Fast Kullback-Leibler optimization algorithm:
software library implementation report

Eugène Suter

November 11, 2009
Contents

1 Introduction 1

2 Background 2
   2.1 Minimizing the Kullback-Leibler divergence with the steepest descent method 2
   2.2 The natural gradient applied to the steepest descent method 3
   2.3 Probabilistic control systems 4

3 Implementation 6
   3.1 Objectives 6
   3.2 Language and nomenclature 6
   3.3 Data structures 7
      3.3.1 Vectors 7
      3.3.2 Matrices 8
      3.3.3 Matrix arrays 9
      3.3.4 Structure representation 10
   3.4 Layout 10
   3.5 OpenMP 11
      3.5.1 Performance considerations 12
      3.5.2 Usage in the optimization library 13
   3.6 CUDA 15
      3.6.1 The architecture 15
      3.6.2 Programming CUDA-enabled devices 16
      3.6.3 Applying the CUDA model to the optimization library 17
      3.6.4 Summation on the CUDA architecture 18
   3.7 Numerical issues 18
      3.7.1 Finite precision arithmetic 18
      3.7.2 Impact on the optimization algorithm 19

4 Test cases 21
   4.1 Test 1 21
   4.2 Test 2 22
   4.3 Test 3 23
   4.4 Test 4 25

5 Benchmarks and performance analysis 27
   5.1 Hardware 27
   5.2 Program and test parameters 27
   5.3 Benchmarking the C library 28
      5.3.1 Compiler optimizations 28
      5.3.2 Fast logarithm 29
      5.3.3 Loop unrolling 32
   5.4 Benchmarking the CUDA library 35
6 API

6.1 Public interface ........................................ 37
6.2 Macros and enumerations .............................. 37
6.3 Functions ................................................. 37
  6.3.1 klopt_vector_init_index ............................. 37
  6.3.2 klopt_vector_set .................................. 38
  6.3.3 klopt_vector_print ................................. 38
  6.3.4 klopt_matrix_set .................................. 38
  6.3.5 klopt_matrix_copy ................................ 39
  6.3.6 klopt_matrix_print ............................... 39
  6.3.7 klopt_matrix_fn ................................... 39
  6.3.8 klopt_ng ............................................ 39
  6.3.9 klopt_icmd_rmajor and klopt_icmd_cmajor ........ 40
  6.3.10 klopt_cc_rmajor and klopt_cc_cmajor .......... 40
  6.3.11 klopt_sg_cmajor and klopt_sg_cmajor .......... 40
  6.3.12 klopt_ng_rmajor and klopt_ng_cmajor .......... 41
  6.3.13 klopt_ctrl_rmajor and klopt_ctrl_cmajor .... 41
  6.3.14 klopt_maxnorm_rmajor and klopt_maxnorm_cmajor .. 41
  6.3.15 klopt_ctg_rmajor and klopt_ctg_cmajor ........ 42
  6.3.16 klopt_optimize ................................... 42
  6.3.17 klopt_chain_optimize ............................ 42

7 Conclusions and closing remarks ......................... 44
Abstract

The software library this report discusses was developed as part of a bursary granted by the Instituto de Engenharia de Sistemas e Computadores - Investigação e Desenvolvimento (INESC-ID Lisboa), under the supervision of Miguel Barão.

The theoretical material this library is based on can be found in Miguel Barão’s doctoral thesis[2], and two conference papers[1, 3], one of which was co-authored with Professor João Miranda Lemos. Although this report will provide an overview of the optimization algorithm, the above mentioned sources contain the actual mathematical proofs and theorems that the algorithm is based on, and should therefore be consulted beforehand.

---

1Miguel J. S. Barão, PhD is with INESC-ID Lisboa and the Évora University’s Computer Science Department.

2João M. Lemos is with INESC-ID Lisboa and Department of Electrical Engineering, Instituto Superior Técnico.
1 Introduction

In this report we will discuss the optimization library developed, first and foremost, by providing an overview of the theoretical material it is based on (section 2). Section 3 will detail the library implementation, from tools used to optimize performance to problems encountered during its development, and finally the bundled test cases. Section 5 includes speed-testing of the library on current computer hardware and data providing insight into performance bottlenecks encountered. Section 6 provides a complete listing of the library’s public interface, including a description of the functions’ tasks, dependencies and limitations. The last section (7) will offer an outlook and closing remarks about the project.
2 Background

This section will provide an overview of the theory on which the optimization library is based.

2.1 Minimizing the Kullback-Leibler divergence with the steepest descent method

In Information Theory, the Kullback-Leibler divergence (also known as relative entropy), is a non-negative value measuring the “distance” or gap between two probability mass functions \( p(x) \) and \( q(x) \) and given by the following formula:

\[
D(p||q) = \sum_{x=1}^{n} p(x) \log \frac{p(x)}{q(x)}
\]  

(1)

Although the Kullback-Leibler divergence isn’t a true distance metric since it isn’t symmetric nor does it satisfy the triangle inequality\[6\], it is still useful to consider it a distance when visualizing a problem (see figure 1).

Figure 1: Visual interpretation of \( D(p||q) \) in \( \mathbb{R}^2 \)

Suppose, for example, that one would like to minimize the KL divergence between the two probability mass functions (p.m.fs) \( p(x) \) and \( q(x) \). Traditionally, this goal is achieved using the iterative steepest descent method (or some variation thereof), where at each iteration, a gradient vector is calculated and used to update \( p(x) \).

In \[4\], \( p(x) \) is mapped to a coordinate system \( \theta \) of \( \mathcal{P} \), a set of probability mass functions. \( p(x) \) has \( n \) parameters \( (\theta^0, \cdots, \theta^n) \) where parameter \( \theta^0 \) is equal to the unit minus the sum of the remaining parameters, since all the probabilities in a p.m.f. add to one. The gradient vector is calculated using the differential of a function \( F(\theta) \), such that at some point \( \theta \),

\[
\nabla F = \begin{bmatrix} \frac{\partial F}{\partial \theta^1} & \cdots & \frac{\partial F}{\partial \theta^n} \end{bmatrix}^T
\]

(2)

gives the direction of greatest change. Using the steepest descent method,

\[
\theta_k = \theta_{k-1} - \eta \nabla F
\]

(3)

\[
\theta_0^k = 1 - \sum_{i=1}^{n} \theta_i^k
\]

(4)
we step towards the target distribution \( q(x) \) until a certain stopping criterion is met. Unfortunately, using this standard gradient with a fixed step size is very unstable, as shown in figure 6. To obtain meaningful results, one of the following options can be implemented: (a) use an adaptive step size, where a new value for \( \eta \) is calculated at each iteration which, unavoidably, introduces extra computational overhead; (b) use a small step size so that the target isn’t “overshot” during the process, however this means convergence will be very slow.

2.2 The natural gradient applied to the steepest descent method

In [4], Barão and Lemos substitute the standard gradient in the minimization of \( D(p||q) \) with the natural gradient method,

\[
\nabla F = G^{-1} \nabla F
\]

where \( G \) is the Fisher information metric. Unfortunately this equation is fairly expensive to calculate directly, since using the \( G \) matrix creates a temporal complexity of \( O(n^3) \). There is, however, a mathematically equivalent equation that avoids using \( G \) when computing \( \nabla F \) which reduces the temporal and spatial complexity to \( O(n) \):

\[
\nabla F = \theta \circ \nabla F - \theta (\theta \cdot \nabla F)
\]

where \( \cdot \) represents the dot product and \( \circ \) denotes the Hadamard (or element-wise) product. Using the natural gradient in equation (3) instead of the standard gradient provides fast and stable convergence to the target p.m.f. using a fixed and fairly large step size (see figure 5).

This optimization method (in theory) assures that all the probabilities in \( p(x) \) will add to one, as per equation (4), they are not independent. In practice however, the software implementation is never ideal because the results are computed with finite precision and are subject to rounding errors which, left unchecked, may cause completely useless results to be produced. In section ?? this problem is discussed further, as it was encountered in one of the library’s test cases. In short, the use of equation (4) to calculate the first element of \( p(x) \) is not viable for very large systems because the accumulation of rounding errors during the summation may produce a value very close to one, such that the machine’s floating point unit sets the value of \( \theta^0 \) to exactly zero instead of a value close to zero.

This means that a method that calculates each of \( p(x) \)’s parameters independently is needed. Coincidentally in [3], Barão provides an alternative method in which the (modified) natural gradient, denoted by \( \nabla F' \) is calculated over the entire probability vector \( p \):

\[
\nabla F' = p \circ \nabla F' - p (p \cdot \nabla F')
\]

The modified standard gradient vector \( \nabla F' \) stays the same except for an extra first element, set to zero:

\[
\nabla F' = \begin{bmatrix}
0 \\
\frac{\partial F}{\partial \theta_1} \\
\vdots \\
\frac{\partial F}{\partial \theta_n}
\end{bmatrix}
\]
Equation (3) becomes

\[ p_k = p_{k-1} - \eta \nabla F' \]  

and equation (4) is no longer necessary. Although this alternative method was not developed with this numerical problem in mind, it does seem to have better numerical properties.

2.3 Probabilistic control systems

The algorithms presented so far can also be pressed into service optimizing probabilistic control systems in which a time-invariant Markov chain is used to represent a system model \( s_{t+1}(u_t, x_t) \) that contains a set of transition probabilities, from the current state \( x_t \) to a future state \( x_{t+1} \).

Establishing a feedback loop, \( u_t \) serves as an input to the system model and is signalled by an external controller matrix \( c(u_t|x_t) \) which influences the state transition from \( x_t \) to \( x_{t+1} \). In turn, the output from the system model is used to update the controller, and so forth. Given a fixed timeframe \( T > 0 \), this procedure is carried out for each \( t = T - 1 \) to \( t = 0 \) and in every individual iteration of \( t \), the controller is optimized using the natural gradient and steepest descent method. Once the controller is deemed to be sufficiently optimized, the algorithm moves on to the next value of \( t \).

