GPU Acceleration of ADMM for Large-Scale Convex Optimization

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zu

AUTOMATIC CONTROL LABORATORY

Master Thesis

GPU Acceleration of ADMM for Large-Scale Convex Optimization

Michel Schubiger
August 2019

Automatic Control Laboratory
Swiss Federal Institute of Technology (ETH) Zürich

Advisors
Dr. Goran Banjac
Samuel Balula
Prof. Dr. John Lygeros

Abstract

Large-scale convex optimization problems arise in various practical applications. Even though there exist many efficient methods for solving these problems, such as the alternating direction method of multipliers (ADMM), they may take minutes or even hours to compute solutions of very large problem instances. In this thesis we explore the possibilities of using a graphics processing unit (GPU) to accelerate ADMM. We use OSQP as a state-of-the-art implementation of ADMM to analyze the potential to parallelize the algorithm. We identify several parts of the implementation that could be accelerated by using a GPU, such as the direct linear system solver, which we replace with an iterative conjugate gradient (CG) method implemented on a GPU. Our implementation written in CUDA C has been tested on many medium- to large-scale problems in applications ranging from engineering to statistics and finance. The GPU-accelerated algorithm outperforms OSQP by up to 2 orders of magnitude for problems that take more than 15 minutes to solve by the standard OSQP implementation.

Acknowledgements

This master thesis was written between February 2019 and August 2019 at the Automatic Control Laboratory at ETH Zurich. The project offered a very interesting insight into the topic of convex optimization and numerical solvers. I am convinced that the valuable experience I was able to gather during my work will be of great help in my future professional career.

First of all, I would like to express my most sincere gratitude to my supervisors Dr. Goran Banjac and Samuel Balula who excellently guided me during my work. I am grateful for their time and support they offered during my time at IfA. Especially, I want to thank Goran for caring sincerely for my work. It was always a great pleasure to work together with him on this project. Furthermore, I also thank Samuel for his tireless efforts to keep the GPU-machines running. Many thanks also to my parents Monika and Philipp Schubiger and my girlfriend Kathrin for their utmost support at any time of the thesis. Last but not least, I want to express my appreciation to Prof. Dr. John Lygeros for giving me the chance to work on this interesting project.

Michel Schubiger
Zurich, August 2019

Contents

Chapter 1

Introduction

1.1 Convex Optimization

An optimization problem is a mathematical problem of the following form

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq 0, \quad i = 1, \ldots, m, \\ & h_j(x) = 0, \quad j = 1, \ldots, p, \end{array} \tag{1.1}$$

where $x \in \mathbb{R}^n$ is called the *optimization variable* and the function f_0 the *objective* or *cost function*. The functions $f_1, \ldots, f_m$ are the *inequality constraint functions* and $h_1, \ldots, h_p$ are the *equality constraint functions*. A point x is called *feasible* if it satisfies all the constraints defined by the constraint functions. Problem (1.1) is called *convex* if the objective function f_0 and all the inequality constraint functions f_i are convex and the equality constraint functions h_j are affine.

Many optimization problems that arise in practice can be formulated or approximated by a convex program (1.1). There are several important subclasses of convex problems that are relevant in practice, including linear programs (LPs), quadratic programs (QPs), second-order cone programs (SOCPs), and semi definite programs (SDPs). Furthermore, convex programs arise as subproblems in non-convex optimization methods, such as sequential quadratic programming (SQP) and mixed integer programming (MIP).

Of particular interest to us is the QP, defined as

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}x^T P x + q^T x \\ \text{subject to} & Fx \leq b \\ & Hx = d, \end{array} \tag{1.2}$$

where $P \in \mathbb{S}_+^n$, $F \in \mathbb{R}^{m \times n}$, and $H \in \mathbb{R}^{p \times n}$. In case $P = 0$ (1.2) reduces to an LP. The applications of QPs range from engineering, control system, signal processing, data analysis to finance. Applications in control system engineering include model predictive control (MPC) [12] and moving horizon estimation (MHE) [1]. In machine learning, many problems can be formulated as QPs including lasso [29], Huber fitting [21] and support vector machines (SVM) [7]. The field of finance uses portfolio optimization [6, 22], to maximize expected return while reducing risk.

There exists an almost complete theory for convex optimization, which for instance can guarantee that a given solution is indeed the optimal solution. Moreover, there are many reliable and efficient algorithms to solve problems of the form (1.1) including (1.2).

1.2 Solution Methods

Active set methods Active set methods for solving QPs select a working set from the set of inequality constraints and pretend that they are binding (*i.e.* hold with equality). A new decent direction is then determined by minimizing the objective constrained to the set of active constraints. Depending on the decent direction, constraints are then added or removed from the working set in each iteration. These methods can easily make use of an approximate solution to reduce the number of working set updates. However, active set methods suffer from a combinatorial explosion in the number of possible working sets that have to be visited in the worst case. This makes them less attractive for large-scale problems. Practical implementations are part of the commercial solvers, such as MOSEK and GUROBI.

Interior-point methods Interior-point methods transform a constrained problem into an unconstrained problem by penalizing constraint violations with the help of a parameterized barrier function φ, *i.e.*

$$\text{minimize} \quad f_0(x) + \kappa^k \varphi(x), \tag{1.3}$$

where κ^k is the barrier parameter that determines the relative weighting between objective and barrier. In each iteration the method solves the unconstrained problem (1.3) using Newton method to obtain $x^\star(\kappa^k)$. The parameter κ is reduced after each iteration and $x^\star(\kappa)$ converges to the optimal solution $x^\star$ as κ approaches zero. Typically, the obtained solution has a very high accuracy within a few tens of iterations. Most solvers, such as GUROBI, MOSEK, and ECOS use a variant called primal-dual interior-point method, such as the Mehrotra's predictor-corrector method [25]. It became the *de facto* standard for practical implementations because of its good performance across many problems. However, interior-point methods do not scale well for really large problems because the Newton update requires the solution of a large linear system with a high accuracy. Thus, their main application lies in small- to medium-scale problems. Furthermore, they are not easily warm started, *i.e.* they cannot take advantage of an x^0 close to $x^\star$ to reduce the number of iterations.

First-order methods First-order methods are a class of optimization methods that use only first-order information of the cost function, *i.e.* gradients or sub-gradients. The restriction to first-order information allows them to scale to very large problem instances. One of the simplest methods is the projected gradient method. In each iteration a step in the steepest descent direction, *i.e.* negative gradient, is taken followed by a projection onto the feasible set. Operator splitting methods are a class of first-order methods including the alternating direction method of multipliers (ADMM) that reformulate (1.1) as a minimization over a sum of two convex functions,

$$\text{minimize} \quad f(x) + g(x).$$

A common splitting is to use g to represent the constraints via the indicator function [5], which is defined as

$$\mathcal{I}_{\mathcal{C}}(x) := \begin{cases} 0 & x \in \mathcal{C} \\ +\infty & \text{otherwise.} \end{cases}$$

ADMM performs alternating minimization steps which allows for a large flexibility to exploit the structure of the two functions f and g separately. ADMM has been shown to provide modest accuracy solutions for a relatively small number of steps. The iterations are typically very cheap and easy to implement. This makes it ideal for large-scale optimization problems, where high accuracy is not needed due to noisy data and arbitrary objective functions [27]. Furthermore, ADMM can be easily warm started and does not require a high accuracy solution of the arising subproblems in order to converge [5].

1.3 GPU Acceleration

Graphics processing units (GPUs) offer an unmatched amount of parallel computation power for their relatively low price. Moreover, they provide far greater memory bandwidths than conventional CPU based systems. This is especially beneficial for applications that process large amounts of data. It is thus no surprise that usage of GPUs has seen a large spike in the field of machine learning in recent years. Applications range from training deep neural networks [16, 18] to autonomous driving [20]. By reducing training times, GPUs have effectively increased the upper limit on the problem size that is still tractable. Furthermore, many machine learning tools, such as PyTorch, TensorFlow, Theano, and CNTK, have native support for GPU acceleration.

Motivated by these success stories, our goal is to explore the possibilities offered by the massive parallelism of GPUs to solve very large QPs with ADMM. Applications that would benefit most from this include portfolio back-testing [6], evaluation of the regularization path in lasso and other regularization problems. The goal is to reduce runtimes from the order of minutes down to seconds.

We are using the open-source implementation OSQP, which is a robust general-purpose QP solver based on ADMM [27] as a starting point. To improve upon OSQP we propose to replace the direct linear system solver with an indirect (iterative) which is better suited for massive parallel platforms, such as GPUs. Furthermore, we explore the benefits of accelerating other steps of OSQP, such as the problem setup and evaluation of stopping criteria.

1.4 Related Work

SCS SCS is an open-source implementation of a convex cone solver that uses ADMM [26]. SCS has demonstrated that GPUs can be used to accelerate the linear system solver used to solve the arising subproblems.

