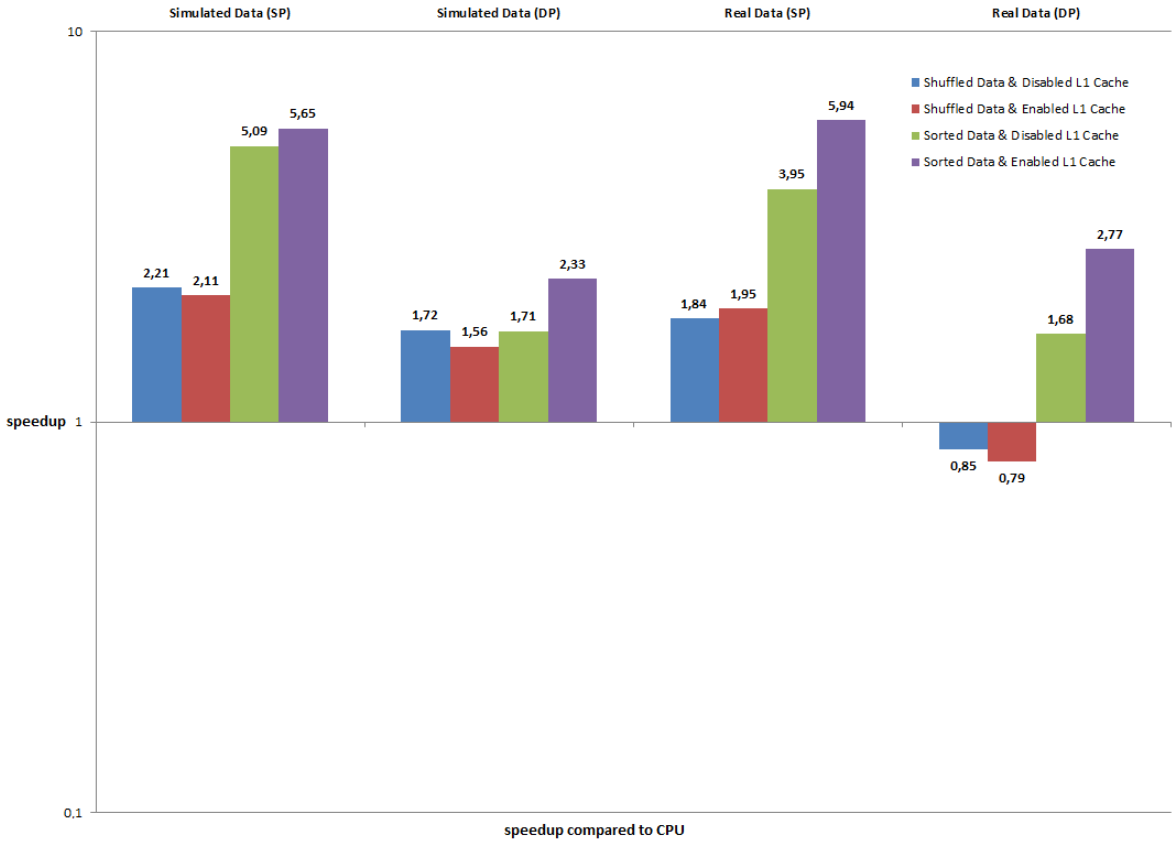

Figure 4: Graph of Combined Results

4.6 Conclusion

In this thesis, an efficient implementation of the corresponding log posterior density and gradient for a non-linear statistical model with a large data set has been presented. With the intention of speeding up the execution time, the implementation was done using NVIDIA's computing architecture CUDA, taking advantage of the massively parallel computation power of a GPU. A sequential implementation was also done for the CPU to use as a reference to measure the difference in run time.

To optimize the GPU implementation, three different techniques was applied. The first one, utilization optimization (see appendix A.3.1), was applied by testing different grid and block sizes in order to maximize the efficiency of the kernels. The result of this optimization gave a speedup of at most around 8 times for the objective function and 1.1 times for the gradient compared to the CPU implementation. The second technique was the optimization of memory throughput (see appendix A.3.2). With the intention of reducing the number of global memory accesses, sorting the data to coalescing memory reads, as well as utilize the L1 cache was tested. The second optimization technique gave a total speedup of at most around 39 times for the objective function and 2 times for the gradient compared to the CPU implementation. The last technique that was applied was the use of single precision floating point units instead of double precision in order to improve instruction throughput as well as memory throughput (see appendix A.3.3). This was carried out by not only changing the type but also using faster arithmetic instructions adjusted for single precision. Applying the last optimization technique resulted in a total speedup of at most around 52 times for the objective function and 4 times for the gradient compared to the CPU implementation.

