GPU Accelerated Gaussian Process Image Retrieval

Lasse Tyrväinen

Masters Thesis
UNIVERSITY OF HELSINKI
Department of Computer Science

Helsinki, April 18, 2016

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Lasse Tyrväinen

Työn nimi — Arbetets titel — Title

GPU Accelerated Gaussian Process Image Retrieval

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Masters Thesis	April 18, 2016	0

Tiivistelmä — Referat — Abstract

Learning a model over possible actions and using the learned model to maximize the obtained reward is an integral part of many applications. Trying to simultaneously learn the model by exploring state space and maximize the obtained reward using the learned model is an exploitation-exploitation tradeoff. Gaussian process upper confidence bound (GB-UCB) algorithm is an effective method for balancing between exploitation and exploration when exploring spatially dependent data in n-dimensional space.

The balance between exploration and exploitation is required to limit the amount of user feedback required to achieve good prediction result in our context-based image retrieval system. The system starts with high amount of exploration and – as the confidence in the model increases – it starts exploiting the gathered information to direct the search towards better results.

While the implementation of the GP-UCB is quite straightforward, it has time complexity of $\mathcal{O}(n^3)$ which limits its use in near real-time applications. In this thesis I present our reinforcement learning image retrieval system based on GP-UCB, with the focus on speed requirements for interactive applications. I also show simple methods to speed up the algorithm running time by doing some of the Gaussian process calculations on the GPU.

Avainsanat — Nyckelord — Keywords
image search, reinforcement learning, CBIR, gaussian processes, multi-armed bandit, UCB, GPU

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction

The growing popularity of devices capable of photography has led to a boom in the number of images taken around the world. Combined with widespread access to Internet it has led to a situation where getting a specific image or an image semantically and aesthetically near enough to the required content is often simply a matter of finding one from the huge pool of online images.

Many are familiar with some more popular web image search engines, such as Google Image Search, Bing Images and Flickr Search. All these system combine context- and content based image search and function well in a limited way, but fail when the user is looking for concepts they can not easily translate into a text search or keywords.

Content based image retrieval is an active and complex research field, and as such there are number of different systems which are state of the art in their respective use cases. Short summary of different approaches and some of these systems are introduced in Section 2.1.

Our image search implementation (IMSE) [1, 2] is a content based reinforcement learning image retrieval system. It starts by showing user a random selection of images, for which the user then gives feedback based on how relevant those images are to the user. After the user feedback is processed the system shows user a new set of images based on the previous selections. The results are thus iteratively refined in order to learn the features present in the images valued by the user.

The novel approach in our system is using Gaussian process upper confidence bound algorithm for selecting relevant images based on user feedback, as well as utilizing graphics processing unit (GPU) to accelerate computation in order to increase the allowable size of the image dataset while keeping the running time reasonable for real time use.

The main topic of this thesis is the GPU acceleration of GP-UCB algorithm, so the algorithm and its implementation, speed and numerical accuracy with different versions will be covered in detail. The remaining parts, such as the user interface, feedback gathering and empirical quality of the results, will only be described briefly.

Section 2 covers the main aspects of image retrieval problems. There is also an overview of different methods used in current content-based image retrieval systems.

Section 3 starts by describing the multi-armed bandit problem and then explains basic principles of the UCB algorithm used to address the exploration-exploitation balancing. Gaussian processes (GPs) and the Gaussian process upper confidence bound algorithm will be presented in section 4. GPU architecture and the GPU accelerated implementation of GP-UCB algorithm will be shown in section 6. Finally the results for speed and numerical accuracy will be in section 8.

2 Image Retrieval

Image retrieval is the task of searching and retrieving images from large collections and databases. There are two major types of image retrieval engines, context- and content-based. Most current popular web-based image search engines – such as Google image search, Bing image search and Flickr search – are mix of the two, but with emphasis on the context-based features, i.e. they return images based context features such as keywords, captions, location information and other context in which the image appears. Their content based search capabilities are mainly based on mapping image content into features that fit into the existing text based search system.

There are some major problems with the context-based approach and pre-generated content features:

1. There may not be any context available for an image and the semantically important features of the image may be hard to identify computationally.

2. Same image can mean semantically different things to different users, or even to the same user at different times. The inverse is also true; people can visualize very different things based on the same description.

3. Textual descriptions and other context features may not capture all relevant features of an image, especially since it is not known in advance what is relevant to the user.

4. Translating the text features to different languages may change the semantic meaning.

2.1 Content-Based Image Retrieval

Content-based image retrieval (CBIR) methods organize images based on their the visual contents, with methods ranging from simple pixel-level similarity comparison to extraction of the image semantic content. Because of the wide variety of methods, the CBIR research also combines results from different fields, such as computer vision, human-computer interaction, data mining, information retrieval and psychology [3].

There are a number of difficulties in content-based image retrieval, largest of which are the the *sensory gap* and the *semantic gap* as defined by Smeulders

et al. [4]:

- "The sensory gap is the gap between the object in the world and the information in a (computational) description derived from a recording of that scene."

- "The semantic gap is the lack of coincidence between the information that one can extract from the visual data and the interpretation that the same data have for a user in a given situation."

Much of the research in the field focuses on bridging these two gaps. Bridging the sensory gap requires methods for handling distortion, clutter, occlusions and other issues which can make it hard to interpret the actual contents of the image. Addressing the semantic gap requires further interpretation of the image in order to assign meaning to its content. It is also necessary to take into account different user requirements and their interpretations of the image contents.

Smeulders et al. introduced a classification of image search domains into *narrow* and *broad*. Images in a narrow domain have limited variability, such as police photographs or single categories of medical imaging. Images in broad domain have high variability and the semantic meanings of the contents of the image are less predictable [4]. IMSE system focuses on image retrieval in the broad domain, so this overview will focus more on general methods than advances in CBIR in narrow domains.

There is a wide range of different methods used in CBIR, some of which focus on feature extraction and others on user interaction. Giving the user the possibility to interact with the system by giving additional input during the search also requires ways to process that input and incorporate it into the relevance criterion [5] , and there are various strategies that have been effectively employed to do so. The next sections contain a short overview of some methods for both feature extraction and query processing.

CBIR System Structure The basic structure of a CBIR system consists of an image database, feature extraction algorithms and a possible feature database, and the image matching and selection algorithm. An outline of the general structure is shown in Figure 2.1. The system takes input from the user, for example a query image or – as in IMSE system – user evaluations of

some existing images in the image database. Output is a selection of relevant images from the image database.

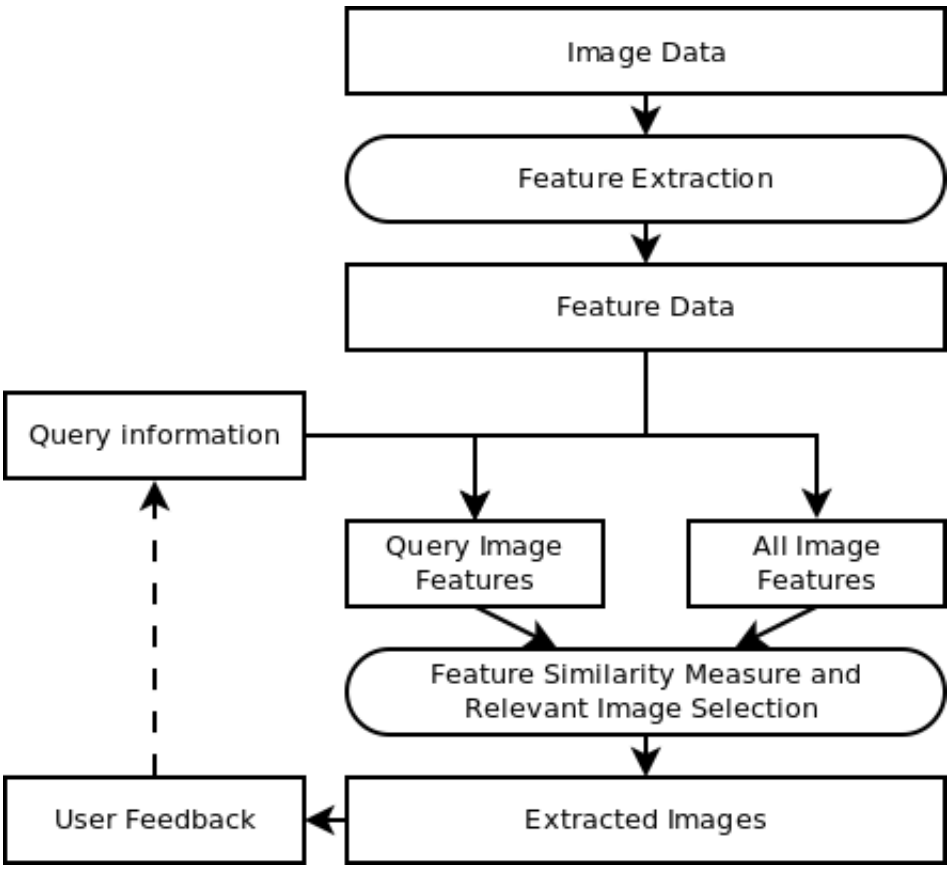

Figure 1: General structure of a content based image retrieval system.

2.2 Low-level Feature Extraction Methods

Feature extraction methods can be roughly divided into low level methods and semantic methods, but there is a fair bit of overlap between the two. Low level features generally do little to bridge the semantic gap, and include features such as color, texture, shape and edge information. Most current CBIR systems combine different low-level and semantic methods.

Listed below are a few common methods for low level feature extraction by category, roughly based on survey by Rajal and Valli [6].

Color methods analyze the color distribution in the image. Common methods include color histogram and its variants, such as invariant color

histogram [7, 8] and local region color histogram [9]. They represent the distribution and intensity of colors in the whole or parts of the image.

Color features also include color moments, which characterize the image color distribution based on different moment functions such as mean, standard deviation and skewness. Other relatively common color based features are dominant color [10] and color correlograms [11]. There is a fairly recent survey of color based methods in CBIR by Van de Sande et al. [12].

Texture based methods analyze the spatial arrangement of color intensities in the image, focusing on the structure of large similar areas. Texture methods include gray level co-occurrence matrix [13], gabor transform and 2-D wavelet transforms [14]. One common application for texture based methods is land cover and surface classification in aerial and satellite images [15, 16].

Shape and edge based methods are closely tied to the texture based methods. Both analyze the spatial arrangement of color intensities, but shape and edge based methods focus on locating the areas where major changes occur. These methods include Gabor filter, histogram of edge directions, region moments, SIFT amongst numerous others.

2.3 Semantic methods

Semantic methods aim to reduce the semantic gap between the low level features and high level semantics. They include feature extraction techniques, preprocessing and organization steps and methods to incorporate user feedback into the query loop. Semantic feature extraction can be done using various supervised and unsupervised machine learning techniques. This is only a brief overview of some of the more popular methods, as there are numerous different methods that have been used.

Unsupervised learning is generally used in CBIR systems either as a preprocessing step to speed up the image retrieval or as a feature extraction method. One example of unsupervised feature extraction is the use of deep auto-encoders [17], while k-means [18] and NCut clustering [19] have been used to cluster related images for easier search. Neural networks have also been successfully used in unsupervised setting – dimensionality reduction with autoencoder is a good example [20].

Supervised learning uses categorized training data, which contains either raw images or low-level image features as inputs and their labels or

some other semantically relevant features as outputs. The algorithm then attempts to learn a generalized mapping from the given input space to output space, which would allow it to give semantic meaning to images not in the training data.

The obvious drawback of supervised learning methods is the requirement for training data. The quality of the results depend on how well the training data has been classified and how well those classes cover relevant features of the dataset. Finding good training data is generally easier in narrow domains where the applications are quite specific, such as medical imaging. In broader domains, for example personal image collection or Internet image search, reliable training data may be harder to obtain.