Mathematically, this can be expressed as

\[ p(x_0:T, u_0:T-1) = p(x_0) \prod_{t=0}^{T-1} s(x_{t+1}|u_t, x_t)c(u_t|x_t) \]  

and where the KL divergence \( D(p||q) \) is calculated between (10) and an ideal joint p.m.f. \( q(x_0:T, u_0:T-1) \).

Of course, attempting to compute equation (10) as-is is impossible for large systems because the storage space required grows rapidly towards infinity as \( T \) increases. In [2, 3] a series of steps are provided to optimize an \( m \times n \) controller matrix for a user-supplied step size, system model, ideal system model (represented by \( S_t \)) and an ideal controller model (denoted by \( C_t \)). The latter two are vectors and can be seen as optimization “targets”. The algorithm used in the library is as follows:

1. Start-up
   (a) \( t = T - 1 \).
   (b) Set the cost-to-go function \( -\log \gamma(x_T) = 0 \).
   (c) Set the controller matrix to \( \frac{1}{m} \).
   (d)

\[ \lambda = \log C_t - D(s_{t+1}||S_{t+1}) \]  

This matrix is computed only once and buffered in auxiliary memory for later use.

2.

\[ \log \hat{c}_t = \lambda - E_{s_{t+1}}[-\log \gamma_{t+1}(x_{t+1})] \]
3. (a) Calculate the modified standard gradient $\nabla F'$:

$$
\nabla F' = \begin{bmatrix}
0 & \cdots & 0 \\
\log \frac{c(1|0)/c(0|0)}{\tilde{c}(1|0)/\tilde{c}(0|0)} & \cdots & \log \frac{c(1|n)/c(0|n)}{\tilde{c}(1|n)/\tilde{c}(0|n)} \\
\vdots & \ddots & \vdots \\
\log \frac{c(m|0)/c(0|0)}{\tilde{c}(m|0)/\tilde{c}(0|0)} & \cdots & \log \frac{c(m|n)/c(0|n)}{\tilde{c}(m|n)/\tilde{c}(0|n)}
\end{bmatrix}
$$

(13)

Note that depending on the problem at hand, the standard gradient may be different and will have to be supplied by the user.

(b) Compute the modified natural gradient:

$$
\tilde{\nabla} F' = c(u_t|x_t) \circ \nabla F' - c(u_t|x_t) \cdot \text{diag}([1 \ldots 1](c(u_t|x_t) \circ \nabla F'))
$$

(14)

(c) Update the controller matrix using $\tilde{\nabla} F'$ and the step, $\eta$:

$$
c(u_t|x_t) = c(u_t|x_t) - \eta \tilde{\nabla} F'
$$

(15)

(d) Calculate the (Riemannian) norm of the standard gradient:

$$
\alpha = \max(\nabla F'^T \tilde{\nabla} F')_{ii}, i = 0, 1, \ldots, n
$$

(16)

The scalar value $\alpha$ is compared against a user-defined threshold $\epsilon$; if $\alpha$ is lower than $\epsilon$ then the loop breaks and execution continues at step 4. Otherwise, execution resumes at step 3.a.

4. Calculate the cost-to-go function:

$$
-\log \gamma_t(x_t) = E_{c_t} \left[ \log \frac{c_t}{\tilde{c}_t} \right]
$$

(17)

5. $t$ is decreased by one and execution resumes at step 2. This procedure is repeated until $t = 0$
3 Implementation

This will describe the software implementation of the algorithms described in section 2, such as development process, design decisions, data structures, test cases and other relevant topics.

3.1 Objectives

The guiding objectives for this project were set at its inception and were roughly defined as follows:

• Create a reference implementation of the fast Kullback-Leibler divergence optimization algorithm described in [2, 3, 4] using the standard C programming language.

• Use available tools to create a shared-memory parallel implementation of the library.

• Port the code to the NVIDIA CUDA general purpose GPU (GPGPU) architecture.

Initially the Basic Linear Algebra Subroutines (or BLAS) were intended to be used to perform the optimization algorithm’s matrix and vector operations, and the availability of highly optimized versions of this library distributed by CPU vendors (Intel’s Math Kernel Library and AMD’s Core Math Library) made the use of these routines seem attractive. However, during the process of writing the code necessary to implement the library, the BLAS turned out to be ill-suited as a base for this implementation.

For example, the reference BLAS lack element-wise multiplications between matrices and between vectors, an operation used very frequently in the library. This was not a major drawback from a programming perspective because Hadamard products are trivial to implement. However, even if this operation were supported in the BLAS, there are still cases where it would be more efficient not to use it. For instance, in equation (7) a vector-vector element-wise product is performed followed by a dot product. From a practical standpoint, it is more efficient to carry out both these operations simultaneously instead of sequentially.

Although the BLAS are not used, the library was developed to be compatible with them in terms of data structures, enumerations and definitions (provided by the cblas.h header file). The library’s calling interface is also similar to the BLAS’ FORTRAN-style interface.

3.2 Language and nomenclature

As mentioned in the previous subsection, the library is written in C (except for the CUDA codebase, which is written in a C dialect created by NVIDIA). More specifically, it doesn’t deviate from the most recent C standard (ANSI C99) and strives to be operating system and compiler agnostic. This is accomplished by avoiding the use of compiler extensions or any other features native to a specific operating system, even if that means sacrificing possible (unportable) performance optimizations.
Another aspect regarding the language was the choice of variable and function naming style. In various algebraic notations, it is common to use characters from the Latin and Greek alphabet to represent matrices (upper-case), vectors (lower-case), and scalar values (lower-case characters from the Greek alphabet). This creates a conflict with some of the conventions used in the C programming language\[9\], where upper-case letters and words are used in macro definitions and in certain typedefs (eg: file pointers), and lower case letters and words are used to name variables and functions. So throughout the library, the conventional C naming style was given precedence over mathematical notation, with Greek letters being expanded to their Latin pronunciation (eg: $\alpha$ is replaced with “alpha”), and matrices and vectors are given lower-case names (or acronyms, eg: cost-to-go is replaced with “ctg”).

To prevent possible conflicts with functions, global variables, definitions and enumerations that may occur when a user links his or her code against the optimization library, the keywords klopt, KLOPT, klopt,cuda and KLOPT,CUDA are reserved for use by the library as name prefixes or suffixes. This is similar to the way the system C library reserves the _t suffix to prevent typedef conflicts.

### 3.3 Data structures

There are three principal data structures in the optimization library: vectors, matrices and matrix arrays.

![Figure 2: Vector (upper left); matrix (lower left); matrix array (right)](image)

Each one of these structures follows the layout used by the reference BLAS.

#### 3.3.1 Vectors

Vectors consist of a linear array of memory, where elements are stored according to a user-defined increment value or stride. Unit stride vectors have their element stored at adjacent memory locations, which is the optimal layout for CPU architectures that posses fast on-chip caches. Non-unit stride vectors should therefore be avoided since a stride equal or larger than the number of elements that a cache line can hold will mean that each element access will generate a cache miss and cause serious performance degradation if the vector is large and/or used repeatedly.
If the stride is negative, then the vector is processed in reverse order. Some of the routines in the BLAS library accept a zero stride value\(^3\) (where the first element of the vector, or a scalar value, used repeatedly as though all the vector’s elements were the same) while others don’t\(^4\). The KL optimization library does not reject zero stride values for vectors, but it is up to the user to make sure that a zero-stride vector is not written to (the first element will be “clobbered” repeatedly and the desired results won’t be obtained).

### 3.3.2 Matrices

To maintain compatibility with the BLAS, matrices are not declared as two-dimensional C arrays but as plain linear arrays. To help index the matrix, the number of rows, columns and a *leading dimension* have to be provided by the user. The layout of matrix elements in memory can be done according to rows, where elements from the same row are in adjacent memory positions, or according to columns, where elements from the same column are in adjacent memory positions. The former is called *row-major order* and the latter *column-major order*.

The native memory layouts used for matrices in C is row-major order, while FORTRAN’s native order is column-major. The C BLAS interface makes provisions for the use of both orders, providing an enumeration to identify them in the `cblas.h` header file.

---

\(^3\)http://www.netlib.org/blas/ddot.f

\(^4\)http://www.netlib.org/blas/dgemv.f
Figure 4: Right: row-major matrix; left: column-major matrix; bottom: matrix memory layout.

As for the leading dimension, it serves as a stride value when indexing a matrix and depending on the memory layout, row-major or column-major, this value is equal to or larger than the number of columns or the number of rows, respectively. This allows the creation of sub-matrices from pre-existing matrices by varying the number of rows and columns while keeping the leading dimension the same.

3.3.3 Matrix arrays

As the name suggest, a matrix array is a composite structure and can be visualized as a “cube” composed of several layers of matrices. This structure is used solely to represent Markov chains in probabilistic control problems, where each “edge” indexes the input ($u_t$), current state ($x_t$) and future state ($x_{t+1}$).
In this case the leading dimension is used as an increment value to obtain the next index of $x_t$ (if the order is row-major) or $u_t$ (if the order is column-major). It is important to mention that the values for the number of rows is associated with the system model input $u_t$ and the number of columns is associated with $x_t$ and $x_{t+1}$.