While aiming for a speed up of at least a factor 10 compared to the reference implementation, the parallelization of the objective function and the gradient resulted in different performance increase. The objective function successively manage to perform the evaluation between 20 and 50 times faster than the sequential reference, however, the gradient only gained a speed up of around 4 times faster than the reference.

4.7 Future Work

Even though the objective function did result in a fairly good speedup, there is still room for improvement, such as improved access pattern and more (or less) instruction level parallelism [9]. The major obstacle however is the gradient calculation, having a very low performance boost compared to the objective function.

The main obstacle with the gradient calculation is the heavy use of atomic operations, reducing the parallelism and increasing the time taken to execute. In order to keep the use of atomic operations to a minimum, the use of shared memory to hold a temporary gradient could be a good approach. This could be implemented by using a look-up table and splitting the parameters into smaller pieces, able to fit into the size-limited shared memory. When the contributing calculations are done, the temporary gradient is written atomically to the global memory.

Aside from improving the function and gradient, the next step of this project would be to implement the complete L-BFGS [8] algorithm on the GPU. This would include parallelization of parameter addition, subtraction, dot product and scalar times vector product.

References

[1] Alexandrescu, A. “Modern C++ Design: Generic Programming and Design Patterns Applied”. Boston: Addison-Wesley. pp. 26–28. 2001

[2] Wang, P. (2006) “Fundamental Optimizations in CUDA”. http://developer.download.nvidia.com/GTC/PDF/1083_Wang.pdf (Accessed: 2012-05-19)

[3] Robert, C,P. “The Bayesian Choice” New York: Springer-Verlag. pp 21-24. 2001

[4] NVIDIA Corporation. (2012) “NVIDIA CUDA C Programming Guide” Version 4.1. http://developer.download.nvidia.com/compute/DevZone/docs/html/ C/doc/CUDA_C_Programming_Guide.pdf (Accessed: 2012-03-10)

[5] NVIDIA Corporation. (2009) “Whitepaper NVIDIA’s Next Generation CUDA Compute Architecture”: Fermi. www.nvidia.com/content/PDF/fermi_white_papers/ NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf (Accessed: 2012-05-19)

[6] Gregoire, M. Solter, N. Kleper, S J. ”Professional C++”. Indianapolis: John Wiley & Sons, Inc. pp 703–757. 2011

[7] Nickolls, J. Dally, W,J. ”The GPU Computing Era,” Micro, IEEE , vol.30, no.2, pp.56–69. 2010

[8] Westerlid, T. “Application of L-BFGS to a Large-Scale Poisson MAP Estimation”. Gothenburg: CPL. pp 5–6. 2012

[9] Volkov V. (2010) “Better Performance at Lower Occupancy”. http://www.cs.berkeley.edu/˜volkov/volkov10-GTC.pdf (Accessed: 2012-05-19)

[10] Byrd, R. H. Nocedal, J. Schnabel R B. (1996) “Representations of quasi-Newton matrices and their use in limited memory methods”. http://users.eecs.northwestern.edu/˜nocedal/PDFfiles/representations.pdf (Accessed: 2012-05-20)

[11] Harris, M. (2007) “Optimizing Parallel Reduction in CUDA”. http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf (Accessed: 2012-05-19)

[12] Farber, R. "CUDA Application Design and Development". Waltham: Morgan Kaufmann. pp 154-155. 2011

[13] The C++ Resources Network. (2000) "C Library" http://www.cplusplus.com/reference/clibrary/ctime/clock/ (Accessed: 2012-05-19)

[14] NVIDIA Corporation. (2012) "The CUDA Occupancy Calculator" http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls (Accessed: 2012-05-19)