Some supervised learning methods that have been used in CBIR systems include:

- Support vector machines have been used in combination with low-level features, such as exact Legendre moments [21] and semantic methods such as the relevance feedback [22].
- Neural networks, which are currently a focus of a lot of research in the field of computer vision. Convolutional neural networks have been especially popular in CBIR applications and have yielded some promising results [23, 24, 25, 26]. Neural networks can also be used in relevance feedback processing step and in an unsupervised fashion.

Relevance feedback is a common approach for refining the results after initial images have been shown. This adds the subjective human perception of image similarity into the CBIR system, and allows the user to steer the search towards a goal not known in advance. However, it is severely limited by human fatigue, as a person doing the search might not be willing to invest much time and effort to get high quality results.

There are a number of points to consider when making a user feedback utilizing system [27]:

1. System responsiveness has to be high enough for real time use.
2. Users may have varying goals, such as a specific image ("target search"), mood or color scheme ("category search").
3. Feedback consistency may vary as users are not always good at selecting the most relevant images.

4. How the user provides feedback and whether unselected images are considered to have negative feedback or are ignored.

5. Balancing exploration and exploitation i.e. should the user be provided with the best matching images or the ones that will give the most information for future search.

Our system has been tested both with color histogram low-level features and semantic features extracted using Overfeat [28], a deep convolutional neural network for image classification, object localization and detection. However, feature extraction and the perceptual quality of the results are not within the scope of this thesis.

In order to address the relevance feedback problems listed above, we use several different methods:

- GPU acceleration to increase the responsiveness of the system, addressing the item (1) above, presented in Section 6.

- Gaussian process regression algorithm to predict the best images to show to the user. This, along with semantic features generated using Overfeat, improves the quality of the results shown to the user (items 2 and 3). This is introduced in Section 4.

- A variant of the Upper Confidence Bound algorithm to balance exploration and exploitation. Combined with the Gaussian process regression it allows us to achieve provably good convergence bounds (item 5). This is introduced in Section 3.

3 Reinforcement Learning

The field of reinforcement learning research aims at designing algorithms by which autonomous agents learn to behave in an appropriate fashion in the given environment. As opposed to supervised learning, where the correct behavior is given, reinforcement learning works in an environment where direct examples of correct behavior are not available [29]. Actions are instead scored by some other performance criterion.

Consider an example of a person attempting to park a car. The information gathered by the driver does not directly tell which way to move the wheel or which pedal to push. Instead the information allows the driver to evaluate the goodness of different actions related to the goal. Inexperienced driver may do this evaluation poorly and will probably require more attempts to get a good result, while an experienced driver can more reliably tell what to do in a given situation and can often park a car with the first attempt.

The experienced driver has – at least partially – learned the correct actions in different situations by trial and error. Choosing wrong actions will lead to worse situation or even an accident, while the correct actions eventually lead to the car being parked. These can be considered to be negative and positive feedback in reinforcement learning scenario.

The driver of the previous car example roughly corresponds to a reinforcement learning agent trying to learn an optimal policy. Reinforcement learning models the problem as a Markov decision process, which is a discrete time stochastic control process satisfying the Markov property. The Markov decision process is represented by a tuple (S, A, P, R, γ), where

- S is a finite set of states,
- A is a finite set of actions,
- P is a the matrix of state transition probabilities $\mathbb{P}(s, s') = \mathbb{P}[s'|s, a]$,
- R is a reward function $R(s, a) = \mathbb{E}[r|s, a]$ and
- γ is a discount factor.

A process satisfying the Markov property is one where the future states only depend on the current state, i.e.

$$\mathbb{P}[s_{t+1}|s_t] = \mathbb{P}[s_{t+1}|s_1, \ldots, s_t],$$

where $s_t \in S$ is the state of the process at time t.

3.1 Exploration vs. Exploitation

The main problem in our image search system is the exploration-exploitation dilemma, where an agent simultaneously attempts to explore the environment to acquire new knowledge and exploit existing knowledge to optimize decisions. A simple instance of this problem is known as the **N-armed bandit problem**, where each 'arm' represents a possible action with an unknown probability distribution of rewards [30].

In such scenario the agent can at each time step either choose to exploit current knowledge by playing the best known arm or choose to explore by playing an arm for which the uncertainty of the reward distribution is the largest. In order to achieve optimal result these actions need to be correctly balanced.

This model has been used to solve problems such as allocation of resources to different projects, optimizing web site designs and for adaptive routing. In each case the payoff from different actions is not well known in advance thus requiring exploration of the different options in addition to exploitation of the best known option.

Problem definition

An N-armed bandit problem is a Markov decision process with one state and N possible actions. As there is only one state, the rewards are associated only with different actions $R(s_t, a_{i,t}) = R(a_{i,t})$, where $i = 1, \ldots, N$ is the index of the action. In the simplest case reward distributions of different arms are assumed to be independent of each other.

The sequence of action decisions is the *decision set* $D_t = x_1, \ldots, x_T$ for some fixed amount of turns T, where $x_t = a_{i,t}$ is the observed action with index i at turn t. T_i is used as a shorthand to denote the number of times action i has been chosen at turn T.

The goal is to maximize the cumulative reward $\sum_{t=1}^{T} R(x_t)$, which is equivalent to minimizing the *cumulative regret* R_T. This is defined by

$$R_T = \sum_{t=1}^{T} (\Delta_t)$$

where $\Delta_t = R(x^*) - R(x_t)$ and $x^* = \text{argmax}_{x \in D} R(x)$, i.e. Δ_t is the difference between the made and an optimum decision.

The best possible asymptotically bounded regret for some reward distributions, such as Gaussian, Poisson and Bernoulli can be calculated using the asymptotical expected bound on the number of non-optimal choices, as shown by Lai and Robbins [31]:

$$\mathbb{E}[T_j] \leq \frac{\ln T}{KL(p_j||p^*)}, \text{ as } T \to \infty,$$

where T_j is number of times the action with index j has been chosen and $KL(p_j||p^*)$ is the Kullback-Leibler divergence between the reward density of choice j and the optimal choice.

3.1.1 Upper Confidence Bound Algorithms (UCBs)

The upper confidence bound algorithm and its variants are widely used to address the exploration-exploitation trade-off, as they are easy to implement and have good performance both in theory and in practice. The basic algorithm has since seen numerous modifications for different use cases.

The two most important aspects of the upper confidence bound algorithms is that (1) they have good bounds for *expected* regret after T decisions rather than asymptotical bounds as the number of decisions approaches infinity and (2) they are simple and practical to implement.

The basic UCB algorithm was introduced by Auer et al. [32]. The first algorithm in their work, UCB1, is shown in Algorithm 1.

Algorithm 1: UCB1 [32]

Initialization: Play each machine (i.e. possible action) once;
while *Not done* **do**
  T = number of plays (decisions) this far;
  T_i = number of times machine i has been chosen;
  $\bar{r}_i$ = average reward obtained from machine i;
  $j = \text{argmax}_i \, (\bar{r}_i + \sqrt{\frac{2 \ln T}{T_i}})$;
  play machine j;
end

Auer et al.[32] show that the expected regret after T choices have been

made using this algorithm with N possible different actions is at most

$$\mathbb{E}[R_T] = \left[8 \sum_{i:\mu_i<\mu^*} \left(\frac{\ln T}{\Delta_i} \right) \right] + \left(1 + \frac{\pi^2}{3} \right) \left(\sum_{j=i}^{N} \Delta_j \right),$$

where μ_i is the expected reward for choice at index $i = 1, \ldots, N$ and $\Delta_i = \mu^* - \mu_i$ is the difference between actual and optimal decision.

This bound follows from the fact that using policy UCB1, the expected number of times a non-optimal action j is chosen after T choices is bounded by

$$\mathbb{E}[T_j] \leq \frac{8}{\Delta_j^2} \ln T + 1 + \frac{\pi^2}{3}.$$

This bound is worse than the bound by Lai and Robbins, as the leading constant in their asymptotical bound is $1/KL(p_j||p^*) \leq 1/(2\Delta_j^2)$. To get better constants for the regret bound, Auer et al. also provide algorithm UCB2. It allows bringing the leading constant arbitrarily close to $1/2\Delta_j^2$, but has an initialization step where each machine is played once.

Both these algorithms apply to the case where the reward distributions of different actions are independent, which not the case in our application. In our case each image is represented by a vector of features, and images close to each other in the feature space are assumed to be similar in some fashion. This information is significant as we cannot ask the use to go through the whole collection of images just as an initialization step.

The Gaussian process UCB algorithm has a lot in common with the basic UCB algorithms, but instead of the policies described above it uses the Gaussian process regression to obtain information on the expected rewards and uncertainties of different choices. The arms are not considered to be independent; the user's feedback gives information on all similar arms, i.e. the arms that are near the selected one in the input space. The next section will describe the basics of the GP regression algorithm.

4 Gaussian Process Regression

Gaussian process regression is a supervised learning method, where the goal is to learn a function f capable of predicting the correct output value for all possible input values [33]. This function has to be induced from a finite number of observations of the – possibly noisy – output of the underlying process. In other words, solving a regression problem with real-valued variables means we try to learn a mapping $h : \mathcal{X} \to \mathcal{Y}$ from some input space of n-dimensional real vectors $\mathcal{X} = \mathbb{R}^n$ to an output space of real variables $\mathcal{Y} = \mathbb{R}$.

GP is a Bayesian method, so instead of attempting to identify a 'best-fit' model of the observations it computes a posterior distribution over the models.

4.1 Bayesian methods

Many traditional regression methods focus on identifying a single best fit model for the observations and use that best fit model to make future predictions. This springs naturally from the assumption in "traditional" frequentist methods that the observations are a sample of an infinitely repeatable fixed process, i.e. the underlying parameters are fixed. In Bayesian inference statistical conclusions about parameters θ of the hidden process are made by calculating the conditional probabilities of the parameters given the observations. In addition to the best fit this also gives information on how certain that estimate is and probabilities for other possible parameters.

Bayesian methods require a prior distribution for the θ, which can be either uninformative or contain assumptions about the underlying process. The Bayes rule is then used for computing a posterior probability distribution

$$p(\theta|\mathcal{D}) = \frac{p(\theta, \mathcal{D})}{p(\mathcal{D})} = \frac{p(\mathcal{D}|\theta)p(\theta)}{p(\mathcal{D})}, \tag{1}$$

where θ is the parameter prior and $\mathcal{D}$ is the set of observations. Thus the posterior probability distribution incorporates both our prior assumptions and the evidence gained from the observations. It allows us to evaluate the uncertainty in the parameters θ after observing the evidence $\mathcal{D}$.

Predictions with Bayes rule

The quantity $p(\mathcal{D}|\theta)$ on the right hind side of equation 1 can be interpreted as the *likelihood function* of the parameter vector θ; it shows how probable the observations are for different parameters.

If the variables are continuous, the probability of the observations can be calculated with $p(\mathcal{D}) = \int p(\theta)p(\mathcal{D}|\theta)d\theta$, and with fixed $\mathcal{D}$ the factor $p(\mathcal{D})$ can be ignored altogether to get

$$p(\theta|\mathcal{D}) \propto p(\mathcal{D}|\theta)p(\theta), \tag{2}$$

which is equal to the verbal description of the Bayes' theorem

$$\text{posterior} \propto \text{likelihood} \times \text{prior}.$$

In the above formula posterior is the posterior distribution of different parameters given the observations, likelihood is the likelihood function over the different parameters θ given the observations, and prior is the prior assumptions about the parameters.