OSQP OSQP is the open-source QP solver that this work is based on. It has been shown to be up to ten times faster than competing state-of-the-art interior-point solvers, such as GUROBI, MOSEK, ECOS, and qpOASES. The implementation provides a single-threaded linear system solver to solve the arising subproblems and can be easily interfaced with the multi-threaded MKL solver Pardiso.

Chapter 2

Graphics Processing Units

GPUs come in many different variations and architectures. In the following sections we will restrict the discussion to the latest Nvidia GPU generation (codename Turing) for simplicity. Most of the concepts that will be discussed also apply to older Nvidia GPUs. For further details see the Nvidia CUDA C Programming Guide [9].

2.1 GPU Architecture

Even though their name contains the word graphics, GPUs can do much more than computing and rendering computer graphics. GPUs have a many-core architecture with thousands of processing cores. The challenge is to leverage the increasing number of cores and develop applications that scale their parallelism. The answer from Nvidia to overcome this challenge is called CUDA, a general-purpose parallel computing platform and programming model.

2.1.1 Hardware Architecture

A GPU consists of an array of several streaming multiprocessors (SMs). SMs are the basic building block of an Nvidia GPU. Each SM contains several integer and floating-point arithmetic units, local caches, shared memory, and several schedulers. The SMs of different hardware generations are not identical, but are backward compatible. The SM of the Turing generation has the following components [9, Appendix H.6]

- 64 FP32 units for single precision arithmetic
- 2 FP64 units for double precision arithmetic
- 64 INT32 units for 32-bit integer arithmetic
- 8 mixed precision Tensor cores for Deep Learning matrix arithmetic
- 8 special functions units for single precision floating-point transcendental functions
- 4 warp schedulers
- 96 KB of shared memory
- a read-only cache

The on-board RAM of the GPU is called *global memory* (in contrast to the shared memory that is local to each SM). The global memory space is much larger than the local shared memory. It is typically in the order of several GB. Some professional cards have up to 24 GB of global

memory. Compared to the shared memory it has a much larger latency and lower bandwidth. Though, compared to system RAM it is still much faster. Bandwidths of 500 GB/s and more are not uncommon, whereas system RAM is limited to 40-50 GB/s.

2.1.2 CUDA Architecture

The main idea of CUDA is to have a large number of threads cooperatively solving one problem. This section explains how threads are organized into cooperative groups, how CUDA is embedded into C, and how CUDA achieves scalability.

Kernels

Kernels are special C functions that are executed on the GPU and are defined by the `__global__` keyword. In contrast to regular functions, kernels get executed N times in parallel by N different threads when called. Each thread is executing the same code, the kernel code, but on different data. The data on which a thread is working is determined by the thread ID. The call of a kernel from C is called a kernel launch. At launch the number of threads has to be specified in the launch configuration, by using the `<<<...>>>` launch configuration syntax.

Thread hierarchy

Whereas the kernel specifies the code that is executed by each thread, the thread hierarchy dictates how the individual threads are organized. CUDA has a two-level hierarchy to organize the threads. The top level is called *grid* and the second level is called *thread blocks*. A grid contains multiple blocks and a block contains multiple threads. The launch configuration of a kernel consists out of the grid size (number of blocks) and the block size (number of threads per block). Figure 2.1 illustrates an example of a 2D grid of size 3-by-2 with a 2D block of size 4-by-3.

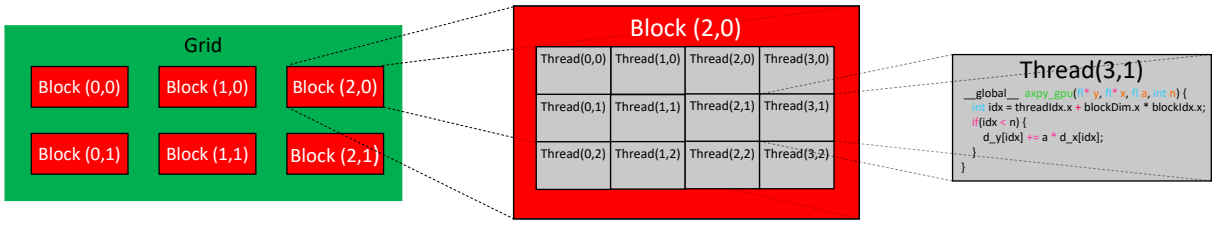

Figure 2.1: Illustration of the CUDA Thread Hierarchy, showing a 3x2 grid of blocks with a block size of 4x3 threads.

The threads within one thread block can cooperate to solve a subproblem. The problem has to be partitioned into independent subproblems by the programmer so that a grid of thread blocks can solve it in parallel. This decomposition of the problem enables automatic scaling of the problem. Each block will be scheduled on one of the available SMs. This can happen concurrently or sequentially, depending on the number of blocks and available hardware as illustrated by Figure 2.2. Note that several blocks can be scheduled on the same SM if there are enough resources available.

Threads within a block have a unique thread index that is accessible through the built-in variable `threadIdx`. It is defined as a 3-dimensional vector, so that threads can be indexed in one-dimension, on a grid, or in 3D space and forming a one-dimensional, two-dimensional, or three-dimensional block of threads. This allows for a natural indexing in the problem domain.

Similarly, thread blocks within the grid have a unique block index that is accessible through the variable `blockIdx`. It is also defined as a 3-dimensional vector and allows for one-, two-, or three-dimensional indexing of the blocks within the grid.

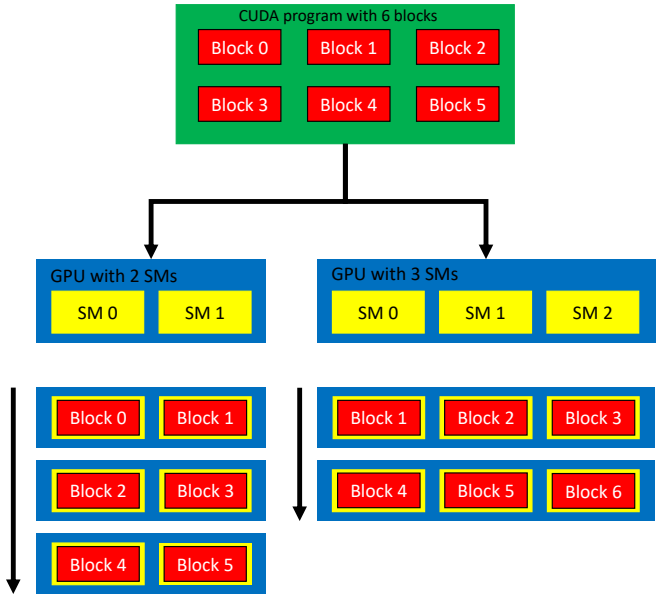

Figure 2.2: The CUDA Thread Blocks are scheduled on the available SMs of the given GPU.

Scheduling

As explained above, thread blocks are scheduled on available SMs during run-time which allows for automatic scaling. This does not mean that all threads of a block can run concurrently on the SM. Given a block to execute, the SM first partitions it into several groups called warps which then are scheduled by a warp scheduler for execution. A warp always has the same size, called the warp size. It is currently fixed at 32 threads per warp.

There are no guarantees on the order of block and warp execution. This enables applications to automatically scale from Desktop to High-Performance Compute GPUs. On the other hand, this introduces a non-deterministic element into the computations. In case of integer arithmetic the result is always deterministic, however for floating-point operations this is no longer true. Since floating point arithmetic is not associative, results may slightly differ from execution to execution.

Memory hierarchy

CUDA threads have access to multiple memory spaces during their execution. There is thread local memory that can only be accessed by one thread. Its lifetime is equal to the lifetime of the thread to which it belongs. Then, there is shared memory that can be accessed by all threads of

the same block. Its lifetime is equal to the lifetime of the thread block. Finally, there is global memory which can be accessed by any thread. Moreover, global memory is persistent across kernel launches.

The CPU can indirectly access global memory by initiating a memory transaction from system RAM to GPU memory or vice versa. Note that such memory transactions are very slow compared to global memory accesses.

Global memory access

Global memory is always accessed via memory transactions of size 32, 64, or 128 bytes. Furthermore, all these transactions have to be aligned to their size, *i.e.* a 32-byte memory transaction has to be aligned to 32-byte.

Threads can request data of word size 1-, 2-, 4-, 8-, or 16-byte from global memory. These memory requests are then coalesced into one or more of the above transactions on a per-warp basis (32 threads).

As an example, consider Figure 2.3 which shows a warp of 32 threads accessing memory from address 128 through 256 whereas each thread requests a different 4-byte word. These requests can be served with a single 128-byte transaction. Note that the 128-byte memory block does not need to be accessed sequentially by the threads in order to achieve full coalescing, *i.e.* one memory transaction per warp.