[15] Suchard, M A. Wang, Q. Chan, C. Frelinger, J. Cron, A. West, M. (2010) "Understanding GPU Programming for Statistical Computation: Studies in Massively Parallel Massive Mixtures" http://www.biomath.ucla.edu/msuchard/publications/papers/Suchard2010.pdf (Accessed: 2012-05-19)

[16] Hwu, W, W. "GPU Computing Gems: Emerald Edition". Burlington: Morgan Kaufmann, pp. 277–291. 2011

[17] Che, S. Boyer, M. Meng, J. Tarjan, D. Sheaffer, J W. Skadron, K. "A performance study of general-purpose applications on graphics processors using CUDA, Journal of Parallel and Distributed Computing", Volume 68, Issue 10, pp. 1370–1380. 2008

[18] NVIDIA Corporation. (2012) "NVIDIA CUDA Zone" .http://www.nvidia.com/object/cuda_home_new.html (Accessed: 2012-05-18)

A The CUDA Architecture

A.1 The CUDA Programming Model

A.1.1 Kernels and Thread Hierarchy

The CUDA software architecture enables developers to execute their programs on NVIDIA GPUs through concurrent global functions, or kernels. The kernel, when called, executes N times in parallel by N different CUDA threads.
These threads are grouped together and organized into a thread block and grids of thread blocks, the structure is up to the programmer or compiler to decide. Each thread block can hold up to 1024 threads, and is organized in three dimensions, (x, y, z). All threads within a thread block will execute an instance of the kernel, and will be assigned a unique thread ID within that thread block. The threads within a thread block can cooperate through shared memory and barrier synchronization. A grid is a structure for blocks to be organized, just like thread blocks, grids uses three dimensional indexes (Fig 5), but the limit of the grid size is substantially bigger, current version (Fermi) can hold up to 65535^3 blocks. Unlike the block structure, threads within the same grid but outside the same block do not have any synchronization or cooperation available between them. Each block within the grid has its own unique block ID [4, 15].

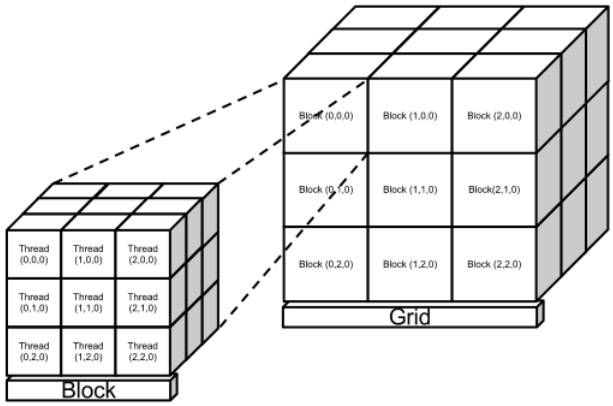

Figure 5: Thread Hierarchy

A.2 GPU Hardware

A.2.1 Streaming Multiprocessor

The heart of the CUDA Architecture, the set of Streaming Multiprocessors (SM) that concurrently executes thousands of threads through a unique architecture called Single-Instruction, Multiple-Threads (SIMT). Each Streaming Multiprocessor contains instruction cache, warp schedulers, dispatch units, registers, execution units and memory (Fig 6).