The fundamental difference between the frequentist paradigm and Bayesian approach can be expressed by their different use of the likelihood function $p(\mathcal{D}|\theta)$. Bayesians consider the observations $\mathcal{D}$ to be fixed and calculate the probability of different parameter variables. In Bayesian reasoning the uncertainty in the parameters is expressed through a probability distribution over θ. It is also incorporated in the prior assumptions of the underlying probabilities of the parameters.

The frequentists assume that $\mathcal{D}$ is just a representative sample of an infinite repeatable process. The parameters of the process are assumed to be fixed, and the frequentist methods focus on determining the parameters using some form of estimator. One such is *maximum likelihood*, in which θ is set to the value that maximises the likelihood function $p(\mathcal{D}|\theta)$.

Given these differences, the Bayesian method fits our problem definition better. It allows us to easily incorporate new observations into the predictions and provides easily understandable measures for best fit and uncertainty of parameters. This is especially true for predicting with Gaussian Process regression, as shown in the next section.

4.2 Predicting with GP regression

Intuitively two input vectors residing near each other in the input space should have highly correlated output values. This can be formalized using Gaussian process, which is a Bayesian method for learning a mapping $\mathbf{f} : \mathcal{X} \to \boldsymbol{y}$ from an d-dimensional real-valued input vectors $\boldsymbol{x} \in \mathcal{X}$ to real-valued outputs $y \in \boldsymbol{y}$.

A Gaussian process is completely specified by its mean function and covariance function [33]. The mean function gives the estimated best fit model for parameters, while the covariance function allows estimating the uncertainty about the mean function in different locations of the parameter space.

Defining mean function $m(x)$ and the covariance function (kernel) $k(x, x')$ of the real process $f(x)$ as

$$m(\boldsymbol{x}) = \mathbf{E}[f(\boldsymbol{x})], \tag{3}$$

$$k(\boldsymbol{x}, \boldsymbol{x}') = \mathbf{E}[f(\boldsymbol{x}) - m(\boldsymbol{x}))(f(\boldsymbol{x}') - m(\boldsymbol{x}'))], \tag{4}$$

the Gaussian process can be written as

$$f(\boldsymbol{x}) \sim \mathcal{GP}(m(\boldsymbol{x}), k(\boldsymbol{x}, \boldsymbol{x}')).$$

If we have prior knowledge that the GP has zero mean and a known kernel function, the posterior can be computed using Bayesian inference conditioned on the observations.

If there are n observed data points and n_* unobserved data points, $K(X, X_*)$ denotes the $n \times n_*$ covariance matrix $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$, where $\boldsymbol{x}_i \in X$ and $\boldsymbol{x}_j \in X_*$. Given observed data points X, unobserved data points X_* and function values $\mathbf{f}$ corresponding to the observed function values at the observed data points, the predictions for unobserved function values $\mathbf{f}_*$ can be calculated by computing the posterior predictive distribution. The predictive Gaussian distribution $\mathcal{N}$ conditioned on the noise free observations is

$$\begin{aligned} \mathbf{f}_* | X_*, X, \mathbf{f} \sim \mathcal{N}(&K(X_*, X)K(X, X)^{-1}\mathbf{f}, \\ &K(X_*, X_*) - K(X_*, X)K(X, X)^{-1}K(X, X_*)). \end{aligned} \tag{5}$$

The case with noisy observations is very similar. Assuming we only have access to noisy observations $y = f(\boldsymbol{x}) + \epsilon$, where ϵ is additive independent

identically distributed Gaussian noise with variance σ_n^2, the prior becomes $\mathrm{cov}(y_p, y_q) = k(x_p, x_q) + \sigma_n^2 I$ and the equation 5 becomes

$$\mathbf{f}_*|\mathbf{y}, X_*, X \sim \mathcal{N}(\bar{\mathbf{f}}_*,\ \mathrm{cov}(\mathbf{f}_*)) \tag{6}$$

$$\bar{\mathbf{f}}_* = \mathbb{E}[\mathbf{f}_*|\mathbf{y}, X_*, X] = K(X_*, X)[K(X, X) + \sigma_n^2 I]^{-1}\mathbf{y}, \tag{7}$$

$$\mathrm{cov}(\mathbf{f}_*) = K(X_*, X_*) - K(X_*, X)[K(X, X) + \sigma_n^2 I]^{-1}K(X, X_*). \tag{8}$$

In our use case the noisy observation assumption allows us to quantify the uncertainty in the feedback given by the user. This addresses the item three in the relevance feedback observations in Section 2.3, the fact that user feedbacks on image relevance are not always consistent.

Gaussian process regression combined with Upper Confidence Bound algorithm gives an excellent tool to address the exploration-exploitation dilemma, as it provides estimates for both best fit values and their uncertainty. This makes the implementation of Gaussian process Upper Confidence Bound algorithm very straightforward.

5 Gaussian Process UCB Algorithm

Using GP optimization in the multi-armed bandit setting is fairly straightforward and intuitively easy to understand. The values in Gaussian process posterior mean are an estimate of the correct values of the unexplored actions given the observed actions. GP posterior variance gives a confidence interval for the mean estimates. As such, using the results of Gaussian process regression in the UCB algorithm only requires a method of balancing exploration and exploitation, which is done by choosing proper weights for mean and variance.

The problem definition is similar to the case of independent arms. We attempt to sequentially optimize a reward function $f : \mathcal{X} \to \mathbb{R}$, where $\mathcal{X}$ is the input space. This is done by choosing a point $\boldsymbol{x}_t \in \mathcal{X}$ and evaluating the result $y_t = f(\boldsymbol{x}_t + \epsilon_t)$ at that point. The goal is to maximize the sum of rewards, which is equal to minimizing the cumulative regret.

Assuming the unknown function is f and its maximum point is $\boldsymbol{x}^* = \text{argmax}_{\boldsymbol{x}\in\mathcal{X}} f(\boldsymbol{x})$, the instantaneous regret at time t is $r_t = f(\boldsymbol{x}^*) - f(\boldsymbol{x}_t)$. As was the case with independent arms, cumulative regret $R_T = \sum_{t=1}^{T} r_t$ at time T is the sum of instantaneous regrets at each time $t = 1, \ldots, T$.

Optimal or a *no-regret* algorithm would be able to choose $\boldsymbol{x}^*$ at each time t. While this is not possible except in the trivial case where $\boldsymbol{x}^*$ is known in advance, it is possible to have *asymptotically* no-regret algorithm where $\lim_{T\to\infty} R_T/T = 0$ given some assumptions about f. However, the result is of limited practical use due to strictness of the required assumptions [34].

In order to enforce smoothness of f it is modeled as a sample from a Gaussian process: "a collection of dependent random variables, one for each $\boldsymbol{x} \in D$, every finite subset of which is multivariate Gaussian distributed in an overall consistent way" [34] [33]. We also assume that GPs not conditioned on the data have mean $\mu \equiv 0$, while the GP conditioned on data is specified by it's mean and covariance functions, as mentioned in section 4. A bounded variance is also assumed, i.e. $k(\boldsymbol{x}, \boldsymbol{x}) \leq 1, \boldsymbol{x} \in D$.

5.1 Experimental Design Algorithm

Experimental design algorithm is a pure exploration algorithm and as such does not suit our purposes directly. However, it will be briefly described here as it provides a useful result for analysis of the GP-UCB algorithm.

If the purpose was to simply maximize the knowledge of the function f as rapidly as possible, we could use Bayesian Experimental Design (ED) [35] algorithm. The aim would then be to maximize the information gain from the noisy observations $\boldsymbol{y}_A = \boldsymbol{f}_A + \epsilon_A$, where $A \subset D$ is set of sampling points in input space D, $\boldsymbol{f}_A = [f(\boldsymbol{x})]_{\boldsymbol{x}\in A}$ and $\epsilon_A \sim N(0, \sigma^2\boldsymbol{I})$.

This approach is wasteful as it aims to decrease the uncertainty globally instead of limiting it to the areas with the best potential rewards. In fact it does not even depend on the actual observations y_t, only utilizing their locations in the input space. However, maximum information gain after T rounds is an essential part of the regret bounds for the GP-UCB algorithm.

The information gain from these sampling points is quantified by the reduction of the uncertainty about f after observing $\boldsymbol{y}_A$:

$$\mathrm{I}(\boldsymbol{y}_A; f) = \mathrm{H}(\boldsymbol{y}_A) - \mathrm{H}(\boldsymbol{y}_A|f).$$

For Gaussian distribution the entropy is $\mathrm{H}(N(\boldsymbol{\mu}, \boldsymbol{\Sigma})) = \frac{1}{2}\log|2\pi e\boldsymbol{\Sigma}|$, which means that in this setting

$$\begin{aligned}\mathrm{I}(\boldsymbol{y}_A; f) =& \mathrm{I}(\boldsymbol{y}_A; \boldsymbol{f}_A)\\ =& \frac{1}{2}\log|\boldsymbol{I} + \sigma^{-2}k(\boldsymbol{x}, \boldsymbol{x'})|,\end{aligned}$$

where $\boldsymbol{x}, \boldsymbol{x'} \in A$ [34].

Finding information gain maximizer in this setting is an NP-hard problem, but it can be approximated effectively using a greedy strategy. Let $F(A) = \mathrm{I}(\boldsymbol{y}_A; f)$. Information gain $F(A)$ is submodular, as shown by Krause and Guestrin [36], so by selecting

$$\boldsymbol{x}_t = \operatorname*{argmax}_{\boldsymbol{x}\in D} \sigma_{t-1}(\boldsymbol{x}), \tag{9}$$

which is equivalent to $\boldsymbol{x}_t = \operatorname{argmax}_{\boldsymbol{x}\in D} F(A_{t-1} \cup \{\boldsymbol{x}\})$, leading to a near optimal solution:

$$F(A_T) \geq (1 - 1/e) \max_{|A|\leq T} F(A),$$

which is a constant factor approximation of the optimal solution [37]. This result is part of the GP-UCB analysis in the following section.

5.2 GP-UCB Algorithm

In order to utilize the information of function values at the sampled points $\boldsymbol{y}_A$ instead just the set of locations A in the input space, we also have to take into account the mean values of the Gaussian process regression. Always selecting the information gain maximizer would be an exploration only strategy shown in Equation 9, while selecting points $\boldsymbol{x}_t = \text{argmax}_{\boldsymbol{x} \in D} \, \mu_{t-1}(\boldsymbol{x})$ would be pure exploitation and would likely get stuck in a local optima.

These two approaches can be balanced by choosing

$$\boldsymbol{x}_t = \underset{\boldsymbol{x} \in D}{\text{argmax}} \, \mu_{t-1}(\boldsymbol{x}) + \beta^(1/2)_t \sigma_{t-1}(\boldsymbol{x}),$$

where β_t are appropriately chosen constant balancing factors for exploration-exploitation tradeoff. This objective greedily selects both points where reward is expected to be high and the uncertainty of the reward is large. This is the *Gaussian process upper confidence bound* rule (GP-UCB) [38, 34]. The pseudocode for GP-UCB algorithm is shown in Algorithm 2.

Algorithm 2: GP-UCB [32]

Input: Input space D, GP prior $\mu_0 = 0$, σ_0, β_t
while *Not done* **do**
  n = number of plays this far;
  $\boldsymbol{x}_n = \text{argmax}_{\boldsymbol{x} \in D} \, \mu_{n-1}(\boldsymbol{x}) + \sqrt{\beta_t} \sigma_{t-1}(\boldsymbol{x})$;
  Sample $y_t = f(\boldsymbol{x}_t) + \epsilon_t$;
  Update μ_t, σ_t using Gaussian process regression
end

5.3 Regret Bounds

The regret bounds for the setting $f \sim GP(0, k(\boldsymbol{x}, \boldsymbol{x'}))$ for finite D were also established by Srinivas et al. [34]. Maximum information gain after T rounds is defined as:

$$\gamma_T := \max_{A \subset D : |A| = T} I(\boldsymbol{y}_A; \boldsymbol{f}_A)$$

For the GP prior Srinivas et al. obtain a regret bound $\mathcal{O}^*(\sqrt{T \gamma_T \log |D|})$ with high probability, where $\mathcal{O}^*$ is a variant of $\mathcal{O}$ without the log factors.