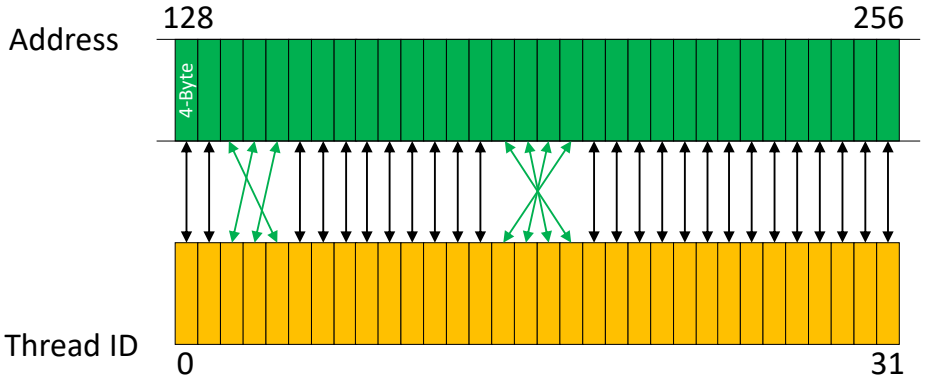

Figure 2.3: An example of a coalesced memory access of a warp to a 128-byte memory block. The green arrows represent non-sequential accesses.

Figure 2.4 shows the first quarter of a warp of 32 threads accessing the addresses 128, 144, 160, ..., *i.e.* with a stride of 16 bytes. Only the first 8 threads can be served by a single 128-byte transaction as shown in the plot. Thus three more transactions are needed to serve all requests. In the worst case the stride is larger than 128-byte and 32 separate memory transactions have to be issued. Strided memory access does not only increase the memory latency due to more transactions, but also reduces the effective bandwidth since only a fraction of the transmitted bytes are actually used. In the example above only 8 out of 32 4-bytes words that were transmitted was useful. Note that the same applies for random memory accesses to global memory.

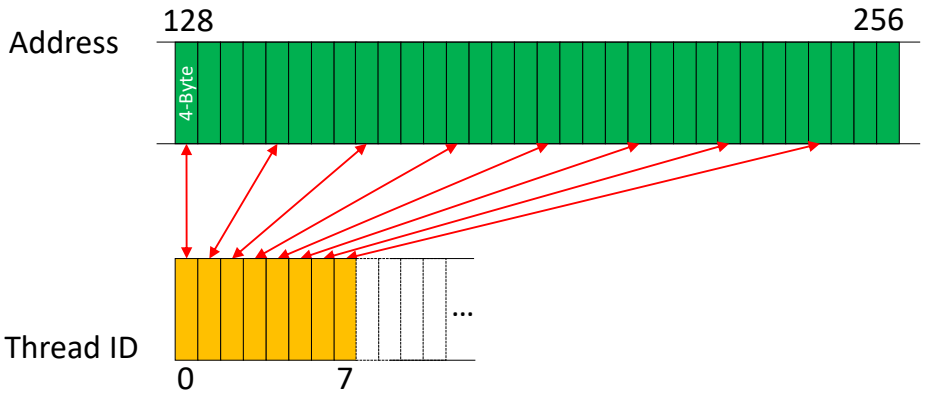

Figure 2.4: Example of a strided memory access with a stride of 4 elements or 16 bytes.

Naming conventions

To avoid confusion between CPU and GPU resources, CUDA introduces the terms host to refer to the CPU and device to refer to the GPU. For pointers referring to host memory the prefix `h_` is used and for pointers referring to device memory `d_` is used.

2.1.3 Accelerating Numerical Methods

Numerical methods make extensive use of floating-point operations, but their performance is not solely determined by the system's floating-point performance. Even though GPUs offer magnitudes larger raw floating-point power than CPUs, it is the memory bandwidth that limits the performance of many numerical operations [30]. Fortunately, GPUs also offer an order of magnitude larger memory bandwidth. However, utilizing the full potential is not an easy task. The parallel nature of the GPU requires different strategies and algorithms.

A simple example: `axpy`

The following code snippets implement the simple Basic Linear Algebra Subprograms (BLAS) routine `axpy` ($y = y + \alpha x$). Listing 2.1 implements the CPU version of `axpy`. It uses a simple `for` loop to iterate through all the elements in x and y.

```
1  void axpy_cpu(float* h_y, float* h_x, float alpha, int n) {
2      for(ind idx = 0; idx < n; ++idx) {
3          h_y[idx] += alpha * h_x[idx];
4      }
5  }
```

Listing 2.1: CPU implementation of `axpy`.

The GPU version of `axpy` is shown in Listing 2.2. It looks very similar to the CPU version, but with two important differences. The first and most important distinction is that the `for` loop has been replaced by a simple `if` statement. Instead of one thread iterating through a loop element by element, there is a thread for each element to be processed. This is a common pattern in GPU computing. The second difference is the usage of the thread ID to determine the data element which the given thread is operating on. The global thread ID is calculated from the local thread

index and the block index as shown in line 2. The `if` statement disables threads with a thread ID higher than the number of elements. This is necessary since threads are launched in blocks and the total number of threads usually does not match the number of elements. The cost of the few idle threads is negligible.

Figure 2.5 compares the achieved memory throughput of the CPU and the GPU implementation of the `axpy` operation. For large vector sizes, the simple GPU implementation is about 15 times faster. It also illustrates the fact that small problems cannot be accelerated well with GPUs. This is generally the case for numerical operations, not just for the given example. There are several reasons for this. First, small problems cannot fully utilize the GPU since there is simply not enough work to keep the GPU busy. Second, launching a kernel comes with a constant overhead that cannot be amortized for small problems.

The GPU reaches a maximal memory throughput of 522 GB/s (See Figure 2.5), which is 85% of its theoretical peak of 616 GB/s. Whereas, the peak number of floating point operations per second (FLOPS) is approximately 100 GFLOPS (not shown in Figure), which is less than 1% of its theoretical peak of 13500 GFLOPS. Thus, the performance is clearly limited by the memory bandwidth.

```
1  __global__ void axpy_gpu(float* d_y, float* d_x, float alpha, int n) {
2      int idx = threadIdx.x + blockDim.x * blockIdx.x;
3      if(idx < n) {
4          d_y[idx] += alpha * d_x[idx];
5      }
6  }
```

Listing 2.2: GPU implementation of `axpy`.

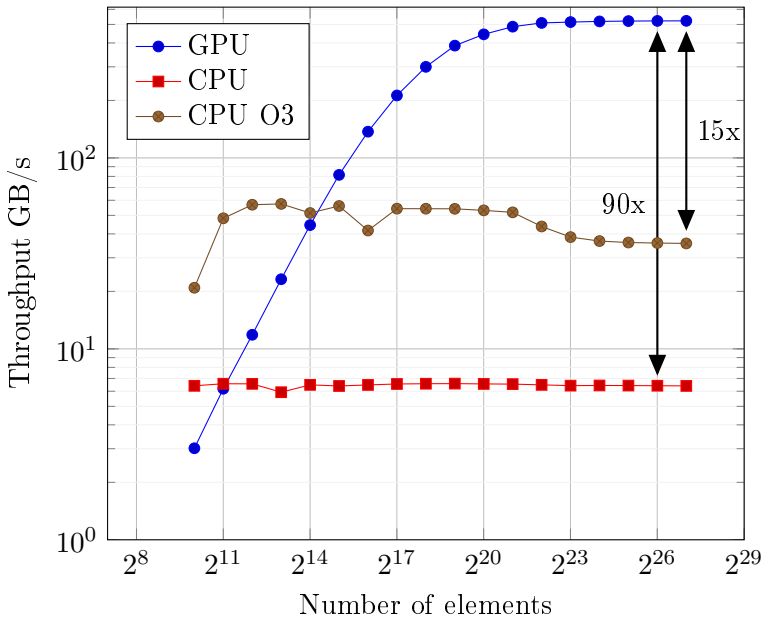

Figure 2.5: Average memory throughput as a function of vector size for `axpy` . blue: simple GPU implementation, red: simple CPU version, brown: CPU version with compiler optimization level O3.

Segmented reduction

A reduction is an operation that takes as inputs a vector x and an associative binary operator $\oplus$ and returns a scalar y [17]. The operator $\oplus$ is used to recursively reduce the number of elements until only one element remains, hence the name reduction operator. More formally, it performs the following calculation:

$$y = (\mathcal{I} \oplus (x_1 \oplus (x_2 \oplus (\ldots)))),$$

where $\mathcal{I}$ is the identity element of the binary operator. This general formulation allows for many different problems to be reformulated as a reduction, *e.g.*

- $x \oplus y \coloneqq x + y$ implements the sum over the elements of a vector.
- $x \oplus y \coloneqq \max(x, y)$ implements the maximum over the elements of a vector.
- $x \oplus y \coloneqq |x| + |y|$ implements the ℓ_1-norm of a vector.
- $x \oplus y \coloneqq \max(|x|, |y|)$ implements the ℓ_∞-norm of a vector.