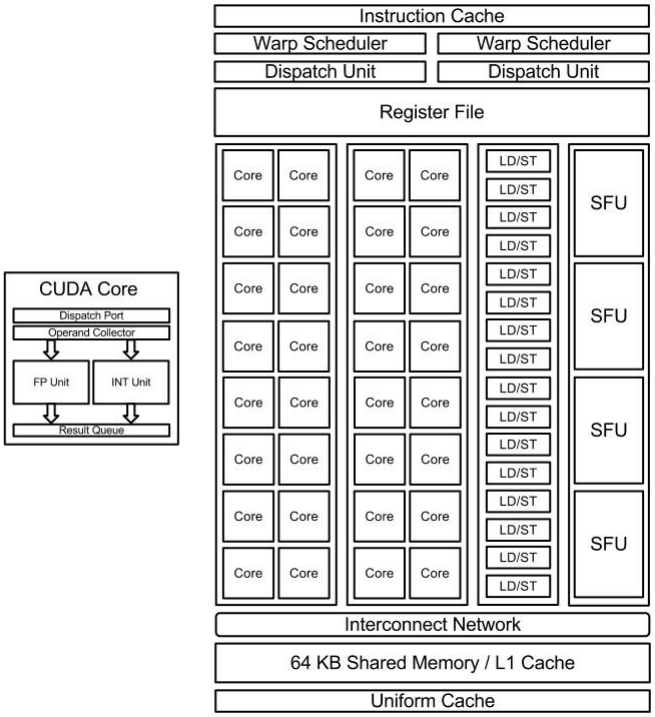

Figure 6: Streaming Multiprocessor

A.2.2 Execution Units and Load/Store Units

The SM has two different execution units, the CUDA core and the Special Function Units (SFU). The SFUs handles and executes complex math operations, such as sine, reciprocal and square root functions. The pipeline of the SFU is decoupled from the dispatch unit and allows for other execution units to be issued while the SFU is occupied [5].

Previously known as a programmable shader or streaming processor core, the CUDA core is a relatively simple processing unit designed to execute only one instruction at a time per thread before rapidly switching to another thread. Each core can do one operation at a time per thread on either an integer unit or a floating-point unit, following the IEEE 754-2008 standard. Due to its multithreaded design, the CUDA cores do not have their own registers, caches or load/store units for accessing memory, instead all resources are shared between each thread in the Streaming Multiprocessor.

To calculate source and destination addresses, the Streaming Multiprocessor is equipped with load-/store units. This unit can execute one store or one load operation per clock cycle, however, when sequential addresses are used for the same thread, the "uniform cache" can be utilized to perform two load operations per cycle [5, 6].

A.2.3 Warp

Every Streaming Multiprocessor manages and executes threads in a group of 32. This collection is called a warp, a reference to the world of weaving. Unlike the thread blocks, the programmer has no saying in how the thread should be divided among the warps. The partitions of threads in warps are issued by the Streaming Multiprocessor, when one or more thread blocks are scheduled to be executed, each warp gets threads with consecutive, increasing thread IDs. Each thread in a warp starts at the same program address, but are free to branch and execute independently. Though, given the SIMT instruction logic, if a thread diverge via a data-dependent conditional branch, the execution of each path taken is serialized by the warp, disabling threads not on that path, converging back to the same execution path when all paths completes. 48 warps can be managed by each SM, utilizing the two warp schedulers and dispatch units [4].

A.2.4 Warp Scheduler

Each Streaming Multiprocessor has two warp schedulers and two dispatch units that work independently of each other. Using this method, two instructions per cycle on two different warps can be issued and executed concurrently. Since 48 warps can be managed per Streaming Multiprocessor, and each warp has 32 threads, 1536 threads per Streaming Multiprocessor can be managed. However, each Streaming Multiprocessor only has 32 CUDA cores to execute instructions from at any given time. Besides CUDA cores, the warp scheduler can also issue instructions to the load/store unit as well as the special function units. The strength of the CUDA infrastructure comes from the fact that switching between threads is instantaneous. When a thread has executed one instruction, another thread can execute on the next clock cycle. This is possible due to warps executing independently, and so the scheduler does not need to check for dependencies from within the instruction stream, resulting in a very high throughput. Using double precision instruction however does not support dual dispatch and will disable the concurrency of the dispatch unit [5, 6].

A.2.5 Multiprocessor Memory

The two types of memory only available inside a Streaming Multiprocessor are a register file and shared memory. The register file is 128Kbyte and divided into 32768 32-bit registers, each register is allocated for individual threads and can only be accessed by that thread. Registers are often used to hold frequently accessed variables, and can be accessed at very high speed. Since the registers are partitioned among the warps in a Streaming Multiprocessor there is a limit on how many registers than can be used per thread before reducing the number of thread blocks per multiprocessor. Given that 1536 threads can be managed concurrently by the Streaming Multiprocessor and the registration allocation unit size is 64 byte, $\frac{32768}{2} = 16384$, $2 \cdot 32$-bit registers can be allocated per SM. Rounding down results in $\frac{16384}{1536} = \lfloor 10\frac{2}{3} \rfloor = 10$, $2 \cdot 32$-bit registers, which is equal to 20, 32-bit registers [4, 15].