More precisely, given $\delta \in (0,1)$ and $\beta_t = 2\log(|D|t^2\pi^2/6\delta))$,

$$Pr\left\{R_T \leq \sqrt{C_1 T \beta_T \gamma_T},\ \forall T \geq 1\right\} \geq 1-\delta,$$

where $C_1 = 8/\log(1+\sigma^{-2})$.

As the previous bound depends on the information gain, bounding the γ_T for appropriate kernels is required for these bounds. For a squared exponential kernel $k(\boldsymbol{x}, \boldsymbol{x}') = \exp(-(2l^2)^{-1}||\boldsymbol{x} - \boldsymbol{x}'||^2)$, where l is a length scale parameter, the bound on information gain is $\gamma_T = \mathcal{O}((\log T)^{d+1})$. This gives a high probability regret bound $\mathcal{O}^*(\sqrt{T}(\log T)^{\frac{d+1}{2}})$ [34].

6 GPU Acceleration

A graphics processing unit (GPU) is specialized computation unit made for accelerating the rendering of graphics. It performs parallel calculations on data to create images in frame buffer for an output to display. In order to understand some of the design choices and limitations of the GPU hardware, it is important to know a bit of their history.

One starting point for the evolution of GPUs was 1990's VGA controllers, which started to include some functions necessary for 3D graphics calculations. Fueled by the customer demand, software makers quickly started utilizing this new programming power to create better 3D graphics, especially in games. This cycle led to manufacturers creating more powerful graphic controllers and led to the first graphics processing unit by Nvidia in 1999.

As the new GPUs kept getting more powerful and some of their fixed function logic was replaced by programmable processors they attracted developers of non-graphical applications. While expressing algorithms as graphics computations was tedious, initial results were good enough to spark more interest in utilizing GPUs for general purpose computation [39]. The manufacturers soon realized the potential new market and started making more flexible hardware to accommodate the needs of general purpose use. Those changes along with the development of toolkits for non-graphical GPU programming eventually led to general purpose processing capable GPUs (GPGPUs). Two leading GPGPU programming platform implementations in use today are Nvidia's CUDA (Compute Unified Device Architecture) and OpenCL maintained by Khronos Group.

While both of the leading platforms are quite well supported and have similar performance, we chose CUDA for our implementation as we have access to Nvidia GPUs.

The processing paradigm of the GPU does not exactly fit the traditional models of computation such as single instruction, multiple data (SIMD) and multiple intructions, multiple data (MIMD) in Flynn's taxonomy [40]. As a slightly simplified example the Nvidia GeForce GTX Titan used in our experiments has 14 streaming multiprocessors, which can perform different instructions to different parts of the data, so at that level the hardware is MIMD. On the other hand each multiprocessor contains 192 32-bit CUDA cores which compute same instructions for different data, so their architecture

is closer to SIMD.

Nvidia refers to the CUDA parallel programming model as SIMT – Single Instruction Multiple Thread. It is closely related to SIMD, but differs on how it handles branching. If a computation branches in a SIMD system, it will serially process all different branches. Only threads following the selected path are able to continue in parallel while other threads are blocked until the selected branch is complete. In the SIMT model the threads are scheduled together in *warps*. Only the threads in a warp share a program counter, so execution path divergence only causes thread blocking inside a warp; different warps execute independently.

The difference in the model of computation often makes direct comparisons between GPUs and CPUs rather misleading, as they are best suited for very different tasks. The computation on GPU is also largely asynchronous with the CPU, so it's possible to run a serial task or a chunk of the parallel task on CPU while the GPU computes the most of the parallel parts of the code. Thus, except for the overhead from data chunking, memory allocations and transfers, and kernel launches, the GPU processing capability can often be added to the existing CPU processing speed instead of directly competing with it.

Even with the advances in general purpose GPU implementations on both hardware and software side, the differences in CPU and GPU programming have to be kept in mind in order to get reasonable performance from the GPU. Current CPU architectures are essentially iterative refinements of the earliest microprocessors from the 1970s, such as Intel 4004 and especially the first x86 architecture processor Intel 8086. On the other hand the GPU general programming capabilities have been added to hardware that was not originally made for general purpose programming. Thus a programmer using GPUs for computation has to be more aware of the limitations of the hardware in order to write reasonably efficient code for the platform.

Some of the requirements for the tasks that can be efficiently processed on a GPU are:

- **Parallelizability**: This is fairly obvious, but the task has to be highly parallelizable in order to utilize the GPU. A GPU can have hundreds or thousands floating point units, so in order to fully utilize it you need to perform that many calculations simultaneously. The computations also need to be mostly independent of each other, as synchronization

and moving data between threads can be expensive.

Slightly less obviously the memory accesses also need to be highly parallel and independent. Since threads inside a grid are not guaranteed to execute in a specific order, there are no guarantees on the order of reads and writes to memory locations if used by multiple threads. Enforcing atomicity can be computationally expensive, although a bit less so on recent GPU generations with driver and hardware level support for some atomic operations.

- **Low degree of branching**: A highly branching code can rarely get good GPU performance. Logic statements use clock cycles, and a branch inside a warp will cause part of the multiprocessor cores to go idle until all branches are done. It can also complicate memory accesses. This reflects the difference in design principles; a CPU is designed to handle flow control and has far more of the physical processor area designated to cache and control units than a GPU.

- **Coalesced memory accesses:** This is related to the parallel memory accesses. When accessing the global GPU memory, a highest memory bandwidth can be achieved when threads in the warp access consecutive, cache line aligned memory locations. This can be hard or impossible to achieve on tasks that require random or widely divergent memory accesses.

  For example if every thread in a warp access an adjacent, cache line aligned 4-byte memory location, that access can be coalesced into a single 128-byte L1 cache transaction. Same 128-byte L1 cache transaction is performed even if the warp only requires one 4-byte memory access, which would lead to only 1/32 of the optimal memory bandwidth being used.

6.1 Nvidia Titan Architecture

Nvidia GeForce GTX Titan is one of the most powerful consumer GPUs on the market at the time of writing. It supports CUDA compute capability 3.5, and contains fourteen GK110 Kepler SMX architecture streaming multiprocessors, for which the overall processing unit composition is shown in figure 6.1. Each GK110 SMX contains 192 single-precision CUDA cores, 64 double-precision

Figure 2: Kepler SMX architecture [41]

units, 32 special function units and 32 load/store units. The base clock rate for multiprocessors is 837MHz. Memory block rate is 3004MHz and the memory bus width is 384 bits.

These give a theoretical 32-bit compute capability of 4.5 TFLOPS and a theoretical maximum memory bandwidth of 288 GB/s. Compared to 225 GFLOPS and 25,6 GB/s of Intel Core i7 3770, the theoretical compute performance of Nvidia Titan is around twenty times faster and memory bandwidth roughly ten times greater. These figures must be considered

in the right perspective; GPU requires massively parallel computations with minimal branching to achieve even a large fraction of the maximum performance, while CPUs are much more flexible.

6.2 Energy efficiency

The specialized design also allows GPUs to be very power efficient compared to most currently available general purpose computation platforms. A reasonably accurate ballpark figure can be calculated using the manufacturer provided thermal design power (TPD) and assuming the processor consumes that much energy for maximum performance. The Intel specifications give i7 3770 a TDP of 77W, while Nvidia specifications list 250W for Titan. This gives us 3 GFLOPS per watt performance for i7 and 18 GFLOPS per watt for Titan.

The performance in actual computational tasks is much lower than the theoretical figures, as full utilization of the GPU is hard to achieve in practice and a GPU accelerated system still needs a CPU in order to function.

Despite these limitations heterogenous computation systems are occupying all top 10 positions in the Green500 November 2015 ranking [42], the most current list of energy efficient supercomputers at the time of writing. The top position is held by the Shoubu supercomputer from RIKEN, which utilizes Intel Xeon CPUs and PEZY-SC 1.4 accelerators [43]. Remaining nine positions are occupied by supercomputers using Intel Xeon processors and graphics processing units. Results for top three systems are shown in Table 1, with the remaining systems in top 10 use Intel Xeon and Tesla K80 or K40 combination and all systems in top 40 using heterogenous computation platforms.

The performance shown in Green500 list is tested using High-Performance Linpack benchmark (HPL) [44]. HPL has been criticized for being too optimized for specific machines and being focused on dense matrix computations [45].

However, even given its limitations the HPL benchmark is widely adopted. It is used here since it is the only actual benchmark for which results are readily available for different platforms. As such it is more useful than theoretical results, even if it does not cover all possible use cases in scientific computing.

#	$\frac{\text{MFLOPS}}{\text{W}}$	Site	Computer	Total Power (kW)
1	7031.58	Institute of Physical and Chemical Research (RIKEN)	Xeon E5-2618Lv3 8C 2.3GH, PEZY-SC	50.32
2	5331.79	GSIC Center, Tokyo Institute of Technology	Intel Xeon E5-2620v2 6C 2.1GHz, NVIDIA Tesla K80	51.13
3	5271.81	GSI Helmholtz Center	Intel Xeon E5-2690v2 10C 3GHz, AMD FirePro S9150	57.15

Table 1: Top 3 most energy efficient supercomputers in the world [42].

6.3 CUDA Thread Organization

Compared to a CPU threads, the GPU threads have a very low launch overhead, but given the hardware limitations shown in SIMT model, they are also a lot less versatile. While there is an ongoing process to make heterogenous computing with CPU and GPU more transparent from the programmer's point of view, writing fast GPU code still requires the programmer to be aware of the strengths and limitations of the underlying technology.

While the basic elements of the thread and memory organization remain the same between different CUDA capable hardware, the optimal and maximum sizes for different elements, such as grid, block and shared memory, vary between hardware versions. Unless otherwise noted, later references for such figures are for the Titan GPU and CUDA compute capability 3.5 hardware.

The threads in CUDA code are organized on three levels, shown in Figure 3. On the lowest level there are individual *threads*, which can be identified in either one-, two- or three-dimensional thread *block* by their 3-component threadIdx vector. These thread blocks are in turn organized into grids, which can also be one-, two- or three-dimensionally indexed. Individual blocks in the grid are identified by their 3-component blockIdx vector. The multidimensionality of the grids and blocks exists just to help in mapping multidimensional problems into CUDA programs, but does not affect the

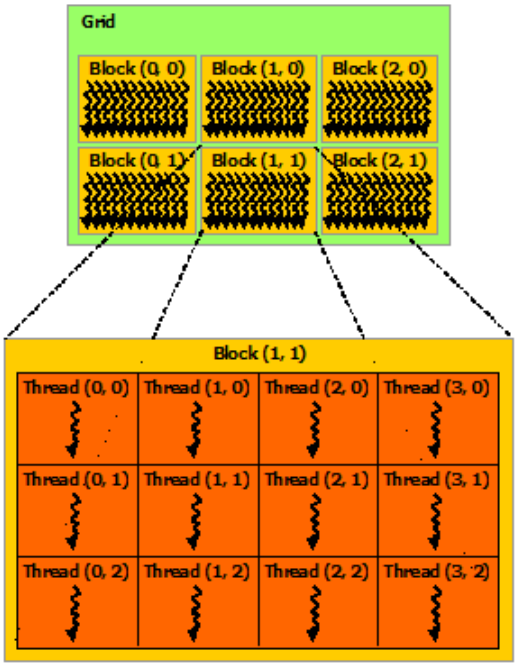

Figure 3: The organization of CUDA threads.

execution performance as the multidimensional structures are mapped into one dimension for execution.