The only difference between a reduction and a segmented reduction is that the latter reduces individual segments of the input vector and outputs a vector that contains the reduction results of the segments. Consider the following example,

$$x = \left[\begin{array}{cc|c|cc|cc} 1 & -2 & 3 & 2 & -4 & 8 & 9 & 2 \end{array}\right]$$
$$y = \begin{bmatrix} -1 & 3 & 6 & 11 \end{bmatrix}$$

where x is the input and y is the output and the scalar addition was used as the reduction operation. There exist very efficient parallel implementations for both versions of the reduction operation. Therefore, any problem that can be reformulated as one of them can be easily accelerated by a GPU.

2.2 CUDA Libraries

Fortunately, there are many libraries shipped with the CUDA toolkit that implement a wide range of functions on the GPU. This saves a huge amount of development time since one does not need to worry about the GPU implementation details. This section introduces the libraries and the functions used in this thesis.

2.2.1 cuBLAS

cuBLAS is the CUDA implementation of BLAS from Nvidia [8]. It enables easy GPU acceleration for applications that make use of BLAS. There are three different versions of the application programming interface (API): cuBLAS API, the standard API; cuBLASXT API, extends the functionality to multiple GPUs; and cuBLASLt API, a light weight library for general matrix-matrix products. Only the standard API is used in this thesis.

C-functions used

We use only level-1 functions, *i.e.* vector-vector operations.

- `cublasIsamax`
  - Used to calculate the ℓ_∞-norm of a vector
- `cublasSdot`
  - Calculates the inner product of two vectors
- `cublasSscal`
  - Scales a given vector by a scalar
- `cublasSaxpy`
  - Performs the `axpy` operation ($y = y + \alpha x$)

2.2.2 cuSPARSE

cuSPARSE is a CUDA library [10] that contains a set of linear algebra subroutines for handling sparse vectors and matrices. Only the sparse matrix subroutines are of interest to us. The cuSPARSE library requires the matrices to be in one of the following sparse matrix formats:

COO format

The coordinate (COO) format is one of the simplest sparse matrix formats. It is mainly used as an intermediate format to perform matrix operations, such as transpose, concatenation, or the extension of an upper triangular matrix to a full symmetric matrix.

The COO format consists of three arrays of size `nnz`, where `nnz` refers to the number of non-zero elements in the matrix.

`Value`	Array that holds the numerical values of the non-zero elements
`RowIndex`	Array that holds the row indices of the non-zero elements
`ColumnIndex`	Array that holds the column indices of the non-zero elements

The number of rows `m` and columns `n` has to be stored as well in order to have a complete description. The cuSPARSE API assumes that the indices are sorted by row indices first and ordered by column indices within one row. This makes the representation unique. For example, consider the 4×5 matrix

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 4 \\ 0 & 5 & 1 & 0 & 0 \\ 0 & 2 & 0 & 0 & 1 \\ 7 & 0 & 1 & 0 & 0 \end{bmatrix}$$

with 8 non-zero elements. Then A has the following representation in the COO format:

$$\begin{aligned} \texttt{Value} &= \begin{bmatrix} 1 & 4 & 5 & 1 & 2 & 1 & 7 & 1 \end{bmatrix} \\ \texttt{RowIndex} &= \begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 \end{bmatrix} \\ \texttt{ColumnIndex} &= \begin{bmatrix} 0 & 4 & 1 & 2 & 1 & 4 & 0 & 2 \end{bmatrix} \end{aligned}$$

This example uses zero-based indexing, which is also used in the rest of the thesis.

CSR format

The compressed sparse row (CSR) format only differs from the COO format in the row indices array, which is compressed in the CSR format. The compression can be understood conceptually as a two-step process. First, the number of non-zero elements in each row is determined from the row indices. This results in an array of length `m`. Secondly, the cumulative sum of this array is calculated and a zero is inserted at the beginning, resulting in an array of length `m+1`. The compressed array of row indices is called row pointer, since it points to the beginning of a row in the other two arrays. The row pointer has two important properties. First,

$$\texttt{RowPointer[m] = nnz},$$

and secondly, the difference between two consecutive elements

$$\texttt{RowPointer[k+1] - RowPointer[k]}$$

is equal to the number of non-zeros elements in row `k`.

Consider the same 4×5 matrix A again. It has the following representation in CSR format:

$$\begin{aligned} \texttt{Value} &= \begin{bmatrix} 1 & 4 & 1 & 2 & 1 & 1 \end{bmatrix} \\ \texttt{RowPointer} &= \begin{bmatrix} 0 & 2 & 4 & 6 & 8 \end{bmatrix} \\ \texttt{ColumnIndex} &= \begin{bmatrix} 0 & 4 & 1 & 2 & 1 & 4 & 0 & 2 \end{bmatrix} \end{aligned}$$

The CSR format is the format used for the computation of sparse matrix-vector multiplication (SpMV) in cuSPARSE. It provides good performance on average for generic sparse matrices. There are other sparse matrix formats suited for computation [11]. However, there is no native support by cuSPARSE for these formats.

CSC format

The compressed sparse column (CSC) format differs from the CSR format in two ways: the values are stored in column major format and the column indices are compressed. The compressed array of column indices is called column pointer since it points to the start of a column in the other two arrays. It contains `n+1` elements.

Consider the same 4×5 matrix A once again. It has the following representation in CSC format:

$$\begin{aligned} \texttt{Value} &= \begin{bmatrix} 1 & 7 & 5 & 2 & 1 & 1 & 4 & 1 \end{bmatrix} \\ \texttt{RowIndex} &= \begin{bmatrix} 0 & 3 & 1 & 2 & 1 & 3 & 0 & 2 \end{bmatrix} \\ \texttt{ColumnPointer} &= \begin{bmatrix} 0 & 2 & 4 & 6 & 6 & 8 \end{bmatrix}. \end{aligned}$$

The two consecutive 6s in the column pointer indicate an empty column in A. The CSC format is not directly used for computations on the GPU. Though, it is useful because the CSC representation can be reinterpreted in the CSR format yielding the CSR representation of A^T. This reinterpretation is done as follows:

$$\begin{aligned} \texttt{m}_{\text{CSC}} &\rightarrow \texttt{n}_{\text{CSR}} \\ \texttt{n}_{\text{CSC}} &\rightarrow \texttt{m}_{\text{CSR}} \\ \texttt{ColumnPointer} &\rightarrow \texttt{RowPointer} \\ \texttt{RowIndex} &\rightarrow \texttt{ColumnIndex} \\ \texttt{Value}_{\text{CSC}} &\rightarrow \texttt{Value}_{\text{CSR}} \end{aligned}$$

C-functions used

- `cusparseCsrmvEx`
  - Used for the SpMV in CSR format
  - Offers two implementations: naive row-based algorithm [4] and a merge-based approach [23]
- `cusparseXcoo2csr`
  - Performs the conversion from COO format to CSR format, *i.e.* compresses the row indices
- `cusparseXcsr2coo`
  - Performs the conversion from CSR format to COO format, *i.e.* expands the row pointer
- `cusparseCsr2cscEx2`
  - Transforms the CSR representation to a CSC representation, *i.e.* transposes the matrix
- `cusparseXcoosortByRow`
  - Sorts an unsorted COO format by row for conversion to the CSR format.

2.2.3 Thrust

Thrust is the CUDA C++ template library [28] based on the C++ standard template library (STL). It provides a high level interface for high-performance parallel applications. It provides all essential data parallel primitives, such as scan, sort, transform, and reduce.

C-functions used

- `reduce_by_key`
  - Implementation of a segmented reduction. A key has to be supplied for each value in the input vector. A segment is defined by consecutive identical keys.

2.3 GPU Specifications

2.3.1 Memory

Memory refers to the on-board RAM of the GPU. The most important metrics are the amount of memory and its bandwidth. The amount of memory limits the problem size that can be processed at once. The memory bandwidth limits the performance of kernels that are memory bound. Examples of memory bound kernels include: SpMV, reduction, `axpy` [30], [19].

Other influencing factors are the memory technology, the memory bus width, and the memory clock frequency. The memory bandwidth is related to the bus width and the memory frequency as:

$$\text{bandwidth} = 2 f_{\text{clk}} w_{\text{bus}},$$

where f_{clk} is the effective memory clock and w_{bus} is the bus width. The factor of 2 arises from the fact that the memory operates in double data rate (DDR) mode.

2.3.2 SM and CUDA Cores

The number of CUDA cores is determined by the number of SMs and the GPU generation. The Turing architecture features 64 CUDA cores per SM. This corresponds to 64 INT32 and 64 FP32 units on the Turing SM.