3.3.4 Structure representation

In the early stages of the library development, vectors, matrices and matrix arrays were represented by C structures (struct). All relevant information (memory location, dimension, etc) was contained in the structure fields. The library also contained a set of functions dedicated to memory management, simplifying the tasks of allocating and de-allocating memory resources for the vector, matrix and matrix array structures.

```
struct vector {
    double *buff;
    unsigned int dim;
    inc;
};
```

Listing 1: A vector represented by a C structure

However, the use of C structures was abandoned when development of the CUDA codebase began, since there was a potential problem regarding the way the compiler for the user code (GCC) and the compiler for the CUDA code (NVCC) would decide to pad or align the structures’ fields. If, for example, the CUDA library used 16 byte alignment for the structure fields, there would be no guarantee that that alignment would be compatible with GCC’s attribute((aligned (16))). Moreover, this alignment feature in GCC is an extension to the C language and, as per the development objectives, should be rejected. Another problem was the way the compilers might pad the structures with extra bytes, causing the sizeof value of a structure to be different between C and CUDA code. As a work-around, the library could copy each structure’s parameter one-by-one, however performing such small memory transfer operations between main memory and the GPU memory is not recommended.[5]

Given the above problems, the library’s interface was changed to resemble that of the BLAS, where the individual structure fields are passed directly to the optimization functions in the form of a long parameter list. The disadvantage is that this FORTRAN-like interface is not as elegant as the use of C structures to house information, however this is more of an aesthetic and stylistic problem than a practical one.

3.4 Layout

The KL optimization library is composed of an optimization interface that is meant to be called by the user (for example, to find an optimal controller for a system model) and a collection of “helper” functions that carry out the numerical calculations described in section 2 as well as others that perform mundane tasks such as allocate memory or print data to a file.

The optimization interface is exactly the same for both the C and CUDA codebases, thereby providing the user with a means to link his or her code to
the library without any drastic source code changes (any such changes, like the floating point type, can be handled automatically by the preprocessor).

In practice, a program written by the user will have to include the klopt.h header file and define a preprocessor macro to select the desired floating point precision when compiled, followed by linking the compiled object with the desired shared optimization library (libklopt.so or libklopt_cuda.so to use the C or CUDA libraries, respectively). Further information about the library macros and internal functions is provided in section 6.

3.5 OpenMP

OpenMP is a standardized application programming interface designed to facilitate shared-memory parallel programming. The standard, maintained by a non-profit corporation called the OpenMP Architecture Review Board and currently at its third revision, has been implemented by major compiler vendors (namely Intel, The Portland Group, Microsoft and many others) and prominent free software projects (such as the GNU Compiler Collection). Therefore, OpenMP is ideal for creating portable concurrent code.

Although OpenMP provides functions that a user can call to manage threads and create parallel code, the standard also defines a collection of preprocessor directives that an OpenMP-aware compiler can use to automatically parallelize a section of sequential code. In [1], OpenMP is described as having a fork-join model, where a single thread (known as the master thread) executes code until a parallel region (defined by the compiler directives) is encountered. Once at the parallel region, a team of threads is spawned (or forked) and the work in this region is distributed among them. As soon as the end of the parallel region is encountered (by all threads in the team, unless otherwise stated), the non-master threads are joined and the program execution is resumed in a sequential manner by the master thread.

Consider the following code segment, where a loop carries out the element-wise product of two $n$-length floating point arrays, $a$ and $b$, and stores the result in array $c$:

```c
for (i = 0; i < n; i++)
    c[i] = a[i] * b[i];
```

Listing 2: Vector Hadamard product in C.

To parallelize the above loop, one need only insert the appropriate OpenMP directive.

```c
#pragma omp parallel for default(none) \
    shared(a, b, c, n) private(i) schedule(runtime)
for (i = 0; i < n; i++)
    c[i] = a[i] * b[i];
```

Listing 3: Parallel vector Hadamard product.

This `pragma` directive informs the compiler what type of parallel region to create (in this specific case, a parallel for loop), which variables are shared and accessible to all threads, and which variables need to be thread-private (such as the index variable, $i$). Note that variables declared as constant are
implicitly shared and do not need to be declared in the `shared` clause. The `schedule(runtime)` clause informs the compiler that the resulting parallel code should be aware of the `OMP_SCHEDULE` environment variable, which the user can manipulate to change the type of thread scheduling used. If this environment variable isn’t set, then the default `static` scheduling scheme is used. The static scheduler consists of simply dividing the number of loop iterations ($n$) evenly between the threads in the team. For example, in a loop with $n = 10$ and two threads, the first thread will start at index $i = 0$ and end at $i = 4$, while the second thread starts at $i = 5$ and ends at $i = 9$.

### 3.5.1 Performance considerations

Although OpenMP provides a very straightforward means to create SMP-aware code, this convenience can have its own performance cost and limit control over thread behavior. There are also problematic aspects of shared-memory parallel programming entirely divorced from OpenMP that need to be taken into account parallelizing a section of code. Some of these problems are presented in this subsection:

**Threading overhead**  
Using multiple threads to distribute work amongst several processors may greatly increase a program’s performance, ideally by a factor of the number of processors available and the amount of parallel code, however the use of multiple threads in a program unavoidably adds a computational overhead associated with managing and scheduling threads during execution. A spatial overhead is also incurred since the amount of stack space required by the program for storing private variables and other information grows linearly with the number of threads used. For example, if a relatively small amount of work has to be carried out then the threading overhead may completely overshadow the amount of work that each thread executes.

**False sharing**  
False sharing is a problem that occurs in shared-memory parallel programs where the memory locations being accessed by each thread are very close together. Consider a hardware architecture that features a hierarchical memory layout, with very small (but fast) caches dedicated to each processor (level one, or $L1$ cache), and a larger (and slightly slower) unified cache shared between all processors (level two, or $L2$ cache).

If, for instance, all threads were to read memory positions within the range of a single cache line, then all processors would have to load the same line of data into their $L1$ caches. If one of the processors were to perform a write operation in this cache line, it would immediately invalidate the corresponding line of data stored in all the other processors’ caches. In architectures that implement cache-coherence schemes, this would force all processors to stall and synchronize the affected cache-line, causing performance degradation. This problem is compounded when multiple threads perform multiple writes to the same memory location and can be a serious limitation to the scalability of the multithreaded code[1].

**Nested loops**  
When the need to parallelize nested loops arises, the best possible option is to parallelize the outermost loop[1]. If the compiler is in-
structed to create a parallel region around the innermost loop, then the threading overhead will be incurred by a factor of the number of outer loops and their respective iteration count. In the following example

```c
for (i = 0; i < m; i++) {
    #pragma omp parallel for private(j) \
    shared(i, m, n, a, b, c)
    for (j = 0; j < n; j++)
        c[i + j] = a[i + j] * b[i + j];
}
```

Listing 4: Parallel region nested within a loop.

the costs of forking and joining the threads, allocating stack space for their individual private variables and data structures is incurred \(m\) times. It is much more sensible to simply move the start of the parallel region to the outermost loop, where the the creation of the thread team takes place only once.

**Load balancing**  Another problem associated with the use of multiple threads in an algorithm is the distribution of the workload. Suppose that a matrix multiplication function, \(C = AB\) is written where the outermost loop is parallelized. \(A, B\) and \(C\) have the following dimensions, respectively: \(m \times n\), \(n \times p\) and \(m \times p\). Considering a row-major order and that the number of rows of matrix \(A\) is smaller than the number of available processors, then there will always be \(N_{procs} - m\) idle processors. The problem worsens as the number of columns of \(A\) increases since each active processor’s load grows while the remaining processors stay idle.

A possible solution would be the use of a scheduler that distributes the work over the rows and columns of the matrices, however it would also have to manage communication between threads within the same row of \(A\) and column of \(B\) (for example, to share partial summation results) which would once again introduce more overhead into the innermost loop. Unfortunately, such a scheduling scheme does not exist in the OpenMP standard, so this case can be seen as a limitation or performance-simplicity trade-off in OpenMP.

### 3.5.2 Usage in the optimization library

Once the reference KL optimization library was written, it was fairly simple to create a shared-memory parallel implementation using OpenMP’s directives, with almost no modifications to the C base code. There were, however, a few instances where using OpenMP naively lead to serious performance degradation, which was the case when implementing the row-major functions for equations (14), (16) and (17).

For illustrative purposes, lets consider the `klopt_ctg_rmajor` function (based upon equation (17)).

```c
for (i = 0; i < rows; i++) {
    ic = i * ldc;
    icc = i * ldcc;
    ic tg = start;
    for (j = 0; j < cols; j++) {
```
Listing 5: Serial version of the row-major cost-to-go-function.

This sequential row-major function worked without any problems and also performed well compared to its column-major counterpart. Using OpenMP directives, this function’s outer loop was parallelized, such that each thread would be assigned a range of rows to process. The problem with this strategy, however, was that all the threads would have to add their computed results to the \( \text{ctg} \) vector without interfering with one another. To accomplish this, the inner loop would have to contain a critical section to prevent threads from overwriting each other’s results.

```c
#pragma omp parallel for default(none) schedule(runtime)
private (i, j, ic, icc, ictg) shared (ctg, ctrl, cctrl, start)
for (i = 0; i < rows; i++) {
    ic = i * ldc;
    icc = i * ldcc;
    ictg = start;

    for (j = 0; j < cols; j++) {
        #pragma omp critical
        {
            ctg[ictg] += ctrl[ic] * (LOG(ctrl[i]) - cctrl[icc]);
        }
        ic++;
        icc++;
        ictg += inc;
    }
}
```

Listing 6: Row-parallel version of the row-major cost-to-go function.

\( \text{ldc} \), \( \text{ldcc} \), \( \text{inc} \), \( \text{rows} \) and \( \text{cols} \) are constant variables. Although the above code functions correctly and produces accurate results, it is slower than the sequential code for the following reasons:

1. Thread synchronization in the critical section. When the threads attempt to write to the same position in \( \text{ctg} \), all but one will be allowed continue while the others wait in a blocked state.

2. False sharing. All processors will be attempting to perform write operations on the same cache line, forcing them to synchronize the affected cache line every time.

Obviously a new strategy is necessary. At a first glance, one may be inclined to parallelize the inner loop to prevent the above mentioned problems, however as previously explained, nesting a parallel section within a loop has performance implications of its own.