Each block is assigned to single multiprocessor for execution so threads inside a block can use registers and shared memory – which are physically on the same chip – for inter-thread communication and can be explicitly synchronized using computationally inexpensive `__syncthreads()` calls. If the size and the resource usage of the blocks are low enough, multiple blocks can be scheduled for parallel execution on a single multiprocessor.

While an obvious solution to best utilize the multiprocessor resources is to launch a large enough block to utilize all available compute units a few times over, it might not always lead to the best possible performance. While a large block allows the multiprocessor to hide memory latencies by computing other warps while one is waiting, smaller concurrently executed blocks can also hide blockwise synchronization overhead by executing threads from the other block.

Some GPU, multiprocessor (SMX) and thread resource limitations are listed in table 2.

Scope	Resource	Amount
GPU	Multiprocessors	14
GPU	CUDA Cores	2688
GPU	Global Memory	6144 MB
GPU	Max Grid Size (x, y, z)	2147483647, 65535, 65535
SMX	Max Threads	2048
SMX	Shared Memory	64 KB
SMX	32-bit Registers	65536
SMX	L1 Cache + Shared Mem.	64 KB
Block	Max Threads	1024
Block	Max Dimensions	1024, 1024, 64
Thread	32-bit Registers	255

Table 2: Some of the resource limitations of Nvidia Geforce GTX Titan GPU.

Just as the goal of the block size and resource usage optimization is to keep the single multiprocessor busy, similarly the grid size is chosen to keep the entire GPU busy. Thus, the number of blocks in the grid should be large enough to schedule at least as many blocks for each multiprocessor as required to fully utilize its resources, although there is no penalty on launching larger grids. This means the number of blocks should essentially be as large as possible within the constraints set by the hardware and data since this also allows better scaling for future devices.

The threads, blocks and grids are the building elements for the programmer and map to the hardware on multiprocessor level. However, the execution of the threads in a block assigned to a multiprocessor is scheduled in *warps*, which on current hardware is a group of 32 threads. This is important to keep in mind when defining the block size, which should usually be a multiple of the warp size for higher occupation and coalesced memory accesses, which is better explained in Section 6.4.

CUDA *kernels* are essentially C functions, which are executed logically in parallel for all threads in the accompanying grid and block structure.

On hardware with compute capability 2.0 or greater, the multiprocessors can execute multiple kernels concurrently with the maximum of 32 simultaneous kernels for compute capability 3.5 device. While not used in our application, this allows splitting the computation into multiple kernels where it is necessary to simplify the code or for dynamic kernel launching, and to create implicit synchronization points without large performance degradation

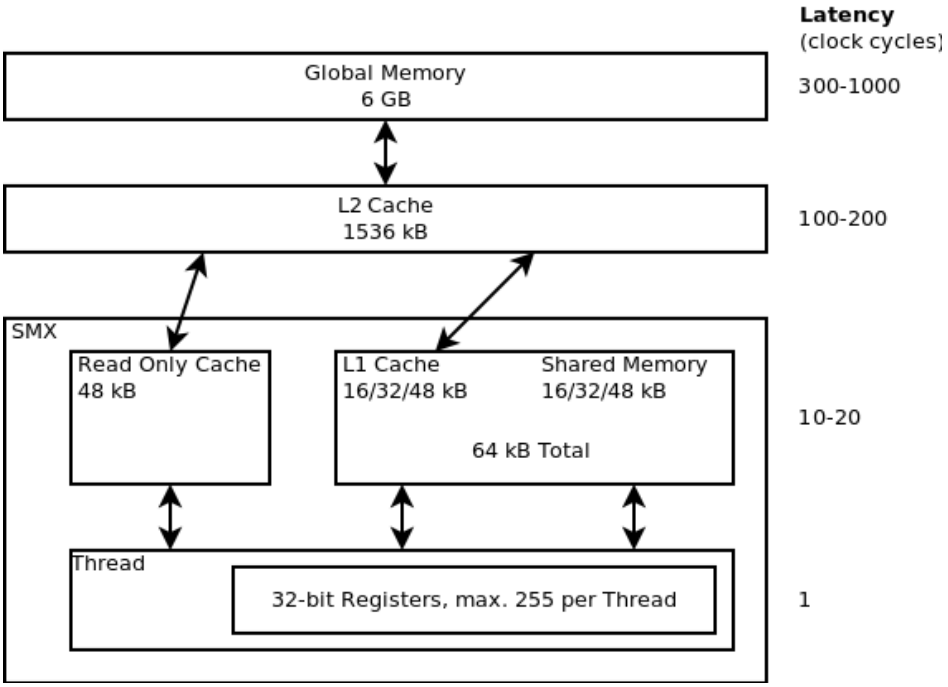

Figure 4: Kepler memory hierarchy and latencies.

from idle SMXs. It also allows the execution of the less parallel parts of the code on the GPU, while the more parallel calculations are running, potentially avoiding slow memory transfers between the GPU and CPU memory.

6.4 CUDA Memory Organization

The memory organization scheme in CUDA and Titan GPU requires a bit more care from the programmer than modern CPU programming. The user needs to be aware of the performance and limitations of different memory areas, an overview of which is shown in Figure 4. While modern CPU memory organization is also complex, the platform is more mature and usually takes more of the memory management burden from the programmer. Achieving similar level of transparent optimization – if even possible for massively parallel processing – is at least a few software and hardware generations away.

From the programmer's perspective the most important distinction is between the on-chip and off-chip memory. The registers and shared memory reside on the SMX chip, and are divided between all blocks concurrently

executing on the same SMX and allow access within few multiprocessor clock cycles. These memory areas are fairly small, with 64K 32-bit registers per multiprocessor and 255 registers per thread, as well as 64K on-chip memory which is split between L1 cache and shared memory.

The off-chip memory areas include global, per-thread local, constant and texture memory. Of these, the most important for our application is the global memory, which – as the name suggests – is accessible for any thread. It is the main memory area in which the GPU processed data resides. Because accessing the global memory takes up to several hundred clock cycles, it is important to get coalesced memory reads and writes in order to minimize the number of memory accesses and maximize the throughput.

Non-coalesced memory accesses can severely reduce the memory bandwidth, as the cached reads and writes will handle a full 128-byte blocks regardless of the number of actually required bytes within that block. This is illustrated in Figure 5.

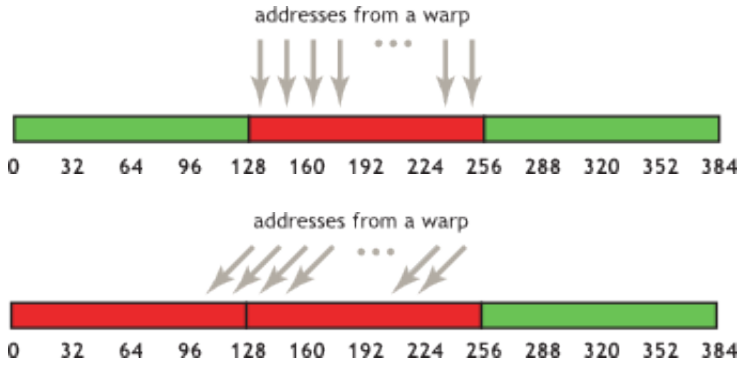

Figure 5: The Effect of aligned and misaligned memory accesses. The aligned access reads the whole 128-byte memory segment in one L1 transaction, while misaligned access requires two transactions [46].

7 GP-UCB Implementation

As both Gaussian process literature and CUDA terminology use the word kernel, the word *kernel* is used when referring to Gaussian process kernels and *CUDA kernel* is used when referring to GPU implementations.

We made two different implementations for running the CUDA code from the Python environment used for the rest of the application. Our first implementation consisted of two parts: Python code and CUDA kernels written in CUDA C. Python code was used to do higher level processing and initialize the PyCUDA environment, which was then used to call the actual CUDA kernels. Library functions were used for all operations other than the distance kernel calculation. Due to problems with interoperability between PyCUDA and Django, the PyCUDA code had to be run as a separate process.

Given the easy implementation of C-Python interfaces, it is then a fairly small step to implement the CUDA code as a separate CUDA C program and simply call that from the Python code. At the time of implementation the PyCUDA had limited support for the features required to implement concurrent memory transfers and kernel execution, which was one reason for switching to pure CUDA C for the third implementation. Other reason for the switch was speed, as the new version does more work outside the CUDA kernel.

Our implementation of Gaussian process focuses on using it for prediction in the UCB algorithm with small number observations. This allows us to avoid some calculations and memory overhead, especially on kernel function matrix computations. Combined with the fact that we only need to calculate the variance instead of the full covariance matrix, the required amount of memory and processing is much smaller than with general purpose Gaussian process implementation.

It is also possible to speed up the Gaussian process calculations by storing a precalculated covariance matrix and simply look up the results. In our application the cost saving measures used above would not apply, and we would need to store the full covariance matrix. Storing a covariance matrix for 500 000 data points using 32-bit floats would require almost one terabyte of memory, which is a prohibitively large amount on current desktops and requires specialized hardware. Even SSDs would be of limited use on large, low dimensional datasets which our implementation can process

quickly. Precalculated covariance matrix would also be far less flexible, as our implementation allows using different kernel functions or parameters between iterations.

Both the Python/PyCUDA and pure CUDA C versions of the algorithm have the same overall structure:

1. Read dataset into CPU memory.
2. Transfer dataset into GPU memory.
3. Upon receiving user feedback, transfer the features of the observed items into GPU memory.
4. Calculate the kernel function matrix between the observed items and the dataset.
5. Calculate everything else on the CPU.
6. Return mean and variance.

7.1 Time requirements

The two sets of calculations done in our Gaussian process implementation are

$$\bar{\mu}_t = K_*[K + \sigma_n^2 I]^{-1}\bar{y}_t$$
$$\bar{\sigma}_t^2 = \mathrm{diag}(K_{**} - K_*^T[K + \sigma_n^2 I]^{-1}K_*),$$

Here $\bar{\mu}_t$ and $\bar{\sigma}_t^2$ are the vectors of mean and variance values for all the elements of $\mathcal{X}$ at time t, while $\bar{y}_t$ is the vector of all relevance scores received so far. K is a shorthand for the covariance matrix between the observed elements of $\mathcal{X}$, K_* is the covariance matrix between the observed and unobserved elements and K_{**} the covariance matrix between unobserved elements of $\mathcal{X}$.

In the following, the variables n and m are the number of unobserved and observed elements, while d is the data dimensionality.

As we only use the variance, there is no need to calculate the full K_{**} matrix. Instead, a vector of n random numbers drawn from the normal distribution is used to simulate the diagonal in the matrix subtraction. We can also avoid the last matrix multiplication in $K_*^T[K + \sigma_n^2 I]^{-1}K_*$, which

would create an $n \times n$ matrix, and calculate only the diagonal of the result instead.

Calculating the $\bar{\sigma}_t^2$ requires two covariance matrix calculations, one matrix inversion and three matrix multiplications. The time complexity of calculating the K matrix with our distance kernel is $\mathcal{O}(m^2 d)$, while the time complexity of K_* calculation is $\mathcal{O}(nmd)$. Matrix inversion is $O(m^3)$. Since $m \ll n$ in our application, the computation of K_* is in reality significantly more time consuming than any other part of the computation.

In our implementation the small size of the matrix inversion allows us to use NumPy implementation of matrix inverse calculation [47] for calculating $K(X, X) + \sigma_n^2 I]^{-1}$ without any significant performance degradation. We also use library functions for other matrix multiplication except for the last operation in $K_*^T[K + \sigma_n^2 I]^{-1} K_*$ calculation, where we only compute the diagonal. The actual running times for different parts of the algorithm are analyzed in Section 8.