The number of SMs determines the number of thread blocks that can be executed concurrently. Assuming full utilization of the GPU, performance can scale with the number of SMs.

2.3.3 Nvidia GeForce RTX 2080 Ti

The GeForce RTX 2080Ti is Nvidia's latest high-end consumer GPU released in 2018. It will be used to perform the numerical tests in Chapter 5. Its technical specifications are shown in Table 2.1. Most notably, the double precision performance is a factor of 32 smaller than the single precision performance. This is due to the fact that each SM only has 2 FP64 units compared to the 64 FP32 units. As we will see in Section 5.2.6, this does not significantly hamper performance of memory bound kernels.

Memory	11 GB GDDR6
Memory Bandwidth	616 GB/s
Memory technology	GDDR6
Memory bus width	352 bits
Memory clock frequency	7000 MHz
Number of SMs	68
Number of CUDA Cores	4352
FP32 (float) performance	13.45 TFlops
FP64 (double) performance	0.420 TFlops

Table 2.1: Specification of the Nvidia RTX 2080Ti GPU.

Chapter 3

OSQP Solver

3.1 Algorithmic Description

3.1.1 Problem

OSQP [27] solves problems of the form

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}x^T P x + q^T x \\ \text{subject to} & l \leq Ax \leq u, \end{array} \tag{3.1}$$

where $x \in \mathbb{R}^n$ is the optimization variable. The Hessian of the objective function is a positive semidefinite matrix $P \in \mathbb{S}_+^n$. The linear cost part of the objective function is defined by the vector $q \in \mathbb{R}^n$. The constraints are given by the matrix $A \in \mathbb{R}^{m \times n}$, the lower bound $l : \{l_i \in \mathbb{R} \cup \{-\infty\}, i = 0, \ldots, m\}$, and the upper bound $u : \{u_i \in \mathbb{R} \cup \{\infty\}, i = 0, \ldots, m\}$. An equality constraint can be defined by setting $l_i = u_i$. For a feasible problem $l \leq u$ has to hold. This formulation is equivalent to problem (1.2).

The size of problem (3.1) is characterized by the tuple (m, n, N), where N is the sum of the non-zeros in A and P, *i.e.* $N = \text{nnz}(A) + \text{nnz}(P)$.

3.1.2 Optimality Conditions

OSQP introduces an auxiliary variable $z \in \mathbb{R}^m$, to obtain the following problem which is equivalent to (3.1):

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}x^T P x + q^T x \\ \text{subject to} & Ax = z \\ & l \leq z \leq u. \end{array} \tag{3.2}$$

The optimality conditions can then be written as

$$Ax - z = 0 \tag{3.3}$$

$$Px + q + A^T y = 0 \tag{3.4}$$

$$l \leq z \leq u, \quad y_+^T(z - u) = 0, \quad y_-^T(z - l) = 0, \tag{3.5}$$

where $y \in \mathbb{R}^m$ is the Lagrange multiplier associated with the equality constraint $Ax = z$ and $y_+ \coloneqq \max(y, 0)$ and $y_- \coloneqq \min(y, 0)$. The primal and dual residuals are defined as

$$r_{\text{prim}} \coloneqq Ax - z \tag{3.6}$$

$$r_{\text{dual}} \coloneqq Px + q + A^T y. \tag{3.7}$$

Certificates of primal and dual infeasibility

It can be shown that certifying primal infeasibility of problem (3.1) is equivalent to finding a vector $\bar{y} \in \mathbb{R}^m$ such that

$$A^T \bar{y} = 0, \quad u^T \bar{y}_+ + l^T \bar{y}_- < 0, \tag{3.8}$$

holds [3].

Similar to (3.8) it can be shown that finding an $\bar{x} \in \mathbb{R}^n$ that satisfies

$$P\bar{x} = 0, \quad q^T \bar{x} < 0, \quad (A\bar{x})_i \begin{cases} = 0 & l_i, u_i \in \mathbb{R} \\ \geq 0 & u_i = +\infty,\ l_i \in \mathbb{R} \\ \leq 0 & l_i = -\infty,\ u_i \in \mathbb{R} \end{cases} \tag{3.9}$$

certifies the dual infeasibility of the problem. For more details we refer the reader to [3].

3.1.3 Algorithm

OSQP uses ADMM [5] to solve (3.2), which results in Algorithm 1. A derivation can be found in [27].

Algorithm 1 OSQP

given initial values x^0, y^0, z^0 and parameters $\sigma > 0$, $\rho > 0$, $\alpha \in (0, 2)$

1: **repeat**
2: $\quad (\tilde{x}^{k+1}, \tilde{z}^{k+1}) \leftarrow \underset{(\tilde{x},\tilde{z}):A\tilde{x}=\tilde{z}}{\operatorname{argmin}} \ \frac{1}{2}\tilde{x}^T P\tilde{x} + q^T\tilde{x} + \frac{\sigma}{2}\|\tilde{x} - x^k\|_2^2 + \frac{\rho}{2}\|\tilde{z} - z^k + \rho^{-1}y^k\|_2^2$
3: $\quad x^{k+1} \leftarrow \alpha\tilde{x}^{k+1} + (1-\alpha)x^k$
4: $\quad z^{k+1} \leftarrow \Pi_{[l,u]}\left(\alpha\tilde{z}^{k+1} + (1-\alpha)z^k + \rho^{-1}y^k\right)$ ▷ $\Pi_{[l,u]}$ denotes the projection onto $[l, u]$
5: $\quad y^{k+1} \leftarrow y^k + \rho(\alpha\tilde{z}^{k+1} + (1-\alpha)z^k - z^{k+1})$
6: **until** termination condition is satisfied

The projection $\Pi_{[l,u]}$ onto the box $[l, u] \coloneqq \{z \in \mathbb{R}^m \mid l \leq z \leq u\}$ is defined as

$$\Pi_{[l,u]}(x) \coloneqq \underset{y\in[l,u]}{\operatorname{argmin}} \|y - x\|_2^2$$

and can be easily computed as

$$\Pi_{[l,u]}(x) = \min\left(\max(x, l), u\right),$$

where the min and max functions are evaluated element-wise.

The solution $(\tilde{x}^{k+1}, \tilde{z}^{k+1})$ of the equality-constrained problem in line 2 can be computed by solving the following linear system

$$\begin{bmatrix} P + \sigma I & A^T \\ A & -\rho^{-1}I \end{bmatrix} \begin{bmatrix} \tilde{x}^{k+1} \\ \nu^{k+1} \end{bmatrix} = \begin{bmatrix} \sigma x^k - q \\ z^k - \rho^{-1}y^k \end{bmatrix}, \tag{3.10}$$

with $\tilde{z}^{k+1}$ recoverable as

$$\tilde{z}^{k+1} = z^k + \rho^{-1}(\nu^{k+1} - y^k).$$

The coefficient matrix in (3.10) is referred to as the *KKT matrix*. This matrix is symmetric quasi-definite (SQD) for all $\rho > 0$ and $\sigma > 0$, *i.e.* it has the form

$$K = \begin{bmatrix} E & A^T \\ A & -F \end{bmatrix},$$

where E and F are positive definite. This property ensures that (3.10) is non-singular and has a well-defined LDL^T factorization [13]. Thus, (3.10) has a unique solution for all $P \in \mathbb{S}^n_+$ and $A \in \mathbb{R}^{m \times n}$. The LDL^T factorization is defined as

$$K = LDL^T$$

where D is a diagonal matrix and L is a lower triangular matrix with a unit diagonal. This factorization can then be used to solve

$$Kx = b$$

in a three-step process by introducing auxiliary variables y and $\hat{y}$. The first step is called forward solve and solves

$$Ly = b,$$

the second computes the solution of the diagonal system

$$D\hat{y} = y,$$

and the last step is called backward solve and solves

$$L^T x = \hat{y}.$$

The second step amounts to a simple element-wise multiplication of y with reciprocal of the diagonal elements of D, while the first and third steps require solutions of triangular systems, which is straightforward to compute.

Alternatively, (3.10) can be reformulated as

$$\left(P + \sigma I + A^T \rho A\right) \tilde{x}^{k+1} = \sigma x^k - q + A^T(\rho z^k - y^k) \tag{3.11}$$

by eliminating ν^{k+1}. The iterate $\tilde{z}^{k+1}$ can then be computed as $\tilde{z}^{k+1} = A\tilde{x}^{k+1}$. Note that the coefficient matrix in (3.11) is positive definite for all $P \in \mathbb{S}^n_+$ and $A \in \mathbb{R}^{m \times n}$. Therefore, the conjugate gradient (CG) method can be used to solve the linear system in an iterative fashion (See Section 4.1).