If each thread can be allowed to write to its own specific portion of the cost-to-go vector while at the same time being given multiple matrix rows to process,
then all the problems considered in the naive parallel version of klopt\_ctg\_rmajor can be almost entirely eliminated.

This strategy was implemented using a technique known as loop blocking or loop tiling, where a matrix’s elements are processed in a series of blocks instead of row-by-row or column-by-column. Generally, loop blocking is used to diminish the number of cache misses when accessing data using large strides in operations such as matrix transpositions. In this case, loop blocking is useful because threads can be assigned blocks of data that don’t interfere with each other, with a fairly low overhead associated with creating the blocks.

```c
#pragma omp parallel default(none) \
private(i, ii, j, jj, ic, icc, ictg) \
shared(ctg, ctrl, cctrl, start)
{
    for (ii = 0; ii < rows; ii += BLOCK_SIZE) {
        #pragma omp for schedule(runtime)
        for (jj = 0; jj < cols; jj += BLOCK_SIZE) {
            for (i = ii; i < rows && i < ii + BLOCK_SIZE; i++)
            {
                ic = i * ldc + jj;
                icc = i * ldcc + jj;
                ictg = jj * inc + start;
                for (j = jj; j < cols && j < jj + BLOCK_SIZE; j++)
                {
                    ctg[ictg] += ctrl[ic] * (LOG(ctrl[ic]) - cctrl[icc]);
                    ic++;
                    icc++;
                    ictg += inc;
                }
            }
        }
    }
}
```

Listing 7: Block-parallel version of the row-major cost-to-go function

The BLOCK\_SIZE macro has a default value of 32, which is the equivalent of 4 cache lines containing double precision floating point values on most modern Intel CPUs. This value may of course be changed, but using a very large block size will reduce the number of blocks in the matrix which may in turn limit the scalability of the code when there are many processors available.

## 3.6 CUDA

In this subsection an overview of NVIDIA’s Compute Unified Device Architecture (CUDA) will be given, followed by details regarding the KL optimization library’s implementation on this platform.

### 3.6.1 The architecture

The general purpose graphics processing unit (or GPGPU) is an emerging computational platform that allows developers to take advantage of the highly parallel nature of modern graphics chips. Unlike conventional central processing units which tend to focus on using fairly high clock speeds to achieve fast code
execution and have a small number of cores, the GPGPU platform leverages a massively parallel shared-memory hardware architecture, in some cases using several hundred cores concurrently to achieve high instruction and data throughput[5]. Algorithms that are well suited to this architecture will often enjoy very big speed-ups when compared to their CPU-based counterparts.

At the moment, there are two major hardware vendors shaping the GPGPU market: NVIDIA, with its CUDA platform, and ATI’s Stream architecture. Although the manner in which these two architectures expose the hardware parallelism to the application developer, is similar, they are not compatible and are (for the most part) proprietary technologies.

### 3.6.2 Programming CUDA-enabled devices

In section 3.5 the use of OpenMP directives to parallelize the library’s C code was discussed, as well as problems associated with them. Some of these issues in OpenMP stem from the fact that it is merely an extension of an existing serial programming model and therefore inherits language constructs that may not be well suited to express a particular algorithm in a parallel manner. The row-major cost-to-go function in section 3.5 is one such case.

The CUDA programming model, on the other hand, was created with instruction-level parallelism in mind and can in many cases allow a developer to more intuitively expose and an algorithm’s parallel nature. To this end, NVIDIA released a software development kit which includes, amongst other things, a compiler for a language based on C (called C for CUDA) which allows programmers to harness CUDA-enabled GPUs. Programs that make use of CUDA-enabled hardware are composed of two main types of code: host code which runs on the computer’s CPU and device code which runs in the graphics card.

C for CUDA defines a new type of function called a kernel which, once called by the host code, is executed $n$ times in parallel, instead of sequentially like traditional C functions. Kernels are declared with the _global_ keyword and always have a void return value. When calling a kernel from host code, an extra set of parameters enclosed in chevrons ($<<<\cdots>>>$) contain information about the number of threads to be used, directly followed by the regular C function parameters. It is also the responsibility of the host code to allocate any necessary memory on the device using the cudaMalloc function and/or copy data from the computer’s main memory to device memory with cudaMemcpy.

For example, the following code snippet is a CUDA kernel which performs the same task as the serial C code from listing 2:

```c
__global__ void vector_hadamard(int n, float *a, float *b, float *c)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n)
        c[idx] = a[idx] * b[idx];
}
```

Listing 8: Vector Hadamard product kernel.

The kernel would then need to be called by the host code in the following manner:

```c
vector_hadamard<<<gridSize, blockSize>>>(n, a, b, c);
```
where blockSize specifies the number of threads per block and gridSize specifies the number of thread blocks to be launched, and a, b and c are pointers to single precision floating point arrays located in the device’s memory.

In this specific example, the block size and grid size are one dimensional values, however it is possible to specify two (or three, in the case of the block size) dimensional values for these parameters using the dim3 type. Two dimensional block and grid sizes would be necessary if, for example, the data structure being processed by the kernel were a matrix.

Current CUDA hardware allows each thread block to contain a maximum of 512 threads, and permits a grid size of up to 65535 blocks (inclusive)[5].

3.6.3 Applying the CUDA model to the optimization library

The suggested implementation strategy in [5] is the analysis of the target algorithm with the objective of exposing any parallelism it may have and identifying sections where parallel execution may be broken. If indeed there are sections of the algorithm that will cause parallel execution to break, then one possible solution is to decompose it into a series of smaller (and easier) to solve problems.

If a problem requires inter-thread communication (such as summation), then threads within the same block can communicate with each other via a fast on-chip memory called shared memory (similar to a CPU’s cache and whose contents are visible only to threads in the same block). If threads from different blocks need to communicate with one another, then they will be forced to do so via the device’s uncached global memory.

Consider, once again, the implementation of the cost-to-go function based on equation (17) as an example of the strategy used in creating the library’s CUDA codebase. After \( c(u_t|x_t) \circ \log \frac{c(u_t|x_t)}{\tilde{c}(u_t|x_t)} \) has been computed, a summation must be performed over the resulting column elements, and subsequently stored in the ctg vector. This summation is carried out in the following manner: a loop is created where the first half of the thread block fetches the values owned by the second half, via shared memory, and adds them to its own value; at the following iteration the loop size is halved and this time only a quarter of the threads in the block will perform the fetch-and-add operation. This procedure is carried out until all the elements in the block’s first row contain the final summation result, at which point these values can be written out to their destination.

```
for (i = CUDA_BLOCK_SIZE >> 1; i > 0; i >>= 1) {
    if (tidx < i)
        aux[tidy][tidx] += aux[tidy][tidx + i];
    __syncthreads();
}
```

Listing 10: Summation over the column elements of of a matrix. tidx and tidy are respectively the row and column coordinates of the current thread. aux is an auxiliary array located in shared memory.

However, when large optimization problems are being dealt with, a single thread block may be insufficient to span the dimension of the \( c \) and \( \tilde{c} \) matrices
and therefore multiple thread blocks will be required. The problem can be solved thusly: in the host section of the code, a loop must be created such that at each iteration a new grid is spawned and is set to continue the calculations where the previous grid left off. This way, the kernel is called as many times as necessary to complete the computation of the cost-to-go vector.

### 3.6.4 Summation on the CUDA architecture

One of the sources of performance bottlenecks in CUDA device code is low instruction throughput which originates from the presence of conditional branches (such as if, switch and for statements) which force threads from the same warp (or group of 32 threads from the same block) to execute their instructions sequentially\[^5\] instead of simultaneously. This problem will inevitably occur in the above mentioned summation algorithm because there will be a time at which the loop delimiter drops below 32 and there are guaranteed to be divergent threads.

Other more complex and efficient summation algorithms are presented in the CUDA SDK documentation, the most efficient of which is actually implemented in a library called CUDPP and distributed with the SDK. However, not only is the library code itself very large (static library file libcudpp.a is about 5.3MB) but the algorithm appears to be best suited at performing fast summations on very large arrays (\(2^{22}\) elements) whereas the largest problem used to test the library has a 256 \(\times\) 256 controller matrix meaning that the largest linear array summations are of 256 elements.

### 3.7 Numerical issues

One of the challenges of many scientific and numerical computing is to perform calculations with real numbers and obtain meaningful and accurate results that can be used to solve problems in the physical world. This section will discuss numerical problems encountered while developing the KL optimization library and the steps taken to identify and compensate for them.

#### 3.7.1 Finite precision arithmetic

In integer arithmetic, where calculations are performed in the discrete set \(\mathbb{N}\), all the values within a range determined by the computer architecture can be represented precisely with a finite number of bits, and mathematical properties such as associativity and distributivity remain valid.

Floating point arithmetic, on the other hand, has several practical problems. Given the continuous nature of a set such as \(\mathbb{R}\), it is physically impossible to represent all the values in a subset of \(\mathbb{R}\) ranging between \(a\) and \(b\) (with \(a \neq b\)), because there are an infinite number of values within this bracket. This means that the storage of a real number will be done by approximation if it can’t be precisely represented, and will carry a rounding error which is dependent on the underlying computer hardware\[^7\]. Most, if not all, modern CPUs implement the IEEE 754 floating point standard, so there are at least well defined behaviors that can be expected from operations such as rounding or denormalization.
3.7.2 Impact on the optimization algorithm

Originally, [2] and [4] were the reference sources for the implementation of the software library. By the time the bulk of the C base code had been implemented, the four main test cases in section 4 had also been written and thoroughly debugged, producing accurate test results using the runtime parameters from [2].

However, to be certain that the library could handle problems of arbitrary size and not only the specific ones in the source material, the program for test case four was modified to receive command-line arguments from the user to specify custom runtime parameters, such as memory order, step size, precision, timeframe and the desired controller matrix size. After running this modified version of the program with a script where many controller matrix sizes were tried, there were several instances where the results written by the program consisted solely of NaNs.