Unlike the register file, shared memory is allocated per thread block, and is therefore shared between neighbor threads. The memory is stored in equally-sized memory models, called banks, and can be accessed simultaneously. However, if two or more threads access the same memory module, a bank conflict emerges and the access has to be serialized. The shared memory provide low latency and high bandwidth memory access if no band conflicts occur, however, according to Vasily Volkov[9] shared memory bandwidth is $\geq$6x lower than register bandwidth. Each Streaming Multiprocessor

has 64 kilobyte of memory available for shared memory and L1 cache, distributed as 48 kilobyte /16 kilobyte or 16 kilobyte / 48 kilobyte, user configurable. The size of shared memory distributed per block may reduce the number of thread blocks per multiprocessor. The maximum number of thread blocks per Streaming Multiprocessor is 8, in that case it would give the maximum of $\frac{49152}{8} = 6144$ byte shared memory per thread block before less thread blocks would be issued per multiprocessor, given that 48 kilobyte of shared memory was used [4, 7, 9].

In excess of the register file and the shared memory, each multiprocessor also has access to a read-only constant memory space located in the device memory. To speed up reads from this memory location, the access is through a constant cache, shared between all functional units [4].

A.2.6 Device Memory

The main memory used by the CUDA architecture is located on the GPU in the form of graphics double data rate (GDDR) DRAM, referred to as global memory. The DRAM is conventionally used to hold video images and texture information for 3D rendering, but function as high-bandwidth, off-chip memory for massively parallel applications. Compared to the system DRAM on the CPU motherboard, the GDDR DRAM has a bit more latency, but higher bandwidth to make up for it. The memory is accessed through the PCI-express interface both from the CPU and from the GPU [7, 15].

To reduce the memory access latency for load, store and atomic operations, a parallel two level cache memory architecture is used. Previously mentioned, each Streaming Multiprocessor has a small first-level (L1) data cache, but they also share a larger common unified second-level (L2) cache (Fig 7). The L2 cache connects with the GPU DRAM interface and the PCI-express interface. The L2 cache stores requests from the SMs and their L1 cache, filling the instruction caches and uniform data caches [7].

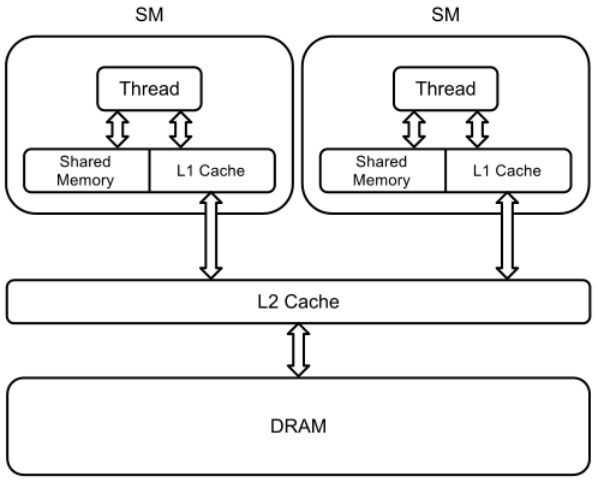

Figure 7: Memory Hierarchy

A.3 Optimization Techniques

To gain maximum performance from CUDA applications, three basic strategies are available: maximum utilization, maximum memory throughput and maximum instruction throughput. To find out which one of these strategies to use, finding the performance limiters by profiling the application is advised.

A.3.1 Utilization

The fundamental rule of maximum utilization on a high level is to assign the right processor for the right work. Serial workloads should be assigned to the CPU while parallel should be assigned to the GPU. If there are multiple kernels, running them in parallel by using asynchronous function calls and streams might be a good way of improving performance. When parallel workloads needs to share data between threads and synchronization is required, algorithms should be mapped so that the inter-thread communication is performed within a single thread block [4].

At a lower level, the application needs to always be "busy" in order to fully use the power of the GPU. In other terms, the warp scheduler should always be issuing instructions for some warp at every clock cycle. Since different instructions take different amount of clock cycles, there should be enough warps to choose from so that there always is at least one that is ready for execution. The word latency is here used as the amount of clock cycles it takes for a warp to be ready to execute its next instruction, and the goal is to always "hide" the latency by having enough ready warps to execute from. The biggest threat to this model is without doubt fetching input operands from the off-chip global memory where latency is between 400 to 800 clock cycles. This can be compared to the local registers with only 22 clock cycles. To completely hide 22 clock cycles, using 2 warp schedulers at the same time, and each scheduler take 2 clock cycles to issue one instruction, 22 warps are required, assuming maximum instruction executing throughput. Synchronization points also have a great contribution to warps not being ready for execution. When warps of the same block needs to wait for the rest of the warps to finish their execution of instructions, these points can force the multiprocessor to idle. To counter this effect, using multiple resident blocks per multiprocessor gives more independent warps [4].