Note that the algorithm can easily be adapted to work with ρ being a positive definite diagonal matrix, in which case it is denoted by R. The scalar multiplications have to be replaced by matrix-vector products and the 2-norm has to replaced by the weighted 2-norm $\sqrt{x^T R x}$.

Warm starting

We can provide $(\hat{x}, \hat{z}, \hat{y})$ as an initial iterate to Algorithm 1 that is close to the actual solution $(x^\star, z^\star, y^\star)$ of (3.2). This is called *warm starting* and can drastically reduce the number of iterations. Typically, this is used when solving a sequence of similar problems where the solution of one problem is close to the one of the previous problem. Prominent applications are MPC, lasso regularization path, and portfolio back testing [6]. If no approximate solution is provided, all values are initialized to zero. This is called *cold starting*.

3.1.4 Convergence

The main convergence result of OSQP [27] shows that the sequence of iterates (x^k, z^k, y^k) generated by Algorithm 1 satisfies the optimality conditions (3.3)–(3.4) in the limit as $k \to \infty$ and condition (3.5) is satisfied by construction. This result only holds when problem (3.1) is solvable. Otherwise, it can be shown that the sequence

$$(\delta x^k, \delta y^k, \delta z^k) \coloneqq (x^k - x^{k-1}, y^k - y^{k-1}, z^k - z^{k-1})$$

always converges, and $\delta y = \lim_{k\to\infty} \delta y^k$ satisfies (3.8) if the problem is primal infeasible, and $\delta x = \lim_{k\to\infty} \delta x^k$ satisfies (3.9) if it is dual infeasible. For a more detailed discussion on the convergence of OSQP we refer the reader to [3].

Note that the convergence result holds for all $\rho > 0$ and $\sigma > 0$. This fact is used in OSQP to adapt the parameter ρ based on heuristics to reduce the total number of iterations needed.

3.1.5 Termination Criteria

The termination criteria of Algorithm 1 ensures that the algorithm stops when either a solution or an infeasibility certificate is found up to some tolerance. The proposed criterion for a solvable problem compares the norms of r^k_{prim} and r^k_{dual} against some tolerance levels, *i.e.*

$$\|r^k_{\text{prim}}\|_\infty \leq \varepsilon_{\text{prim}}, \quad \|r^k_{\text{dual}}\|_\infty \leq \varepsilon_{\text{dual}},$$

where the tolerance is set to

$$\begin{aligned}
\varepsilon_{\text{prim}} &\coloneqq \varepsilon_{\text{abs}} + \varepsilon_{\text{rel}} \max\{\|Ax^k\|_\infty, \|z^k\|_\infty\}, \\
\varepsilon_{\text{dual}} &\coloneqq \varepsilon_{\text{abs}} + \varepsilon_{\text{rel}} \max\{\|Px^k\|_\infty, \|A^T y^k\|_\infty, \|q\|_\infty\},
\end{aligned}$$

for some $\varepsilon_{\text{abs}}, \varepsilon_{\text{rel}} > 0$. In OSQP the default tolerance levels are set to $\varepsilon_{\text{abs}} = \varepsilon_{\text{rel}} = 10^{-3}$.

The following criterion is used to check for primal infeasibility

$$\begin{aligned}
u^T(\delta y^k)_+ + l^T(\delta y^k)_- &\leq \varepsilon_{\text{pinf}} \|\delta y^k\|_\infty \\
\|A^T \delta y^k\|_\infty &\leq \varepsilon_{\text{pinf}} \|\delta y^k\|_\infty,
\end{aligned}$$

where $\varepsilon_{\text{pinf}} > 0$ is some tolerance level. The criterion for dual infeasibility is defined similarly as

$$q^T \delta x^k \leq \varepsilon_{\text{dinf}} \|\delta x^k\|_\infty,$$

$$\|P\delta x^k\|_\infty \leq \varepsilon_{\text{dinf}} \|\delta x^k\|_\infty,$$

$$(A\delta x^k)_i \begin{cases} \in [-\varepsilon_{\text{dinf}}, \varepsilon_{\text{dinf}}] \, \|\delta x^k\|_\infty & l_i, u_i \in \mathbb{R} \\ \geq -\varepsilon_{\text{dinf}} \|\delta x^k\|_\infty & u_i = +\infty \\ \leq \varepsilon_{\text{dinf}} \|\delta x^k\|_\infty & l_i = +\infty \end{cases}$$

where $\varepsilon_{\text{dinf}} > 0$ is the dual infeasibility tolerance level. OSQP sets the default infeasibility tolerance levels to $\varepsilon_{\text{dinf}} = \varepsilon_{\text{pinf}} = 10^{-4}$.

3.2 Implementation Details

3.2.1 Preconditioning

First-order methods are known to be sensitive to the problem scaling, *i.e.* their convergence rate can vary significantly for ill-conditioned problems. To reduce the sensitivity to ill-conditioned problems, a technique called *preconditioning* is often used. Preconditioning improves the convergence rate by reducing the condition number of the problem. For example, the optimal diagonal preconditioner minimizes the condition number of a matrix. However, to find the optimal diagonal preconditioner, we need to solve an SDP which is typically harder to solve than the original QP. Thus, it is seldom worth to calculate it.

To circumvent this issue, OSQP uses a simple heuristic to compute the preconditioner called *matrix equilibration*. The goal is to find a diagonal scaling $S \in \mathbb{S}^{m+n}_{++}$ of the problem data such that the columns of SMS have the same ℓ_∞ norms, where M is defined as

$$M \coloneqq \begin{bmatrix} P & A^T \\ A & 0 \end{bmatrix}, \tag{3.12}$$

and the scaling S is defined as

$$S \coloneqq \begin{bmatrix} D & 0 \\ 0 & E \end{bmatrix},$$

where $D \in \mathbb{S}^n_{++}$ and $E \in \mathbb{S}^m_{++}$ are both diagonal and positive definite matrices.

The matrix equilibration heuristic used by OSQP is based on the *Ruiz equilibration*.

Algorithm 2 Modified Ruiz equilibration

given A, P, q, l, u
initialize $c = 1$, $E = I$, $D = I$, $\delta = 0$, $\bar{P} = P$, $\bar{A} = A$, $\bar{l} = l$, $\bar{u} = u$
for iter $= 1,\ldots,$#iterations **do**
  for $i = 1, \ldots, n+m$ **do**
    $\delta_i \leftarrow 1/\sqrt{\|M_i\|_\infty}$ ▷ M_i is the i-th column of M
  end for
  $\bar{D} \leftarrow \mathbf{diag}(\delta_1, \ldots, \delta_n)$, $\bar{E} \leftarrow \mathbf{diag}(\delta_{n+1}, \ldots, \delta_{n+m})$
  $\bar{P} \leftarrow \bar{D}\bar{P}\bar{D}$, $\bar{A} \leftarrow \bar{E}\bar{P}\bar{D}$, $\bar{q} \leftarrow \bar{D}\bar{q}$, $\bar{l} \leftarrow \bar{E}\bar{l}$, $\bar{u} \leftarrow \bar{E}\bar{u}$
  $\lambda \leftarrow 1/\max\{\mathrm{mean}(\|\bar{P}_i\|_\infty), \|\bar{q}\|_\infty\}$ ▷ $\bar{P}_i$ is the i-th column of $\bar{P}$
  $\bar{P} \leftarrow \lambda\bar{P}$, $\bar{q} \leftarrow \lambda\bar{q}$
  $D \leftarrow \bar{D}D$, $E \leftarrow \bar{E}E$, $c \leftarrow \lambda c$
end for
$S \leftarrow \mathbf{diag}(D, E)$
return S, c

The modified Ruiz procedure is described in Algorithm 2. It performs a fixed number of equilibration iterations. Each iteration performs the following steps: First the column norms of M are calculated and problem data is scaled by the inverse square root of the norms accordingly. The second step normalizes the cost function by scaling it by the inverse of the maximum of the average column norm of $\bar{P}$ and the norm of $\bar{q}$.

Scaling the matrices P and A effectively results in a transformed problem given by

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}\bar{x}^T\bar{P}\bar{x} + \bar{q}^T\bar{x} \\ \text{subject to} & \bar{l} \leq \bar{A}\bar{x} \leq \bar{u}, \end{array}$$

where $\bar{x} = D^{-1}x$, $\bar{P} = cDPD$, $\bar{q} = cDq$, $\bar{A} = EAD$, $\bar{l} = El$ and $\bar{u} = Eu$. The scaled dual variable is given by $\bar{y} = cE^{-1}y$ and the scaled residuals are

$$\bar{r}_{\text{prim}} \coloneqq Er_{\text{prim}}, \quad \bar{r}_{\text{dual}} \coloneqq cDr_{\text{dual}}.$$

The Ruiz equilibration is usually stopped when $\text{diag}(\delta_1, \ldots, \delta_{n+m})$ is close to the identity matrix by some measure, *i.e.* $\|\text{diag}(\delta_1, \ldots, \delta_{n+m}) - I\| \leq \varepsilon$. However, OSQP always performs 10 iterations by default, which is sufficient in most cases.