The existence of a coding bug that could affect only certain problem sizes without any specific pattern was improbable, especially since this preliminary testing was done without OpenMP or any other compiler optimizations enabled, so that threading bugs or aggressive code modifications from the compiler couldn’t influence the outcome of the algorithm execution. A problem resulting from the use of floating point arithmetic was therefore more likely to be the source of the malfunction.

But how to detect the source of the problem? Firstly, the various library functions were modified to dump their computed values into a text file, where the results could be analyzed. Using a test case that executed correctly (controller size $64 \times 64$) and one that malfunctioned ($32 \times 32$), the individual, dumped routine values were scrutinized for discrepancies. There was indeed a point at which the expected data from the latter test case was not coherent with the former, in the ctrl function. This function is responsible for updating the controller matrix (see equation (15)), and in its original form the first element of each column was calculated by subtracting the sum of the remaining column element from one ($1.0$). It so happens that for certain problem sizes, there would be a column who’s first element would be computed to be exactly zero, instead of a value close to zero as observed in the test case that executed successfully. These near-to-zero values also happened to be in the order of $10^{-16}$, which in double precision floating point arithmetic is where the unit roundoff occurs[8], meaning that they could be “out of reach” due to the accumulation of rounding errors in the summation operation.

In fact, to test this hypothesis, the Kahan summation algorithm presented in [8], which strives to keep track of and correct error accumulation, was implemented and used by the ctrl routine. With the use of this numerical method, the optimization library finally yielded correct results for all the cases tested. However, there were still problems surrounding its usage: (a) the Kahan summation algorithm, when implemented as a separate function, caused performance deterioration for the column major functions, and would be even more impractical to use in the row-major functions; (b) although the library yielded correct results when configured to use double precision floating point values, it continued to produce the same problem mentioned earlier when compiled in single precision floating point mode. Note that the unit roundoff for single precision IEEE floating point values is in the order of $10^{-8}$, which meant that even the
Kahan algorithm wouldn’t be able to correct the rounding errors accumulated during summation. In [8] it is also suggested that to obtain numerically stable algorithms the subtraction operation with values contaminated with rounding errors should be avoided (even though this may sometimes be unavoidable). Obviously, a different approach to solving the problem was needed. Coincidentally, in [3] Barão describes a means of computing the modified natural gradient ($\nabla F'$) over the entire probability distribution $p$, such that all the elements of the updated controller matrix can be computed independently. With it, the library also yields accurate results when compiled in single precision mode and called by the fourth test case with any problem size. This method provides better numerical properties than the original by avoiding the use of summation altogether when calculating the first row of the controller matrix, and it doesn’t negatively impact performance.
4 Test cases

The test cases presented in this section were taken directly from [2] and implemented in the C programming language.

4.1 Test 1

This is a very simple test case, aimed at demonstrating the efficiency and stability of the natural gradient applied to the iterative minimization of the Kullback-Leibler divergence.

Given two p.m.f.s, \( p(x) = \left[ \frac{1}{3}; \frac{1}{3}; \frac{1}{3} \right] \) and \( q(x) = [0.2494; 0.0025; 0.7481] \) where \( p \) is the initial guess and \( q \) is the target distribution, the standard gradient of the KL divergence between the two is calculated as follows:

\[
\nabla F' = \begin{bmatrix}
0 & 0 & 0 \\
-1 & 1 & 0 \\
\vdots & \ddots & \vdots \\
-1 & 0 & 1
\end{bmatrix} \log \frac{p}{q}
\]

The \texttt{klopt\_optimize} function from the library is called to minimize \( D(p||q) \) using the natural gradient and the steepest descent method, with a step size \( \eta = 0.18 \) and a precision threshold of \( 1 \times 10^{-8} \) (to be used as the stopping criterion). Convergence to the target distribution \( q \) is swift and precise (see figure 5).

![Figure 5: Minimization of \( D(p||q) \) using the natural gradient method, where \( q \) is represented by the green asterisk and \( p \) is represented by the red asterisks.](image)

Consider figure 6 where the standard gradient is used in the steepest descent method, in an attempt to minimize the KL divergence between \( p \) and \( q \). A small
fixed step size of 0.009 is used with a threshold value of $1 \times 10^{-8}$, and while at first the process may appear to be performing well, it quickly becomes unstable and the target is overshot.

**Figure 6:** Minimization attempt of $D(p||q)$ using the standard gradient method. This representation of the process in $\mathbb{R}^2$ clearly shows the erratic behavior of the algorithm as the target distribution is approached.

### 4.2 Test 2

This test case is another artificial problem, similar to the previous one, except that in this case the objective is to minimize the KL divergence between two joint probability distributions $p(x, y)$ and $q(x, y)$, the latter being the optimization target. $p(x, y)$ is generated by the chain rule as follows:

$$p(x, y) = p(y, x)p(x)$$  \hspace{1cm} (19)

where $p(y, x)$ is an $m \times n$ transition matrix provided by the user and $p(x)$ is free. The standard gradient for this KL divergence is given by

$$\nabla F' = \begin{bmatrix} 0 & 0 & 0 \\ -1 & 1 & 0 \\ \vdots & \ddots & \vdots \\ -1 & 0 & 1 \end{bmatrix} \left( P_{Y|X} \circ \log \frac{P_{Y,X}}{Q_{Y,X}} \right)^T \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$$  \hspace{1cm} (20)

$P_{Y,X}$, $P_{Y|X}$ and $Q_{Y,X}$ are matrices representing $p(y, x)$, $p(y|x)$ and $q(y, x)$, respectively and $\circ$ stands for the element-wise product. Note that since the values of $q(y, x)$ never change, $\log Q_{Y,X}$ only needs to be calculated once.
4.3 Test 3

Test case three is probabilistic control problem (albeit a fairly simple one), and as such, it uses the algorithm presented in section 2.3 by calling the klopt_chain_optimize function. The problem consists of a system, \( s(x_{t+1}|u_t, x_t) \), with nine states \( (x_t) \) and three inputs \( (u_t) \):

The probability to transition from some current state \( x_t \) to a future state \( x_{t+1} \), as well the the state-change restrictions are as follows:
The matrix array representing the system model is constructed using this table, and in this specific test case the relevant data for column-major and row-major memory order is stored in ASCII text files in the test3 directory. The data is stored in this manner for legibility, and can easily be altered by editing the test3_cmajor_data.txt and test3_rmajor_data.txt files. The makefile associated with this test case will automatically generate C source code and an array, containing the system model data, by using a scanner that reads one of the above mentioned text files.

The default execution parameters for this test case are a step size of 0.18, a $1 \times 10^{-8}$ precision and a 10 quanta timeframe. The ideal system model is targeted at state three and the ideal system controller is neutral. After the library finishes computing the optimized controller, the test program prints it to the standard output.

$$S(x_t) = \frac{1}{1008} \begin{bmatrix} 1 & 1 & 1000 & 1 & 1 & 1 & 1 & 1 \\ \end{bmatrix}^T$$  \hspace{1cm} (21)

$$C(u_t) = \begin{bmatrix} \frac{2}{3} & \frac{1}{3} & \frac{1}{3} \\ \end{bmatrix}^T$$  \hspace{1cm} (22)

Figure 9: Optimized controller matrix for $s(x_{t+1}|u_t, x_t)$ at $t = 0$. 

<table>
<thead>
<tr>
<th>input ($u$)</th>
<th>maintain state</th>
<th>rotate right</th>
<th>rotate upwards</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.9</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td>2</td>
<td>0.05</td>
<td>0.05</td>
<td>0.9</td>
</tr>
<tr>
<td>3</td>
<td>0.05</td>
<td>0.9</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Table 1: Transition data for the system model.
4.4 Test 4

While the three previous test cases were created mostly to demonstrate the natural gradient’s overall robustness compared to the standard gradient, this final test case is intended to provide a means to execute large scale problems using the optimization library.

This problem is described in [2] as an unstable first order linear system,

\[ x_{t+1} = 1.01x_t + 0.01u_t + w_t, \quad w_t \sim \mathcal{N}(0, 0.1) \]  

(23)

where \( \mathcal{N} \) is the normal distribution with a mean \( \mu = 0 \) and a variance \( \sigma^2 = 0.1 \). \( x \) and \( u \) are obtained by sampling data from an 8-bit A/D converter[4], within ranges \([-2, 2]\) and \([-10, 10]\) respectively. This provides a very large problem to work with, where the controller matrix \( c(u_t|x_t) \) has \( 2^8 \times 2^8 = 256 \times 256 \) parameters and the system model contains \( 256 \times 256 \times 256 = 16777216 \) transition probabilities. The target state and controller vectors are also comprised of 256 elements and are, respectively, given by:

\[ S(x_t) = 0.5\mathcal{N}(-1, 0.04) + 0.5\mathcal{N}(1, 0.04) \]  

(24)

\[ C(u_t) = \mathcal{N}(0, 9) \]  

(25)

To obtain a probabilistic formulation of the above equations, the values they produce are normalized such that \( \sum_{x_t=1}^{256} S(x_t) \) and \( \sum_{x_t=1}^{256} C(u_t) \) are equal to one.

For the purposes of benchmarking the library, the fourth test case’s program doesn’t have parameter such as dimension, timeframe, step size and precision hard-coded into its source code. Instead, these parameters are specified by the user as command line arguments when running the program. For example, to find the optimized controller for the above mentioned system, the program should be run thusly:

```
./test 4 -t 100 -s 0.177 -p 1e-8 -o cmajor -d 256x256
```

Listing 11: Command line execution of the fourth test case

where the -t flag specifies the timeframe, -s the step size, -p the precision, -o the memory order (which can be cmajor or rmajor indicating column-major and row-major memory order, respectively) and -d specifies the controller matrix size with the left-hand value indicating the number of inputs and the right-hand value the number of states. The value used for the step size was chosen somewhat by trial-and-error as the closest value to 0.18 (used in [2]) that functioned for all the problem sizes used in the benchmarks, and for both double and single precision floating point modes.