7.2 CUDA kernel implementations

The first implementation of the distance function for $K(X, X_*)$ calculation is shown in algorithm 3. It calculates `x`, `y` and `z` indices using the block index, block dimensions and thread index. The algorithm uses the full image feature matrix `feat` for the calculations. To extract the training and prediction sets from the image matrix it is indexed using index vectors `obs-idx` and `unobs-idx`, which in turn are indexed using `x` and `y` respectively.

Each thread on GPU calculates the distance between values `data[obs-idx[x]][z]` and `data[unobs-idx[y]][z]` divided by `dims` and adds it to the result matrix `K_x[x][y]`.

Because multiple threads would be writing to the same location in the result matrix, this has to be done using atomicAdd operation. There is no explicit copying to shared memory, as each of the original values are only used once per processed block.

The first version shows how reasonably efficient CPU code can be very inefficient when mapped directly to the GPU. The three loops of the CPU code – over observed datapoints, unobserved datapoints and dimensions – are replaced by three-dimensional grid and block structure. A single thread calculates the distance between one value in the observed data vector and unobserved data vector. It then divides the result by the number of

Algorithm 3: Initial code for the CUDA kernel used to calculate $K(X_*, X)$.

```
Data: feat, obs-idx, unobs-idx, n-obs, n-unobs, dims
Result: K_x
x = blockIdx.x * blockDim.x + threadIdx.x;
y = blockIdx.y * blockDim.y + threadIdx.y;
z = blockIdx.z * blockDim.z + threadIdx.z;
if (z > dims || y > n-unobs || x > n-obs) return;

atomicAdd(&K_x_gpu[y * n-shown + x],
    fdividef(
        fabsf(
            feat[obs-idx[x] * dims + z] - feat[unobs-idx[y] * dims + z]
        ), dims));
```

dimensions and adds it to the value in the result matrix cell. The calculation of $K(X, X)$ could be implemented using the same kernel function, simply using `obs-idx` and `n-obs` in the place of `unobs-idx` and `n-unobs`.

The major problems with this implementation were:

- Repeated uncoalesced reads and writes to the result matrix with atomicAdd operation. The atomicAdd in itself is not the cause of the poor performance, but instead the memory bandwidth is limited by the repeated uncoalesced global memory accesses.
- Unnecessary floating point division on each write to the result matrix.

The calculation of the diagonal of the $\text{cov}(\mathbf{f}_*)$ matrix is also implemented as a CUDA kernel. It is a straightforward modification of the basic matrix multiplication, which can be found in CUDA samples provided by Nvidia. It uses the thread x coordinate, calculated as shown in Algorithm 3, to index the result vector and a for loop for calculating the result for each index. As its processing takes less than one percent of the total time, no further optimization was done.

The listed problems were fairly easy to correct for the second version of the code (GPUv2) shown in Algorithm 4. Changing matrix to column-based made it simple to loop over dimensions while retaining coalesced memory access pattern. Using a temporary register variable for intermediate result removed the need for repeated global memory access, thus minimizing the

issue of the race condition. Nevertheless, register memory bank conflicts may affect performance so the loop size was set to equal the number of data points in the data chunk being processed to remove race condition between threads.

Algorithm 4: CUDA kernel code for GPUv2 algorithm. Variables x and y indicate the starting index in the two data matrices, and they are also used to calculate the index for the result in matC. Variables i and j loop over the matA and matB in column size loops.

```
Data: matA, matB, matC, n, m, cols
Result: K_x
x = blockIdx.x * blockDim.x + threadIdx.x;
y = blockIdx.y * blockDim.y + threadIdx.y;
chunksize = n * cols;
i = x;
j = y;
tmp = 0;
for (; i < chunksize; i += n, j += m) do
    tmp += fabs(matA[i] - matB[j]);
end
matC[x * m + y] = fdividef(tmp);
```

Assigning each chunk a number of data points that splits evenly to warps gives correctly aligned coalesced memory access for the chunk size loop in the kernel. Chunking makes it easy to avoid data padding or other methods often used to achieve warp-size aligned data as we can simply process the non-aligned last chunk on CPU if necessary. Due to the asynchronous nature of the GPU computation, this can be done in parallel while GPU is processing the rest of the data. It also allows easy scaling to multiple GPUs or nodes.

8 Results

The most computationally intensive part of the algorithm is calculating the distances between observed and unobserved points of data. It is the main focus of the optimization and will be covered more thoroughly. Comparisons for the scaling between matrix inversion and distance calculations are also shown, as well as total running times over data dimensionality, number of observations and total size of the data.

Testing was done on a server with Intel Xeon processor with two Nvidia

GeForce GTX Titan GPUs, although only one was used by the GPU algorithm.

The images used for testing were taken from MIRFLICKR 1 million image set [48]. The 4096-dimensional feature data was extracted using Overfeat deep convolutional neural network [28]. Due the hardware memory constraints only part of the images and feature data was used in the experiments.

8.1 Computation time

The single core CPU speed comparison baseline for calculating the $K(X_*, X)$ matrix was obtained by running the distance calculation using Python Scipy package. While the scipy.spatial.distance.cdist function is not the fastest available, it is reasonably well optimized and significantly faster than a naive C implementation.

The initial GPU distance implementation (GPUv1) was already significantly faster than the baseline CPU code despite being fairly inefficient. It did a float division for each dimension of the input vectors, as well as an atomic addition due to the fact there were multiple threads writing into the same result cell.

In the second version (GPUv2) the data is stored in a column first format and the kernel accesses the data one column at a time. This allows easy cache line aligned coalesced memory accesses and eliminates access conflicts when writing the results into the matrix. Using a register variable for storing the intermediate results removes the need for atomic operations and repeated global memory accesses. Floating point division is only done once for each result cell, just before writing the result from temporary register variable into the result matrix.

The second version also splits the data into smaller chunks to allow concurrent data transfers and computations on the GPU as well as calculates part of the data concurrently on the CPU if the data does not split evenly into defined chunk size. By defining the chunk size to be a multiple of the block size, it also avoids the requirement for data padding or other handling of the non-blocksize data on GPU further simplifying the GPU code implementation.

The performance of GPUv2 also depends on how the data is stored in the main memory. As the image feature data set is static in our case, it is possible to order it in a fashion where each chunk of the image feature data

is stored contiguously. This significantly improves the performance of the data transfer from main memory to GPU memory.

If the data was stored in the original column-wise format, each feature for all images would be stored contiguously in the main memory. This would require strided memory access, e.g. after transferring first feature of the first image chunk, the algorithm would need to skip over the first feature of all the images not in the chunk in order to reach the second feature for the images in the chunk. By storing all the features for each chunk contiguously we avoid the need for strided memory access.

A comparison of the calculation times between the single core CPU implementation and the GPU versions can be seen in Table 3. Scaling for both versions is very close to linear, with GPUv1 being consistently over eight times faster than the single core CPU version and GPUv2 around 40 times faster than the CPU and nearly five times faster than the GPUv1. The performance difference between other versions and GPUv2 increase as the data size increases, as the increased calculation time dominates over the time required for data transfer between CPU and GPU memory.

Data points	cdist (s)	GPUv1 (s)	GPUv2	Speedupv1	Speedupv2
2000	1.219	0.145	0.035	8.40	34.83
5000	3.211	0.386	0.088	8.32	36.49
10000	6.520	0.803	0.165	8.12	38.80
20000	13.182	1.628	0.333	8.10	39.59
30000	19.832	2.470	0.491	8.03	40.39
60000	39.803	4.902	0.931	8.120	43.22
120000	92.25	10.20	1.996	9.04	46.21

Table 3: A comparison of the $K(X_*, X)$ distance matrix calculation CPU and GPU running times.

This speedup allows using larger datasets while doing the calculations almost in real time, which is important in our application. Since the speed scaling is linear over the number of required calculations, i.e. it is linear over features as well as over the number of data points, this technique allows scaling up to millions of data points on lower-dimensional data.

The scaling is also close to linear over the number of observations up to the point where pseudoinversion of the $K(X, X)$ matrix starts dominating the calculation time. Compared to the distance calculation of the $K(X, X)$ matrix on CPU the turning point is around 4100 observations, as can be

seen from Figure 6. This can be further improved by calculating the matrix pseudoinverse on GPU, which can be done using CUDA 7 cuSOLVER library QR factorization routines.

In practice the time used for pseudoinverse is never large enough to be a concern in our application. It would dominate the calculation time only if the size of the data matrix was small compared to the number of observations, which in our case can only happen with a particularly persistent user willing to go through very large number of images in order to locate the right one.

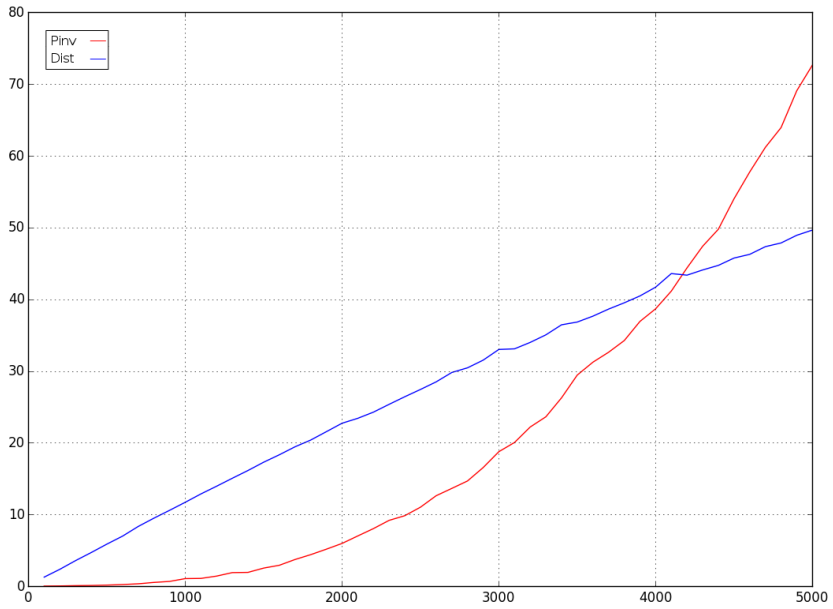

Figure 6: Running times for the $K(X_*, X)$ distance matrix calculation and pinv$(K(X, X))$ calculation in seconds. Pseudoinverse in red and distance matrix calculation in blue.

As can be seen, the running time of the $K(X_*, X)$ grows linearly over the number of observations, while the pseudoinverse running time growth is quadratic. Obviously the actual running time for the pseudoinverse calculation is negligible in our use scenario, where the number of observations is usually much less than 1000.

8.2 Accuracy

Although numerical accuracy and the absolutely correct order of the results is not critical for our application, overly large variation from a more accurate implementation might still give less optimal results and thus degrade the user experience. As such it is important to compare the performance with a reference implementation to make sure the results remain within reasonable distance from the optimal results.

Since the results are calculated from the initial data on each loop, there is no chance of floating point errors accumulating over multiple rounds. This means we only need to consider the errors in the initial distance matrix calculations and their effect on the Gaussian process results.

8.2.1 GPU Kernel accuracy

To quantify the error, the accuracy of the calculated distances was compared to results obtained with Python SciPy function scipy.spatial.distance.cdist [49]. I will only consider the case with Overfeat output, for which the starting values are the default printed decimal output of the Overfeat program with 6 digits of precision. The results are only calculated for multiples of the chunk size, as non-evenly splitting data would have been calculated partially on the CPU with the GPUv2 implementation.