3.2.2 Step-Size Parameter Update

As already mentioned in Section 3.1.4, OSQP uses a heuristic to update the step-size parameter ρ or R. Moreover, $R \in \mathbb{S}^m_{++}$ is chosen to be a positive definite diagonal matrix with different elements R_i depending on the constraint type. The diagonal elements are defined by

$$R_i = \begin{cases} \bar{\rho}, & l_i \neq u_i \\ 10^3\bar{\rho}, & l_i = u_i, \end{cases}$$

where $\bar{\rho} > 0$ is a scalar parameter, *i.e.* the step-size related to an equality constraint is assigned a larger value. The value of $\bar{\rho}$ is not fixed, but is updated based on the ratio of the dual and primal residuals. The following rule

$$\bar{\rho}^{k+1} \leftarrow \bar{\rho}^k \sqrt{\frac{\|\bar{r}^k_{\text{prim}}\|_\infty / \max\{\|\bar{A}\bar{x}^k\|_\infty, \|\bar{z}^k\|_\infty\}}{\|\bar{r}^k_{\text{dual}}\|_\infty / \max\{\|\bar{P}\bar{x}^k\|_\infty, \|\bar{A}^T\bar{y}^k\|_\infty, \|\bar{q}\|_\infty\}}}$$

is used to adapt $\bar{\rho}$. The initial value is set as $\bar{\rho}^0 = 0.1$. Note that this rule uses the scaled version of the residuals.

The update of ρ changes the KKT matrix in (3.10) and requires a numerical re-factorization. To save on computation cost, this is only done if it is really necessary. OSQP requires the accumulated iteration time to be greater than a certain percentage of the initial factorization time and the new value to be larger or smaller than the old one by a certain threshold.

3.3 Bottlenecks

This section identifies and analyzes the computational hot spots in OSQP and evaluates to what extent they can be accelerated by parallelization. Profiling the runtime shows that all routines and operations whose execution time scales with the total number of non-zeros N represent a potential bottleneck.

3.3.1 Linear System Solver

The most severe bottleneck is the linear system solver used to calculate the solution to the KKT system (3.10). To see why, we will look into the LDL^T factorization a bit closer. Consider the following linear system

$$Qx = b, \tag{3.13}$$

where Q and b are an SQD matrix and a vector given by

$$Q = \left[\begin{array}{cc|c} 2 & 2 & 0 \\ 2 & 4 & 1 \\ \hline 0 & 1 & -1 \end{array}\right], \quad b = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}.$$

The LDL^T factorization of Q is

$$Q = LDL^T = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0.5 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & -1.5 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0.5 \\ 0 & 0 & 1 \end{bmatrix}. \tag{3.14}$$

To solve (3.13) with the help of LDL^T factorization (3.14) we first perform the forward solve

$$Ly = b, \tag{3.15}$$

i.e.

$$\begin{aligned} y_1 &= b_1 && = 1 \\ y_2 &= b_2 - y_1 && = -1 \\ y_3 &= b_3 - 0.5y_2 && = 1.5. \end{aligned}$$

Then, we perform the backward solve

$$L^T x = \hat{y} = D^{-1} y, \tag{3.16}$$

i.e.

$$\begin{aligned} x_3 &= \hat{y}_3 && = -1 \\ x_2 &= \hat{y}_2 - 0.5x_3 && = 0 \\ x_1 &= \hat{y}_1 - x_2 && = 0.5. \end{aligned}$$

There are two bottlenecks in (3.15) and (3.16). First, to compute y_i we need to have computed $y_1, \ldots, y_{i-1}$ in advance. This dependency on the previous elements makes the forward solve hard to parallelize. The same argument holds for the backward solve, where we need to compute x_{i+1} before computing x_i. The second bottleneck is that the backward solve can only start after the forward solve has been performed since the backward solve starts with y_n which is calculated last in the forward solve. This bottleneck is not pronounced for small to medium sized problems, but hinders the scaling to large-scale problems.

3.3.2 Termination Criteria Evaluation

The evaluation of termination criteria described in Section 3.1.5 requires the computation of three matrix products $Px^k, Ax^k, A^T y^k$. The computation cost of each of the products scales with the number of non-zeros in the respective matrices. OSQP reduces this cost by only checking the termination every k iterations ($k = 25$ by default).

The evaluation of the primal and dual infeasibility criterion requires additional three matrix-vector products $P\delta x^k$, $A\delta x^k$, $A^T \delta y^k$. However, OSQP uses the fact that all the conditions of the criterion have to be met and checks computationally cheaper conditions first. Thus, some of the matrix products do not have to be calculated every time.

3.3.3 Scaling

Profiling reveals that the matrix equilibration procedure requires a substantial amount of time for larger problems. Specifically, the column norm calculations in line 4 and the scaling of the matrices A and P in line 7 of Algorithm 2 determine the runtime. The matrix scaling requires two matrix-matrix multiplication, one pre-multiplication and one post-multiplication. Fortunately, the scaling matrices are diagonal and the matrix-matrix multiplications are equivalent to scaling the rows or columns of A and P. Nevertheless, each matrix-matrix multiplication requires $\text{nnz}(G)$ multiplications, where G is the sparse matrix to be scaled.

3.3.4 Potential for Acceleration

Since the direct solver approach with the LDL^T factorization is not suited to be parallelized, an iterative solution strategy, *i.e.* an indirect solver, sounds as a more promising approach. Indirect methods do not posses any dependencies between the solution components that restrict direct methods, such as LDL^T factorization, from being parallelized. Furthermore, iterative methods can more easily make use of the sparsity of a problem than direct solvers. Since direct solvers typically have to perform a matrix permutation before the factorization to exploit sparsity, while indirect methods only need to perform SpMV. This makes them well suited for sparse large-scale problems.

An additional benefit of an indirect solver is that updating (3.10) does not require a new factorization. This is especially helpful for adapting the step-size parameter ρ which has been shown to help reduce the number of iterations.

The main operation of an indirect solver is the SpMV which can be efficiently implemented on a GPU. Thus, indirect solvers are best suited to accelerate the linear system solver on the GPUs.

In the second step all the sparse matrix-vector products used in the evaluation of the termination criterion can be replaced by GPU routines. The speedup of the termination check can then be used to check the termination conditions more often and thus potentially stop the algorithm earlier.

The column norms of the matrix M that is used in the matrix equilibration can be calculated in parallel with a segmented reduction. Furthermore, the pre- and post-multiplication of a diagonal matrix with a sparse matrix can also be parallelized.

Chapter 4

GPU Acceleration

4.1 Linear System Solver and Conjugate Gradient Method

Throughout this chapter we consider linear systems of equations of the following form

$$Kx = b \tag{4.1}$$

where $K \in \mathbb{S}^s_{++}$ is a symmetric positive definite matrix, $x \in \mathbb{R}^s$ is the solution and $b \in \mathbb{R}^s$ is referred to as the right-hand side.

4.1.1 The Conjugate Gradient Method

The CG method is an iterative method for solving linear systems of equations of the form (4.1). Theoretically, it can compute an exact solution in s iterations [25, Chap. 5]. However, in practice one aims to terminate the method long before s. This yields an approximate solution to the system (4.1). The CG method is especially well suited to solve sparse large-scale problems, such as those arising in OSQP, since its iterations can be easily parallelized.

Problem (4.1) can be reformulated as an unconstrained optimization problem:

$$\text{minimize} \quad \varphi(x) \coloneqq \tfrac{1}{2}x^T K x - b^T x,$$

which can be easily seen by looking at the gradient of φ

$$\nabla\varphi(x) = Kx - b =: r(x).$$

Rewriting $\varphi(x^k)$ as

$$\varphi(x^k) = \tfrac{1}{2}\|x^k - x^\star\|_K^2 + \varphi(x^\star),$$

where the K-norm is defined by

$$\|v\|_K = \sqrt{v^T K v}$$

and $x^\star$ is the minimizer of $\varphi(x)$, shows that an iterative minimization of $\varphi(x)$ yields a sequence of ever-better approximations to (4.1) as measured in the K-norm [14, §11.3].