When the program terminates, it prints the amount of time (wall-clock time) the library spent computing the optimized controller matrix, as well as the total program run time, from start to finish.
Figure 10: Surface plot of the optimized controller matrix for a $64 \times 64 \times 64$ system.
5 Benchmarks and performance analysis

This section presents details regarding the performance of the KL optimization library for both the C and CUDA codebases during their development and the techniques used to identify and, where possible, overcome bottlenecks.

5.1 Hardware

The KL optimization library was developed on a low-end x86_64 desktop computer (see computer A below), however for the purposes of benchmarking the library and determining its parallel scalability, a more powerful computer was provided by the INESC-ID:

Computer A
- CPU: Intel Pentium Dual Core (E2140), 1.6 GHz, 1 MB L2 cache, 32 KB L1 data/instruction cache
- Memory: 1 GB
- GPU: NVIDIA GeForce 8600GT, 512 MB global memory, 4 multiprocessors, 8 cores per multiprocessor
- Operating system: Slackware 13.0, x86_64 build, Linux kernel 2.6.30.6 patched with Perfctr to allow access to hardware counters.

Computer B
- CPU: Intel Core 2 Quad (Q6600), 2.4 GHz, two 4MB L2 caches, shared by processor core pairs, 32 KB L1 data/instruction cache
- Memory: 8 GB
- GPU: NVIDIA GeForce 8400
- Operating system: Gentoo, x86_64 build

The execution time graphs for the C library that will be presented in this section are based on data collected from the computer provided by the INESC-ID (computer B) since it has a more powerful CPU, while the benchmark data for the CUDA library originates from computer B because it possesses a more capable graphics card.

5.2 Program and test parameters

In section 4 four test cases for the library were presented, however only the last one (see section 4.4) is capable of generating data sets large enough to create a significant load on the CPU and perhaps cause (and expose) memory bottlenecks.

The test program is linked against several different builds of the library which illustrate the evolution of its performance as problems were found, and where possible, corrected. The system model dimensions used are powers of two.
ranging from $32 \times 32 \times 32$ to $256 \times 256 \times 256$, with the latter being the largest of the test cases.

For the most part, only one of the memory orders (column-major) is used when performing the preliminary testing since it provides the most cache-friendly data layout and simplest code, when compared to the row-major equivalents of the functions (for example: listing 7). At the end of the section, the execution times of the final revision of the library will be presented for both column-major and row-major codebases.

Finally, the execution times presented are for the KL optimization library only, not including the runtime of the test program itself.

5.3 Benchmarking the C library

5.3.1 Compiler optimizations

The very first performance benchmark carried out was a simple test run of the library without any compiler optimizations, to ensure that the bare code was functioning correctly and the results were not being influenced by any of GCC’s compile-time optimization modifications. As soon as the accuracy of the computed results was confirmed, then a more aggressive optimization mode was tried. GCC’s level two optimization mode performs several modifications to the source code, from arranging instructions to so as to avoid pipeline stalls, to memory alignment optimizations and many others.

![Figure 11](image)

Figure 11: Comparison of the sequential library code compiled with GCC level zero and level two optimization.

Figure 11 shows the serial (non-OpenMP) execution of test case four against the library compiled with GCC level zero and level two optimization, on computer B. Obviously the optimizations offered by the compiler alone already pro-
vide a significant performance increase (roughly a $2\times$ speedup) and should be used whenever possible.

### 5.3.2 Fast logarithm

After using compiler optimizations to speedup the code execution, the next step taken was to find possible problems with the library itself by identifying areas that consumed the most CPU time.

There are tools specifically designed for this task called *profilers*, which examine the code execution and present a report when the program terminates. The means by which the profilers accomplish this may vary. For example, the GNU GProf profiler inserts code into the target program at compile-time which performs the timing operations and dumps the results for the user to examine. This particular means of code profiling may not be desirable since inserting code into the program may, even if unintentionally, alter its behavior in an unwanted manner.

Another performance analysis tool called Perfsuite, developed by the National Center for Supercomputing Applications at the University of Illinois, is capable not only of creating a profile of a program based on the amount of time each function consumes without modifying the it’s source code, but it is also able to access the CPU’s internal performance counters to determine other details such as pipeline stalls, cache miss/hit ratios, and other hardware related information.

Having used Perfsuite in timer mode on the development computer (A), a profile was generated showing the relative time percentile that each function occupied during execution.

<table>
<thead>
<tr>
<th>Samples</th>
<th>Self %</th>
<th>Total %</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>868</td>
<td>46.74%</td>
<td>46.74%</td>
<td>_ieee754_log</td>
</tr>
<tr>
<td>467</td>
<td>25.15%</td>
<td>71.89%</td>
<td>klopt_cc_cmajor</td>
</tr>
<tr>
<td>110</td>
<td>5.92%</td>
<td>77.81%</td>
<td>klopt_ng_cmajor</td>
</tr>
<tr>
<td>92</td>
<td>4.95%</td>
<td>82.77%</td>
<td>_ieee754_exp</td>
</tr>
<tr>
<td>69</td>
<td>3.72%</td>
<td>86.48%</td>
<td>log</td>
</tr>
<tr>
<td>54</td>
<td>2.91%</td>
<td>89.39%</td>
<td>calc_sys_cmajor</td>
</tr>
<tr>
<td>34</td>
<td>1.83%</td>
<td>91.22%</td>
<td>klopt_maxnorm_cmajor</td>
</tr>
<tr>
<td>33</td>
<td>1.76%</td>
<td>93.00%</td>
<td>_GI___isnan</td>
</tr>
<tr>
<td>31</td>
<td>1.67%</td>
<td>94.67%</td>
<td>klopt_icmd_cmajor</td>
</tr>
<tr>
<td>27</td>
<td>1.45%</td>
<td>96.12%</td>
<td>klopt_ctrl_cmajor</td>
</tr>
<tr>
<td>24</td>
<td>1.29%</td>
<td>97.42%</td>
<td>klopt_sg_cmajor</td>
</tr>
<tr>
<td>21</td>
<td>1.13%</td>
<td>98.55%</td>
<td>??</td>
</tr>
<tr>
<td>9</td>
<td>0.48%</td>
<td>99.03%</td>
<td>normal_dist</td>
</tr>
<tr>
<td>7</td>
<td>0.38%</td>
<td>99.41%</td>
<td>exp</td>
</tr>
<tr>
<td>4</td>
<td>0.22%</td>
<td>99.62%</td>
<td>_finite</td>
</tr>
<tr>
<td>2</td>
<td>0.11%</td>
<td>99.73%</td>
<td>_mpn_rshift</td>
</tr>
<tr>
<td>2</td>
<td>0.11%</td>
<td>99.84%</td>
<td>klopt_ctg_cmajor</td>
</tr>
<tr>
<td>2</td>
<td>0.11%</td>
<td>99.95%</td>
<td>_printf_fp</td>
</tr>
<tr>
<td>1</td>
<td>0.05%</td>
<td>100.00%</td>
<td>_mpn_mul_1</td>
</tr>
</tbody>
</table>

**Listing 12:** Profile of the function execution times in the optimization library.

The profile produced by Perfsuite was fairly surprising since it was expected that the function that calculates the $\tilde{c}(u_t|x_t)$ matrix, *klopt_cc_cmajor*, would be featured at the top of the list as the most time consuming routine. Instead it is
the standard C library’s natural logarithm function, \( \log \), which takes up almost half of the execution time!

Fortunately, a recent paper published by two researchers from the International Computer Science Institute at Berkeley describes an algorithm which allows very fast logarithm computation, by sacrificing a certain amount of floating point precision. In [?], Vinyals and Friedland present the algorithm, dubbed ICSI\textit{Log}, and data demonstrating the performance gains they experienced when they applied it to their work, which coincidentally, also involves Gaussian mixture models.

![Diagram illustrating the ICSI\textit{Log} algorithm concept.](image)

The figure above\(^5\) is an outline of the algorithm, which in short, relies on the pre-computation of a range of logarithms stored in a lookup table for later use. The idea is to be able to create a table large enough to provide sufficient numerical precision while being small enough to reside within the CPU cache. Discarding least significant bits from the mantissa allows a smaller lookup table to be created, with the penalty of reduced precision. The more bits kept, the larger the table will be and the slower the algorithm. There is, however, a “middle-ground” for this performance-precision tradeoff presented in [?] whereby using a 14 bit quantized mantissa, the performance gains remain high while the loss of precision isn’t sufficient to negatively affect the overall results.

The authors of the paper also provide a link to a C implementation of the algorithm, which for convenience sake, has been bundled with the KL optimization library source code.

<table>
<thead>
<tr>
<th>Function Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Samples</strong></td>
</tr>
<tr>
<td>483</td>
</tr>
<tr>
<td>110</td>
</tr>
<tr>
<td>101</td>
</tr>
<tr>
<td>100</td>
</tr>
</tbody>
</table>

\(^5\)Figure used with express permission from Oriol Vinyals.
Listing 13: Profile of the library using the ICSI logarithm with 14 bit mantissa.

As listing 13 shows, logarithm computation no longer occupies the bulk of the program execution time, and as expected, the klopt_cc_cmajor function is the most time-consuming routine.

Figure 13: Serial execution of the library using the ICSILog algorithm (and GCC level two optimization) compared to the same code using the GNU C Library’s logarithm function.