The following precision and accuracy results only apply if the values being represented lie within the range of normal values of the IEEE 754 floating point format, which is shown in Figure 8.2.1. In addition to values above the floating point maximum and minimum, they also does not apply to subnormal floating point numbers near zero, i.e. values with zero exponent and no implicit leading 1-bit.

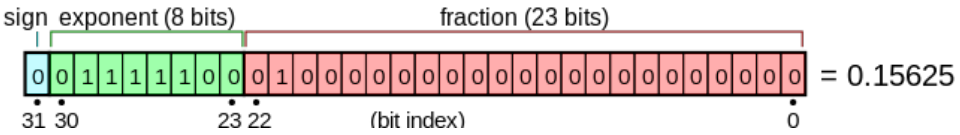

Figure 7: IEEE 754 32-bit floating point number example [50].

There is a possible rounding error – known as wobble – when converting decimal numbers to binary and vice versa, as well as in any floating point calculation [51]. Since wobble can waste almost a full bit of precision when

converting decimal to binary, uniquely representing 6 digits in normal precision binary float requires up to $6 * \frac{\log 10}{\log 2} + 1 \approx 20.93$ bits. This unique representation does not mean the decimal number could be exactly represented, merely that same binary representation is not shared by any two decimal numbers in the input space i.e. there is enough *precision* to uniquely represent the number but the representation is not necessarily *accurate*.

This means the size of the operands in addition and subtraction can differ up to 3 bits or $3 * \frac{\log 2}{\log 10} \approx 0.9$ decimal places before loss of precision in 32-bit floating point addition – and subtraction, which is in principle similar to addition but with sign bit inverted – can potentially happen.

The error in the addition and subtraction, and thus the error in sum accumulation as well, stems from the way floating point addition works. In order to add two floating point numbers together, they are shifted so that the corresponding bits line up. If one of the values is significantly larger than the other, limited size of the mantissa may cause some of the least significant bits in the smaller value to be discarded. While this can happen in a single addition operation, summation using an accumulator variable is especially prone to this kind of error.

There are various methods to minimize the error, such as increasing the precision of the floating points by using doubles or using Kahan summation algorithm [52] or exact floating point summation with faithful rounding [53]. We did not use any additional methods for minimizing the error, as they are also significantly slower to run on a GPU and, with the exception of simply converting floats to doubles, also harder to implement.

There are two locations in the algorithm 4 where loss of precision is possible – subtraction and summation. If the `matA[i]` and `matB[j]` magnitude difference is large enough, some of the least significant bits may be discarded in the subtraction. It is also possible that the accumulation of the sum to a temporary variable in the for loop can also cause accumulation of a large floating point error.

The loss of precision in the subtraction is less critical for our algorithm; if the magnitude of the operands – and thus the distance between two data points in that dimension – was large enough to cause loss of precision, the least significant bits are unlikely to contain critical distance information.

The error accumulating in the summation is worse, as the difference between the accumulator variable and summands grows as the dimensionality

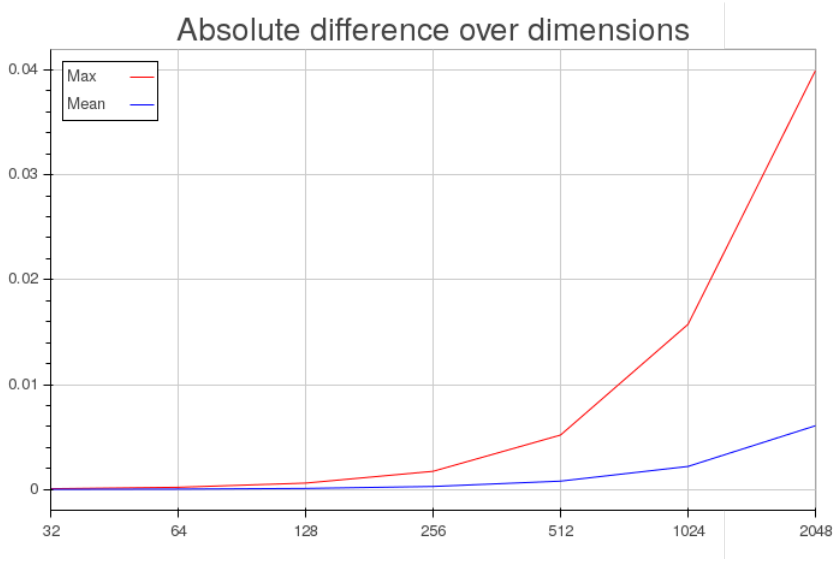

Figure 8: The maximum and mean relative difference between SciPy calculated and GPU calculated distances over the number of dimensions, with 4096 data points and 128 observations.

of the data increases. For example, if all summands are close to 1 in a 4096-dimensional vector, the final addition in the summation is roughly $4095 + 1$. The difference in floating point exponent is $139 - 127 = 12$ and the mantissa of the result has to be shifted accordingly, which means the least significant bits will be lost. This loss of precision can be seen in the empirical results in Table 4 and Figures 8 and 9.

The SciPy cdist function calculating Manhattan distance also does summation in naive fashion, but given the limited accuracy of the initial values and the comparatively small number of features, 64-bit floating point values (double precision) are enough to store full precision of the summation results. Compared to a 32-bit floating point with 23-bit mantissa, the double precision floating points have 52-bit mantissa and can uniquely represent at least 15 significant decimal digits as opposed to the at least 6 digits of the single precision format.

SciPy also allows using NumPy float128 data type, which on Intel x86 architectures is an extended precision double – a padded 80-bit floating point number [54, 55, 56]. This 80-bit floating point format has 15 bit exponent and 64 bit mantissa. Because the accuracy of doubles is sufficient to store the full precision of our summation results, we get – within the required

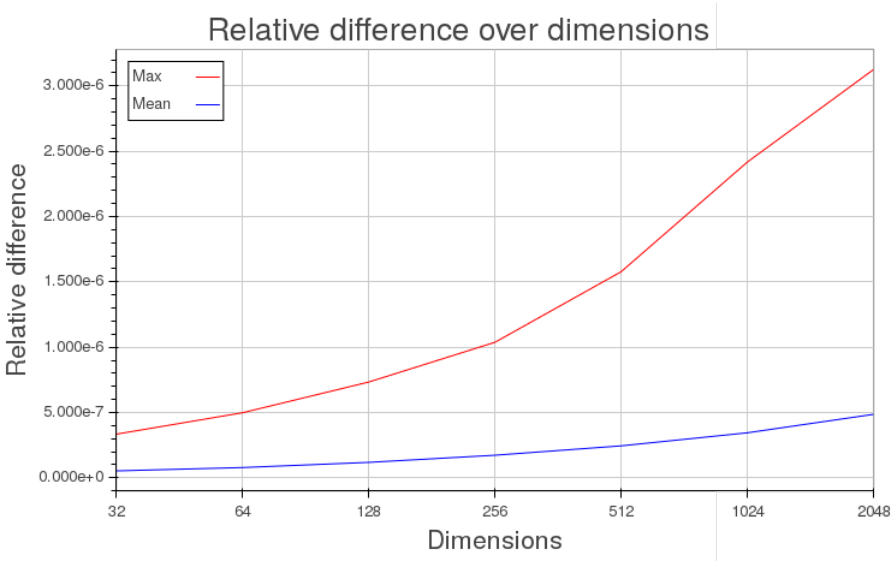

Figure 9: The maximum and mean relative difference between SciPy calculated and GPU calculated distances. The relative difference is $\frac{|x_i - y_i|}{\max X}$, where $x \in X$ are the distance values calculated using doubles in SciPy, $y \in Y$ are calculated using GPUv2 and i are indices $[0, |X|]$. Results were calculated with 4096 data points and 128 observations.

precision – exactly matching distance calculation results using doubles and extended precision doubles.

Dims	Max diff	Mean diff	Max rel diff	Mean rel diff
64	6.1035156e-05	1.0306016e-05	4.7683716e-07	9.0542017e-08
128	0.00024414062	3.6993064e-05	6.5565109e-07	1.2231612e-07
256	0.00073242188	0.00012575462	1.0728836e-06	1.9761592e-07
512	0.0017089844	0.00034622755	1.4901161e-06	2.9111106e-07
1024	0.0053710938	0.00094844773	2.0861626e-06	3.8137023e-07
2048	0.013671875	0.0026632845	2.5033951e-06	4.8529182e-07
4096	0.047851562	0.0083170682	4.3511391e-06	7.493536e-07

Table 4: The difference between GPUv2 results and SciPy results for 1024 data points and 64 observations. Max and mean diff are the absolute differences between calculated results. Max and mean rel diff are calculated as $\frac{|x_i - y_i|}{\max X}$, where $x \in X$ are the distance values calculated using doubles in SciPy, $y \in Y$ are calculated using GPU and i are indices $[0, |X|]$.

8.2.2 Gaussian Process Accuracy

The distance matrix calculation is just the first step in computing the Gaussian process result for UCB algorithm. We also need to take into account the possible error accumulating from the matrix operations in the mean and covariance calculations in equations

$$\bar{\mathbf{f}}_* = \mathbb{E}[\mathbf{f}_*|\mathbf{y}, X_*, X] = K(X_*, X)[K(X, X) + \sigma_n^2 I]^{-1}\mathbf{y},$$
$$\text{cov}(\mathbf{f}_*) = K(X_*, X_*) - K(X_*, X)[K(X, X) + \sigma_n^2 I]^{-1}K(X, X_*).$$

It turns out that the matrix multiplication in NumPy uses a BLAS implementation – in our case ATLAS library – which have quite strict numerical stability requirements [57, 58, 59]. Given the limited accuracy of our starting values and the fact the matrix operations are done using 64-bit values, they do not add significant error to the result.

Figures 10 and 11 shows the effect of the single precision floating point calculation error compared to a double precision implementation in the output to the user. These figures show the difference in indices of the sorted results with the SciPy and GPUv2 algorithms. If a data point with the highest score in SciPy result has the second highest score in GPUv2 result, it would have index difference of one – and correspondingly there would be at least one other data point with index difference of at least one. These results were calculated on actual data with 128 observations. The difference over number of dimensions was calculated using 4096 points of data, while difference over data size was calculated using 4096 dimensions. There is a slight difference in the order of the images, which may affect results shown to the user, but given the low mean index difference it is unlikely to cause significantly worse result than using a more precise algorithm.

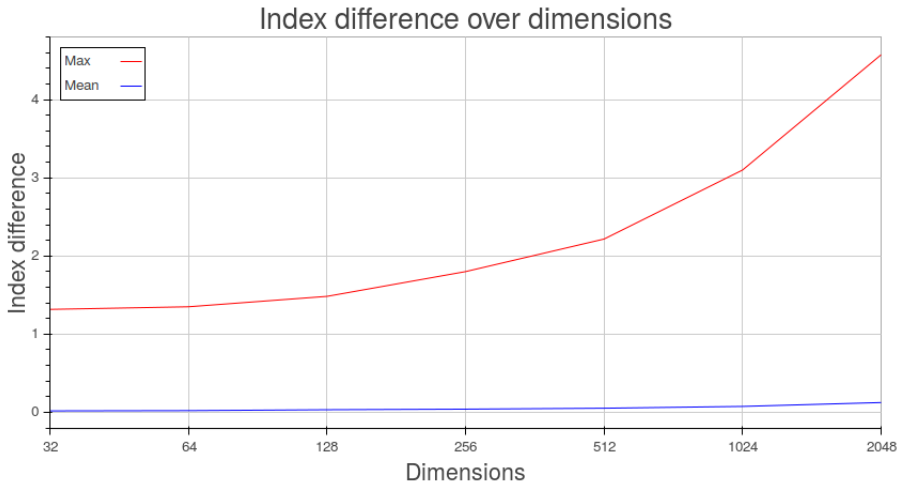

Figure 10: The maximum and mean difference in sorted data point indices between SciPy calculated and GPU calculated Gaussian process, where score for data point $x = \mu + \sigma$.