The concept of occupancy is the ratio of active warps to the maximum number of warps supported on a Streaming Multiprocessor. A measurement of how efficiently the kernel will run on the GPU. The limiting factors in the amount of warps supported by a SM are directly related to the amount of registers, shared memory and size of thread blocks. To find out the occupancy from the configurations with the previously mentioned variables, NVIDIA has released a simple Occupancy Calculator [14]. This tool discourages the user from using too much shared memory as well as too many registers. Vasily Volkov [9] however discusses the possibility to use instruction-level parallelism together with thread-level parallelism as an alternative approach to occupancy to increase throughput and performance.

A.3.2 Memory Throughput

Maximizing memory throughput starts with minimizing data transfers with low bandwidth, or specifically, global memory. As previously stated, the latency for reading or writing data from global memory is quite high compared to using on-chip memory such as registers or shared mem-

ory. When accessing global memory, each warp coalesces the access of the threads within the warp into a 32-, 64- or 128-byte memory transaction, depending on size of data accessed by each thread together with memory address distribution. For example, if 4 threads within the same warp each read one double value of 8 byte from global memory, this could be read in one single memory transaction (Fig 8) [4].

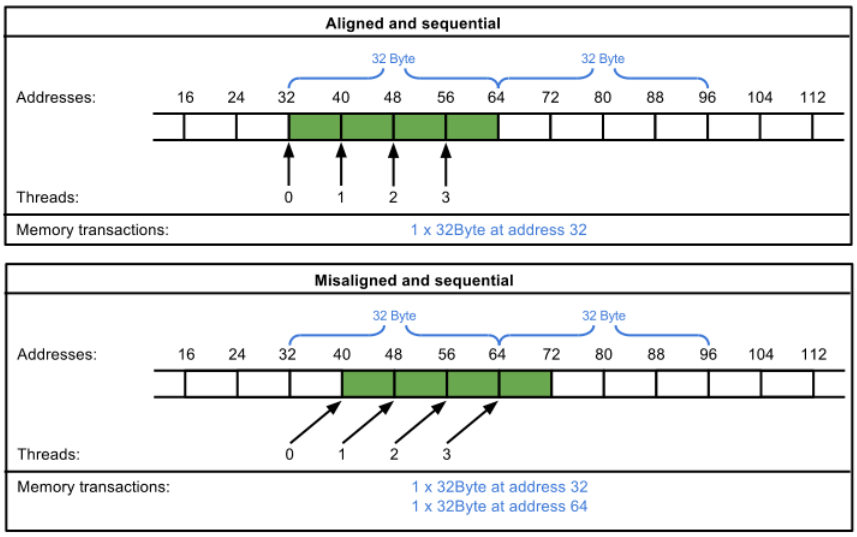

Figure 8

To maximize memory throughput of the shared memory, having no bank conflicts, described in appendix A.2.5, will yield the highest performance [4].

The use of the L1 cache could be useful when frequent access to the same global memory location occurs within the same warp. However, since all the global memory access will be stored in both the L1 and the L2 cache, having scattered memory locations will reduce the throughput, and the L1 cache should be disabled to reduce over-fetch [4].

A.3.3 Instruction Throughput

Basically there are three approaches for maximizing instruction throughput: Choosing the right arithmetic instruction, minimizing divergence and simply reducing the amount of instructions.

When it comes to arithmetic instructions, the number of clock cycles required for execution does of course depend on the instruction, but also on the data type. For example, single-precision functions have higher throughput than double-precision equivalents, but the result will implicitly loose precision.

Any control flow instruction (if, switch, do, for, while) can have a major impact on performance from causing threads of the same warp to take different paths (diverge). By letting the warp diverge,

all paths needs to be serialized and the number of instructions that needs to be executed increases for that warp [4].