Executing the test program on computer B, the observed execution time for the two largest problem sizes has halved, while the execution time for the two smallest problems has roughly reduced three-fold. This is of course compared to the code that had only been optimized by GCC (-O2).
<table>
<thead>
<tr>
<th>System size</th>
<th>GCC -O0  (s)</th>
<th>GCC -O2  (s)</th>
<th>GCC -O2 w/ ICSILog (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 × 32 × 32</td>
<td>0.1781</td>
<td>0.1163</td>
<td>0.0363</td>
</tr>
<tr>
<td>64 × 64 × 64</td>
<td>0.8321</td>
<td>0.4974</td>
<td>0.1743</td>
</tr>
<tr>
<td>128 × 128 × 128</td>
<td>4.3249</td>
<td>2.3582</td>
<td>1.0500</td>
</tr>
<tr>
<td>256 × 256 × 256</td>
<td>24.7747</td>
<td>11.6696</td>
<td>6.2413</td>
</tr>
</tbody>
</table>

Table 2: Serial execution wall-clock time for the three library builds presented so far.

5.3.3 Loop unrolling

Since the klopt.cc.cmajor function became the new performance bottleneck after adding the ICSI logarithm to the library, the next task was to try to reduce its execution time. However, seeing as the algorithm already presented good memory access patterns for all the buffers used in the function, this would not be an easy (or even possible) task.

Using Perfsuite to gather details about the library’s memory behavior, it was clear that the new hindrance to obtaining further speedups was low instruction throughput:

<table>
<thead>
<tr>
<th>Level 2 total cache accesses</th>
<th>837982502</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 2 cache misses</td>
<td>312488538</td>
</tr>
<tr>
<td>% cycles stalled on any resource</td>
<td>42.831</td>
</tr>
<tr>
<td>Level 1 instruction cache misses per issued instruction</td>
<td>0.000</td>
</tr>
<tr>
<td>Level 1 cache miss ratio (data)</td>
<td>0.046</td>
</tr>
<tr>
<td>Level 1 cache miss ratio (instruction)</td>
<td>0.000</td>
</tr>
<tr>
<td>Level 2 cache miss ratio (instruction)</td>
<td>0.413</td>
</tr>
</tbody>
</table>

Listing 14: Snippet from the Perfsuite hardware profile on computer A.

From the above listing, one can clearly see that although the lowest cache level (L1) performs well for both instruction and data accesses, level two cache usage is mediocre with a miss rate of 0.3729 for data accesses and 0.413 for instruction accesses.

To try to reduce the number of L2, a 3-way loop unrolling of the three nested loops in klopt.cc.cmajor was attempted, where in each loop two arithmetic operations would be performed instead of one operation in the innermost loop. This would shorten the number of loops over xt, xt+1 and ut by half while at the same time making greater use of the S(xt) and C(ut) vectors with every operation performed. Unfortunately, this strategy worsened the performance of the function since the added arithmetic operations, some of which accessed data in the system model over large strides, served only to increase the number of conflicting cache lines in the L2 cache and degrade the overall performance as the system size increased (see figure 14).
In section 3.5, more specifically 3.5.2, the differences between implementing parallel row-major and column-major versions of the library were explained as well as the solutions found to some of the potential problems.

In the column-major benchmark, the threading overhead actually reduces the library’s performance for the two smallest problem sizes when more than one thread is used. Similar behavior is observed in the row-major test, however, in this part of the library the issue is exacerbated by the fact that there are several functions which rely on loop-blocking to take advantage of OpenMP. When the number of blocks available is lower than the number of threads spawned by OpenMP then there will be inactive threads consuming resources, but not carrying out any work. The benefits the loop-blocking in row-major functions is only visible in very large-scale systems (above $128 \times 128 \times 128$).

**Figure 14:** Comparison of the library using unrolled loops in klopt\texttt{cc\_cmajor}, against the previous revisions

### 5.3.4 Column-major and row-major performance with OpenMP

In section 3.5, more specifically 3.5.2, the differences between implementing parallel row-major and column-major versions of the library were explained as well as the solutions found to some of the potential problems.

In the column-major benchmark, the threading overhead actually reduces the library’s performance for the two smallest problem sizes when more than one thread is used. Similar behavior is observed in the row-major test, however, in this part of the library the issue is exacerbated by the fact that there are several functions which rely on loop-blocking to take advantage of OpenMP. When the number of blocks available is lower than the number of threads spawned by OpenMP then there will be inactive threads consuming resources, but not carrying out any work. The benefits the loop-blocking in row-major functions is only visible in very large-scale systems (above $128 \times 128 \times 128$).
Figure 15: Benchmark of the column-major library functions, compiled with GCC level two optimization, ICSILog and with OpenMP enabled. The serial non-OpenMP build is also plotted for reference.

Figure 16: Benchmark of the row-major library functions, compiled with GCC level two optimization, ICSILog and with OpenMP enabled. The serial non-OpenMP build is also plotted for reference.
5.4 Benchmarking the CUDA library

The CUDA library codebase, unlike its C counterpart, did not have the same problems with regards to exposing the KL optimization algorithm’s parallelism. CUDA-enabled GPUs, unlike regular CPUs, do not possess a hierarchical memory with several cache levels between the processor and the main memory. This meant that more source code could be shared between the row-major and column-major functions since cache-sensitive memory access patterns were no longer a concern.

However, the highly parallel nature of the C for CUDA language also imposed restrictions and limitations on operations such as inter-thread communication, making operations such as summation which were trivial on the CPU much more costly and potentially a performance bottleneck.

The ICSI logarithm was also adapted to function with CUDA, however as the following graphs show, the standard logarithm functions provided with the CUDA runtime were not impairing the execution of the functions and kernels in any way (in fact ICSIlog was slightly slower compared to the “vanilla” CUDA base e logarithm).

![Figure 17: Speed test of the CUDA codebase, with and without ICSIlog.](image)

Notice how, in figure 17, the plots for the column-major and row-major CUDA functions match each other almost exactly. This is due to the fact that the differences between the column-major and row-major CUDA code are really quite small, when contrasted with their C counterparts.

Unfortunately, the CUDA implementation did not provide a performance boost to the library as was initially speculated that it might. The necessity to perform frequent data transfers between host memory and device memory coupled with the inevitable loss of performance associated with the summation algorithm used, lead to the CUDA implementation of the KL optimization algorithms...
algorithm being inferior to the CPU version in terms of execution time.

It must be noted, however, that the graphics card used for the tests is a fairly low-end model by today’s standards, with regard to the number of available multiprocessors, clock speed and memory model. Perhaps the library would perform better on a more modern GPU, but even then any speedup obtained would still be limited by the software implementation of the KL optimization algorithm’s implementation.
6 API

6.1 Public interface

The KL optimization library was designed to allow a developer, who wishes to use it, to have access to a standard set of functions that perform the numerical computations necessary to minimize the KL divergence between two probability distributions (eg: section 4.1) or to obtain an optimized controller matrix for a given probabilistic system model (eg: section 4.3).

This public interface consists of two header files: klopt.h and klopt_cuda.h. The former contains all the actual function prototypes, type definitions and macros that a developer will need in order to use the library, while the latter provides a few CUDA specific functions that may be useful to the developer when writing code for the GPGPU. For the most part, including only klopt.h will be sufficient when interfacing with the library.

In the following subsections, the components of this public interface are described.

6.2 Macros and enumerations

Since the library was designed to be built and function according to user specifications, the floating point type of the library is not hard-coded into the klopt.h header file and is instead defined according to a compile-time macro definition. By defining either the KLOPT_SINGLE_PRECISION or the KLOPT_DOUBLE_PRECISION macros, the correct floating point type for the C language will be chosen and automatically defined in the header file.

So, for example, to select the double C type when building the library (and subsequent projects based upon it), one needs only to use a compiler flag to enable the correct macro:

```
$ gcc -DKLOPT_DOUBLE_PRECISION -lklopt_project.o -o project
```

It is important that this type be consistent whenever attempting to use the library.

Finally, the KLOPT_CUDA macro exists merely to disable certain subsections of the library which are not available in the CUDA codebase.

The CBLAS_ORDER enumeration from the cblas.h header file is used to specify the memory order of elements in a matrix. This enumeration will affect the value of a matrix’s leading dimension, such that if the order is CblasRowMajor, then the leading dimension must be equal to or greater that the number of columns in the matrix; if the order is CblasColMajor, then the leading dimension must be equal to or greater than the number of matrix rows. Generally, this leading dimension is specified by a variable named ld, or some variation thereof.

6.3 Functions

6.3.1 klopt_vector_init_index

Argument list:

```
(const size_t n, const int inc)
```
Return type: size_t  
OpenMP: ✗  
CUDA: ✗  
Description:

This function uses the length of a vector and its stride to calculate the array position at which the first element of the vector is located. If inc is positive, then the return value will be zero. If inc is negative then the return value will be equal to the last element in the array where the vector is stored.

6.3.2 klopt_vector_set

Argument list:

```
((const size_t n, const KLOPT_FLOAT alpha, KLOPT_FLOAT *v,  
const int inc))
```

Return type: void  
OpenMP: ✓  
CUDA: ✗  
Description:

Sets all the elements in vector v to the floating point constant alpha.

6.3.3 klopt_vector_print

Argument list:

```
((const size_t n, const KLOPT_FLOAT *v, const int inc, FILE *fp))
```

Return type: void  
OpenMP: ✗  
CUDA: ✗  
Description:

Prints the contents of vector v to the file pointed to by fp, with a precision of eight decimal places.

6.3.4 klopt_matrix_set

Argument list:

```
((const enum CBLAS_ORDER order, const size_t rows, const size_t cols,  
const KLOPT_FLOAT alpha, KLOPT_FLOAT *ma, const size_t ld))
```

Return type: void  
OpenMP: ✓  
CUDA: ✓  
Description:

Sets all the elements in ma to the floating point constant alpha.
6.3.5 klopt_matrix_copy

Argument list:

\[
(\text{const enum CBLAS\_ORDER order}, \text{const size\_t rows}, \text{const size\_t cols}, \\
\text{const KLOPT\_FLOAT *}ma, \text{const size\_t lda}, \text{KLOPT\_FLOAT *}mb, \\
\text{const size\_t ldb})
\]

Return type: void
OpenMP: ✓
CUDA: ✓

Description:

Copies matrix \(ma\) to matrix \(mb\).