Figure 11: The maximum and mean difference in sorted data point indices between SciPy calculated and GPU calculated Gaussian process, where score for data point $x = \mu + \sigma$.

9 Conclusions

In this thesis I have presented a simple method for accelerating Gaussian process upper confidence bound algorithm with graphics processing unit in order to allow larger image datasets in our content based image retrieval system. Even with a simple, fairly unoptimized GPU implementation using a powerful – but still consumer grade – GPU we were able to use the system with 30 to 40 times more data than with the same computer setup without the GPU.

Using only a single CPU core for the Gaussian process calculations our implementation is able to process 3000 image datasets with 512 features per image in around 2 seconds, which might still be considered reasonable waiting time in an application. With a GPU implementation of the Gaussian process we can process 120000 images in roughly the same time – 40 times the amount of a single core CPU implementation. It would be possible to further increase the performance by balancing the calculations between a multicore CPU implementation and the GPU, but due to time constraints it was not possible for this version.

It is also apparent that care must be taken when implementing algorithms on GPU. While similar measures are important on CPU algorithms as well, even greater care must be taken on GPU where the implicit parallel process may hide the points of error accumulation. It may also be tempting to make a 32-bit implementation, especially with consumer grade GPUs which have large difference between 32-bit and 64-bit floating point performance.

Using 32-bit calculations is perfectly viable in applications where numerical accuracy is not critical and the algorithm does not reuse the previous results in a fashion that would cause accumulation of large floating point errors. Even then the effect of reduced precision is noticeable and might actually become a problem with very high dimensional data.

In order to allow processing even larger sets of image data, future work for this algorithm should include a distributed verson. While the structure of the GPUv2 algorithm makes moderately sized distributed implementation relatively straightforward, there are lot of details that need to be considered. The algorithm should also be combined with other search methods in order to further limit the search space and refine the results.

References

[1] D. Głowacka and S. Hore, "Balancing exploration–exploitation in image retrieval," in *Proceedings of UMAP*, 2014.

[2] S. Hore, L. Tyrvainen, J. Pyykko, and D. Glowacka, "A reinforcement learning approach to query-less image retrieval," in *Symbiotic Interaction*, pp. 121–126, Springer, 2014.

[3] J. Z. Wang, N. Boujemaa, A. Del Bimbo, D. Geman, A. G. Hauptmann, and J. Tesić, "Diversity in multimedia information retrieval research," in *Proceedings of the 8th ACM international workshop on Multimedia information retrieval*, pp. 5–12, ACM, 2006.

[4] A. W. Smeulders, M. Worring, S. Santini, A. Gupta, and R. Jain, "Content-based image retrieval at the end of the early years," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 22, no. 12, pp. 1349–1380, 2000.

[5] D. Glowacka and J. Shawe-Taylor, "Content-based image retrieval with multinomial relevance feedback," in *Proceedings of ACML*, 2010.

[6] I. F. Rajam and S. Valli, "A survey on content based image retrieval," *Life Science Journal*, vol. 10, no. 2, pp. 2475–2487, 2013.

[7] J. Domke and Y. Aloimonos, "Deformation and viewpoint invariant color histograms.," in *BMVC*, pp. 509–518, 2006.

[8] T. Gevers and H. Stokman, "Robust histogram construction from color invariants for object recognition," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 26, no. 1, pp. 113–118, 2004.

[9] X. Wang, J. Wu, and H. Yang, "Robust image retrieval based on color histogram of local feature regions," *Multimedia Tools and Applications*, vol. 49, no. 2, pp. 323–345, 2010.

[10] R. Min and H. Cheng, "Effective image retrieval using dominant color descriptor and fuzzy support vector machine," *Pattern Recognition*, vol. 42, no. 1, pp. 147–157, 2009.

[11] J. Huang, S. R. Kumar, M. Mitra, W. Zhu, and R. Zabih, "Image indexing using color correlograms," in *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*, pp. 762–768, IEEE, 1997.

[12] K. E. Van De Sande, T. Gevers, and C. G. Snoek, "Evaluating color descriptors for object and scene recognition," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 32, no. 9, pp. 1582–1596, 2010.

[13] D. A. Clausi, "An analysis of co-occurrence texture statistics as a function of grey level quantization," *Canadian Journal of remote sensing*, vol. 28, no. 1, pp. 45–62, 2002.

[14] N. Suematsu, Y. Ishida, A. Hayashi, and T. Kanbara, "Region-based image retrieval using wavelet transform," in *Proc. 15th international conf. on vision interface*, pp. 9–16, Citeseer, 2002.

[15] L. Soh and C. Tsatsoulis, "Texture analysis of sar sea ice imagery using gray level co-occurrence matrices," *Geoscience and Remote Sensing, IEEE Transactions on*, vol. 37, no. 2, pp. 780–795, 1999.

[16] E. Gadelmawla, "A vision system for surface roughness characterization using the gray level co-occurrence matrix," *NDT & E International*, vol. 37, no. 7, pp. 577–588, 2004.

[17] A. Krizhevsky and G. E. Hinton, "Using very deep autoencoders for content-based image retrieval.," in *ESANN*, Citeseer, 2011.

[18] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Applied statistics*, pp. 100–108, 1979.

[19] U. Von Luxburg, "A tutorial on spectral clustering," *Statistics and computing*, vol. 17, no. 4, pp. 395–416, 2007.

[20] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006.

[21] C. Rao, S. S. Kumar, B. C. Mohan, *et al.*, "Content based image retrieval using exact legendre moments and support vector machine," *arXiv preprint arXiv:1005.5437*, 2010.

[22] R. Liu, Y. Wang, T. Baba, D. Masumoto, and S. Nagata, "Svm-based active feedback in image retrieval using clustering and unlabeled data," *Pattern Recognition*, vol. 41, no. 8, pp. 2645–2655, 2008.

[23] M. Koskela and J. Laaksonen, "Convolutional network features for scene recognition," in *Proceedings of the ACM International Conference on Multimedia*, pp. 1169–1172, ACM, 2014.

[24] A. Karpathy and L. Fei-Fei, "Deep visual-semantic alignments for generating image descriptions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3128–3137, 2015.

[25] J. Wan, D. Wang, S. C. H. Hoi, P. Wu, J. Zhu, Y. Zhang, and J. Li, "Deep learning for content-based image retrieval: A comprehensive study," in *Proceedings of the ACM International Conference on Multimedia*, pp. 157–166, ACM, 2014.

[26] J. E. Sklan, A. J. Plassard, D. Fabbri, and B. A. Landman, "Toward content based image retrieval with deep convolutional neural networks," in *SPIE Medical Imaging*, pp. 94172C–94172C, International Society for Optics and Photonics, 2015.

[27] X. S. Zhou and T. S. Huang, "Relevance feedback in image retrieval: A comprehensive review," *Multimedia systems*, vol. 8, no. 6, pp. 536–544, 2003.

[28] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "Overfeat: Integrated recognition, localization and detection using convolutional networks," *arXiv preprint arXiv:1312.6229*, 2013.

[29] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, pp. 237–285, 1996.

[30] M. N. Katehakis and A. F. Veinott Jr, "The multi-armed bandit problem: decomposition and computation," *Mathematics of Operations Research*, vol. 12, no. 2, pp. 262–268, 1987.

[31] T. Lai and H. Robbins, "Asymptotically efficient adaptive allocation rules," *Adv. Appl. Math.*, vol. 6, pp. 4–22, Mar. 1985.

[32] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Mach. Learn.*, vol. 47, pp. 235–256, May 2002.

[33] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[34] N. Srinivas, A. Krause, S. M. Kakade, and M. Seeger, "Gaussian process optimization in the bandit setting: No regret and experimental design," *arXiv preprint arXiv:0912.3995*, 2009.

[35] K. Chaloner and I. Verdinelli, "Bayesian experimental design: A review," *Statistical Science*, pp. 273–304, 1995.

[36] A. Krause and C. E. Guestrin, "Near-optimal nonmyopic value of information in graphical models," *arXiv preprint arXiv:1207.1394*, 2012.

[37] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher, "An analysis of approximations for maximizing submodular set functions—i," *Mathematical Programming*, vol. 14, no. 1, pp. 265–294, 1978.

[38] L. Dorard, D. Glowacka, and J. Shawe-Taylor, "Gaussian process modelling of dependencies in multi-armed bandit problems," in *Proceedings of the 10th International Symposium on Operational Research SOR 09*, pp. 77–84, 2009.

[39] A. Medlar, D. Głowacka, H. Stanescu, K. Bryson, and R. Kleta, "Swiftlink: parallel mcmc linkage analysis using multicore cpu and gpu," *Bioinformatics*, vol. 29, no. 4, pp. 413–419, 2013.

[40] M. Flynn, "Some computer organizations and their effectiveness," *Computers, IEEE Transactions on*, vol. C-21, pp. 948–960, Sept 1972.

[41] NVIDIA Corporation, "Kepler gk110," 2012.

[42] CompuGreen, LLC, "The green500 list," 2015.

[43] PEZY Computing, "Pezy-sc many core processor," 2015.

[44] A. Petitet, C. Whaley, J. Dongarra, and A. Cleary, "Hplinpack," 2008.

[45] H. Gahvari, M. Hoemmen, J. Demmel, and K. Yelick, "Benchmarking sparse matrix-vector multiply in five minutes," in *SPEC Benchmark Workshop (January 2007)*, 2007.

[46] NVIDIA Corporation, "Cuda best practices guide," 2015.

[47] S. van der Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: A structure for efficient numerical computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[48] M. J. Huiskes and M. S. Lew, "The mir flickr retrieval evaluation," in *MIR '08: Proceedings of the 2008 ACM International Conference on Multimedia Information Retrieval*, (New York, NY, USA), ACM, 2008.

[49] E. Jones, T. Oliphant, P. Peterson, *et al.*, "SciPy: Open source scientific tools for Python," 2001–. [Online; accessed 2015-09-10].

[50] Fresheneesz, "Example of ieee 754 floating point number," 2007. [Online; accessed 2015-10-13].

[51] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991.

[52] W. Kahan, "Pracniques: further remarks on reducing truncation errors," *Communications of the ACM*, vol. 8, no. 1, p. 40, 1965.

[53] S. M. Rump, T. Ogita, and S. Oishi, "Accurate floating-point summation part i: Faithful rounding," *SIAM Journal on Scientific Computing*, vol. 31, no. 1, pp. 189–224, 2008.

[54] S. van der Walt, S. Colbert, and G. Varoquaux, "The numpy array: A structure for efficient numerical computation," *Computing in Science Engineering*, vol. 13, pp. 22–30, March 2011.

[55] Intel Corporation, "Intel c++ compiler 16.0 user and reference guide," 2015.

[56] IEEE Standards Association *et al.*, "Standard for floating-point arithmetic," *IEEE 754-2008*, 2008.

[57] N. J. Higham, "Exploiting fast matrix multiplication within the level 3 blas," *ACM Transactions on Mathematical Software (TOMS)*, vol. 16, no. 4, pp. 352–368, 1990.

[58] M. Badin, P. D'Alberto, L. Bic, M. Dillencourt, and A. Nicolau, "Improving the accuracy of high performance blas implementations using adaptive blocked algorithms," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2011 23rd International Symposium on*, pp. 120–127, IEEE, 2011.

[59] R. C. Whaley and A. Petitet, "Minimizing development and maintenance costs in supporting persistently optimized BLAS," *Software: Practice and Experience*, vol. 35, pp. 101–121, February 2005. `http://www.cs.utsa.edu/~whaley/papers/spercw04.ps`.