6.3.6 klopt_matrix_print

Argument list:

\[
(\text{const enum CBLAS\_ORDER order}, \text{const size\_t rows}, \text{const size\_t cols}, \\
\text{const KLOPT\_FLOAT *}ma, \text{const size\_t ld}, \text{FILE *}fp)
\]

Return type: void
OpenMP: ✗
CUDA: ✗

Description:

Prints the elements of matrix \(ma\) to the file pointed to by \(fp\). The data is printed as it is encountered in memory, with a precision of eight decimal places.

6.3.7 klopt_matrix_fn

Argument list:

\[
(\text{const enum CBLAS\_ORDER order}, \text{const size\_t rows}, \text{const size\_t cols}, \\
\text{KLOPT\_FLOAT *}ma, \text{const size\_t lda}, \text{KLOPT\_FLOAT *}(*fn)(\text{KLOPT\_FLOAT}))
\]

Return type: void
OpenMP: ✓
CUDA: ✓

Description:

Applies the function \(fn\) to all of the elements in matrix \(ma\).

6.3.8 klopt_ng

Argument list:

\[
(\text{const size\_t n}, \text{const KLOPT\_FLOAT *}p, \text{const int incp}, \\
\text{const KLOPT\_FLOAT *}sg, \text{const int incs}, \text{KLOPT\_FLOAT *}ng, \\
\text{const int incn})
\]

Return type: void
OpenMP: ✓
CUDA: ✓
Computes the natural gradient of the KL divergence using the current position, p, and the divergence’s standard gradient, sg. The results are stored in the ng vector.

6.3.9 klopt_icmd_rmajor and klopt_icmd_cmajor

Argument list:

\[
(const \text{ size}_t \text{ rows}, \text{ const size}_t \text{ cols}, \text{ const KLOPT FLOAT } \ast \text{sm}, \text{ const size}_t \text{ lds}, \text{ const KLOPT FLOAT } \ast \text{s}, \text{ const int incs}, \text{ const KLOPT FLOAT } \ast \text{c}, \text{ const int incc}, \text{ KLOPT FLOAT } \ast \text{icmd}, \text{ const size}_t \text{ ldi})
\]

Return type: void
OpenMP: ✓
CUDA: ✓

Description:

These functions compute the ideal controller vector minus the KL divergence, and buffer the results in the icmd matrix (see equation (11)). sm refers to the system model, s to the ideal system vector and c to the ideal controller vector. The column-major function expects a global floating point buffer called klopt_aux to be available and to be at least as long as rows.

6.3.10 klopt_cc_rmajor and klopt_cc_cmajor

Argument list:

\[
(const \text{ size}_t \text{ rows}, \text{ const size}_t \text{ cols}, \text{ const KLOPT FLOAT } \ast \text{sm}, \text{ const size}_t \text{ lds}, \text{ const KLOPT FLOAT } \ast \text{ctg}, \text{ const int inc}, \text{ const KLOPT FLOAT } \ast \text{icmd}, \text{ const size}_t \text{ ldi}, \text{ KLOPT FLOAT } \ast \text{cctrl}, \text{ const size}_t \text{ ldc})
\]

Return type: void
OpenMP: ✓
CUDA: ✓

Description:

These functions compute the \( \tilde{c}(u_t|x_t) \) matrix, as described in equation (12). sm represent the system model, ctg the cost-to-go vector, and icmd the ideal controller minus the system’s KL divergence. The results are stored in the cctrl buffer.

6.3.11 klopt_sg_cmajor and klopt_sg_cmajor

Argument list:

\[
(const \text{ size}_t \text{ rows}, \text{ const size}_t \text{ cols}, \text{ const KLOPT FLOAT } \ast \text{cctrl}, \text{ const size}_t \text{ ldc}, \text{ const KLOPT FLOAT } \ast \text{ctrl}, \text{ const size}_t \text{ ldc}, \text{ KLOPT FLOAT } \ast \text{sg}, \text{ const size}_t \text{ lds})
\]

Return type: void
OpenMP: ✓
These functions calculate the standard gradient of the KL divergence for a given probabilistic control system, with the results being written in the $sg$ matrix. $ctrl$ and $cctrl$ represent the $c(u_t|x_t)$ and controller matrices, respectively. Refer to equation (13) for details on the computations performed.

### 6.3.12 klopt_ng_rmajor and klopt_ng_cmajor

**Argument list:**

```c
(const size_t rows, const size_t cols, const KLOPT_FLOAT *sg,
 const size_t lds, const KLOPT_FLOAT *ctrl, const size_t ldc,
 KLOPT_FLOAT *ng, const size_t ldn)
```

**Return type:** void

**CUDA:** ✓

**Description:**

Both these functions compute the natural gradient matrix using the standard gradient, $sg$, and the controller at the current timeframe, $ctrl$. They also expect there to exist a $cols$-length global buffer named $klopt_aux$, for purposes of auxiliary calculations. The results are written to the $ng$ buffer.

### 6.3.13 klopt_ctrl_rmajor and klopt_ctrl_cmajor

**Argument list:**

```c
(const size_t rows, const size_t cols, const KLOPT_FLOAT step,
 const KLOPT_FLOAT *ng, const size_t ldn, KLOPT_FLOAT *ctrl,
 const size_t ldc)
```

**Return type:** void

**CUDA:** ✓

**Description:**

These functions’ only task is to use the natural gradient matrix, $ng$, and a fixed user defined step size, $step$, to update the controller matrix. The results are written to the $ctrl$ matrix.

### 6.3.14 klopt_maxnorm_rmajor and klopt_maxnorm_cmajor

**Argument list:**

```c
(const size_t rows, const size_t cols, const KLOPT_FLOAT *sg,
 const size_t lds, const KLOPT_FLOAT *ng, const size_t ldn,
 KLOPT_FLOAT *sgnorm)
```

**Return type:** void

**CUDA:** ✓
Description:

These routines compute the maximum value of the natural gradient’s norm. This scalar value is then written to `sgnorm`. `sg` and `ng` are the standard and natural gradients, respectively. The row-major function assumes the existence of a `cols`-length global floating point buffer called `klopt_aux`.

### 6.3.15 klopt_ctg_rmajor and klopt_ctg_cmajor

**Argument list:**

```
(const size_t rows, const size_t cols, const KLOPT_FLOAT *cctrl, const size_t ldcc, const KLOPT_FLOAT *ctrl, const size_t ldc, KLOPT_FLOAT *ctg, const int inc)
```

**Return type:** void

**OpenMP:** ✓

**CUDA:** ✓

**Description:**

These are the cost-to-go functions, and they implement equation (17).

### 6.3.16 klopt_optimize

**Argument list:**

```
(const size_t n, const KLOPT_FLOAT step, const KLOPT_FLOAT threshold, KLOPT_FLOAT *p, const int incp, const klopt_calc_sgv calc_sg)
```

**Return type:** void

**OpenMP:** ✗

**CUDA:** ✗

**Description:**

The purpose of this function is to use the natural gradient function from section 6.3.8 to minimize the KL divergence of two probability distributions (`p` and `q`), knowing only the current position (`p`) and having access to the standard gradient update function (`calc_sg`).

The steepest descent method, coupled with the natural gradient, a fixed step size and precision threshold, is used to minimize $D(p||q)$. The updated values of `p` are written to a vector of the same name.

### 6.3.17 klopt_chain_optimize

**Argument list:**

```
(const enum CBLAS_ORDER order, const size_t rows, const size_t cols, const size_t timeframe, const KLOPT_FLOAT step, const KLOPT_FLOAT threshold, const KLOPT_FLOAT *sm, const size_t lds, const KLOPT_FLOAT *s, const int incs, const KLOPT_FLOAT *c, const int incc, KLOPT_FLOAT *ctrl, const size_t ldc, const klopt_calc_sgv calc_sg)
```

42
Return type: void
OpenMP: ✓
CUDA: ✓
Description:

This routine acts as the “glue” that unites all the other functions described in the API section, to achieve the goal of finding an optimized controller matrix, \( ctrl \), for a system model \( sm \) and target system and controller vectors \( s \) and \( c \). The algorithm behind this function has already been succinctly described in section 2.3.

This routine is also responsible for allocating and managing auxiliary memory for the computations (eg: klopt.aux buffer, amongst others). In the case of the CUDA variant of this function, it also performs the data transfers between host memory and the GPU’s global memory.
7 Conclusions and closing remarks

The KL optimization library, as per the objectives set at the project’s inception, was developed to provide a C implementation of the fast and efficient KL optimization algorithms presented by Professors M. Barão and J. Lemos.

After overcoming several obstacles from different fields such as concurrent and parallel programming, numerical computational methods and performance optimization, one can come to the conclusion, in all honesty, that the project’s objectives have been met. This does not, however, mean that the project has no potential to be developed any further. Quite on the contrary, there are still aspects of the software library that can be improved (such as the addition of a “smart” adaptive step size in the steepest descent method). Also, the emergence of new massively parallel, yet somewhat hybrid hardware architectures such as Nvidia Fermi and (in the future) Intel’s Larabee processors will create new ways in which the same algorithms can be implemented and optimized.

Due to its modular and fairly operating system agnostic nature, modifying the library to function on new systems or simply altering the existing algorithms should be straight-forward and without a high risk of “breaking” the codebase.

Finally, there is also the possibility for development to start on projects whose objectives are not necessarily tied into working on the library’s codebase per se, but rather revolve around using the library as a wheel in a larger system of cogs solving large scale problems, perhaps in areas such as traffic routing or the manipulation of gene regulatory networks.
References