Conjugate directions

A corner stone of the CG method is its ability to cheaply generate a set of minimization directions with property called conjugacy. A set of non-zero vectors $\{p^0, p^1, \ldots, p^s\}$ is conjugate with respect to the symmetric positive definite matrix K if

$$(p^i)^T K p^j = 0, \quad \forall i \neq j.$$

Successively minimizing along the conjugate directions p^k, *i.e.*

$$\begin{aligned} \alpha^k &= \operatorname*{argmin}_{\alpha} \varphi(x^k + \alpha p^k) \\ x^{k+1} &= x^k + \alpha^k p^k \end{aligned} \tag{4.2}$$

produces approximations x^k to $x^\star$ that minimize φ over the expanding subspace S^k which is spanned by the previous conjugate directions $\{p^0, p^1, \ldots, p^k\}$. From the expanding subspace we have that $\varphi(x^k) \leq \varphi(x^{k-1})$ and thus x^k is a better approximation of the solution to (4.1) [25, Thm. 5.2]. The minimization step in (4.2) can be evaluated analytically as

$$\alpha^k = -\frac{(Kx^k - b)^T p^k}{(p^k)^T K p^k}.$$

which is very easy to compute.

The residual at step k is defined as $r^k \coloneqq Kx^k - b$.

Conjugate gradient

There are many choices for the conjugate direction set $\{p^0, p^1, \ldots, p^{s-1}\}$. For example, the eigenvectors of K form a conjugate set of vectors with respect to K. However, it is impractical to compute the eigenvectors for large-scale systems.

The conjugate gradient method is conjugate direction method that can compute a new direction p^k based on the previous vector p^{k-1} only. This property allows CG to have a very low memory and computation requirement for generating the conjugate set, since p^{k+1} can be generated on the fly. The new direction p^k is calculated as a linear combination of the negative gradient $-r^k$ and the previous direction p^{k-1}.

$$p^k = -r^k + \beta^k p^{k-1}.$$

The scalar β^k is determined by the conjugate requirement $(p^k)^T K p^{k-1} = 0$ as

$$\beta^k = \frac{(r^k)^T K p^{k-1}}{(p^{k-1})^k K p^{k-1}}. \tag{4.3}$$

The first conjugate direction p^0 is set to be in the negative gradient direction *i.e.* $p^0 = -r^0$. Combining the successive minimization along the conjugate directions (4.2) and the computation of new conjugate directions (4.3) results in the CG method.

Preconditioning

The CG method is a first-order method and thus its convergence rate is sensitive to the problem scaling. To improve the convergence of the CG method, the linear system can be preconditioned with a coordinate transformation

$$\hat{x} = Cx$$

where C is a non-singular matrix. Applying the CG method to the transformed variable $\hat{x}$ and then inverting the transformation to express all equations in terms of x results in the preconditioned conjugate gradient (PCG) method described in Algorithm 3 [25, Alg. 5.3]. It turns out that C is not explicitly needed, but rather acts through $M = C^T C$. Setting the preconditioner $M = I$ to the identity recovers the regular CG method.

Algorithm 3 PCG method

given x^0, preconditioner M
1: **initialize** $r^0 \leftarrow Kx^0$, $y^0 \leftarrow M^{-1}r^0$, $p^0 \leftarrow -y^0$, $k \leftarrow 0$
2: **while** $\|r^k\| \leq \varepsilon\|b\|$ **do**
3: $\quad \alpha^k \leftarrow -\frac{(r^k)^T y^k}{(p^k)^T K p^k}$
4: $\quad x^{k+1} \leftarrow x^k + \alpha^k p^k$
5: $\quad r^{k+1} \leftarrow r^k + \alpha^k K p^k$
6: $\quad y^{k+1} \leftarrow M^{-1} r^{k+1}$
7: $\quad \beta^k \leftarrow \frac{(r^{k+1})^T y^{k+1}}{(r^k)^T y^k}$
8: $\quad p^{k+1} \leftarrow -y^{k+1} + \beta^{k+1} p^k$
9: $\quad k \leftarrow k + 1$
10: **end while**

Generally, a good preconditioner should capture the essence of the original matrix, *i.e.* $M \approx K$ and it should be easy to solve the linear system $My = r$ [14, §11.5]. There are many strategies for choosing the preconditioner M, the simplest being the Jacobi preconditioner, also known as the diagonal preconditioner. It simply contains the diagonal elements of K, which makes it extremely cheap to compute.

Other more powerful preconditioner include the incomplete Cholesky, the incomplete LU and the polynomial preconditioners. The incomplete preconditioner produce an approximate decomposition of K with a high sparsity, such that solving $My = r$ is cheap. The family of polynomial preconditioners include the Chebyshev and least square polynomial preconditioner [2]. Both of them require bounds on the spectrum of K, *i.e.* the smallest and largest eigenvalue.

4.2 Solving the KKT System

We are interested in solving the KKT system defined in (3.10). Since the conjugate gradient method requires a positive definite system, the reduced form of the KKT matrix (3.11) is used. Note that we use the general diagonal matrix formulation of ρ, *i.e.* R, throughout this chapter.

4.2.1 Implementation Details

Note that the reduced KKT matrix (3.11) is not explicitly needed. It is sufficient to be able to apply the effect of a matrix multiplication to a vector. This can be achieved by the following steps

$$\begin{aligned} z &\leftarrow RAx \\ y &\leftarrow Px + \sigma x + A^T z, \end{aligned} \tag{4.4}$$

where $z \in \mathbb{R}^m$ is a temporary variable.

Matrix storage The preferred way of storing the matrices P and A on the GPU is the CSR format since it has a superior SpMV performance on the GPU. However, in order to implement (4.4) a matrix-vector multiplication with A^T is required. Using the CSR format to perform this operation is about 10x slower than the multiplication with A [10, §4.6]. This can be mitigated by using the CSC format to store A, which can be interpreted as the CSR representation of A^T. Though, this means that we store the matrix A once as CSR and once as CSC. In other words, A and A^T are stored in CSR.

The matrix P is symmetric by definition and thus only the upper triangular part needs to be stored. However, on the GPU a performance penalty has to be paid during the matrix-vector product operation when using upper triangular matrices [10, §4.6]. Thus, the matrix is stored in its full representation, *i.e.* upper and lower triangular parts and the diagonal. Furthermore, σI is stored together with P, *i.e.* we store $P + \sigma I$ in CSR. This step is not done for performance reason, but to keep the PCG solver agnostic to the problem structure.

Preconditioner The incomplete preconditioners require the explicit matrix representation and thus are not a viable option. Thus, the Jacobi (diagonal) preconditioner seems as a better option. The solution to the system $My = r$ amounts to a simple diagonal matrix-vector product. The diagonal of the Jacobi preconditioner of the reduced KKT system can be written as

$$\operatorname{diag}(M) = \operatorname{diag}(P + \sigma I) + \rho_{\text{ratio}} \operatorname{diag}(A^T R_0 A), \quad \rho_{\text{ratio}} \in \mathbb{R}_{++}$$

The calculation of the preconditioner is split into two parts. First, the diagonal of the matrix $P + \sigma I$ is stored. Then the diagonal of the product $A^T R_0 A$ has to be calculated for some initial value R_0 such that $R = R_0 \rho_{\text{ratio}}$ holds. This allows the preconditioner to be updated quickly without re-evaluating the diagonal of the product again in case R is updated. Note that the product only has to be evaluated on the diagonal, and thus the full product is not required. The diagonal entries a_{ii} of $A^T R_0 A$ can be calculated from the weighted 2-norm of the columns of A as

$$a_{ii} = \|A_i\|_R^2, \quad \text{where} \quad \|v\|_R^2 = v^T R v.$$

This can be implemented with a segmented reduction with the addition operator(*i.e.* segmented sum) on the GPU. A temporary array of size $\operatorname{nnz}(A)$ is allocated for this purpose and used to store the result of the pre-multiplication of R and A followed by an element-wise multiplication with the values of A. Then a segmented sum, with segments defined as the rows of A, is used to sum up the scaled square term in each row.

Termination criteria and warm starting The minimization step, *i.e.* the solution of the KKT system (3.10), does not need to be carried out exactly for ADMM to converge [5]. This fact is used to motivate an early termination of PCG. This means that the PCG algorithm only performs a small number of iterations (typically less than 10) in each ADMM iteration to get a rough approximation at first. In successive ADMM iterations the PCG solver is then warm started, *i.e.* x^0 is initialized to the previous solution $\tilde{x}^{k-1}$ of the PCG-solver. This allows the solver to improve the approximate solution further. As ADMM progresses and the right-hand side of the KKT system does not change significantly, the PCG algorithm obtains more and more accurate solutions.

Finding a good termination criterion for the PCG algorithm is a trade-off between the number of ADMM iterations and the number of PCG iterations per ADMM iteration. Since PCG iterations

are still expensive compared to the rest of the ADMM algorithm, minimizing the total number of PCG iterations is the goal. The following strategies are based on checking the norm of the PCG residual r^k against some tolerance $\varepsilon > 0$, that is adapted as ADMM progresses.

SCS strategy This strategy has been adopted from the SCS [26] solver implementation. It features a monotonic decrease of the tolerance as ADMM progresses. It exploits the fact that ADMM does not require very accurate solutions at the beginning and can save a lot of PCG iterations at the start. The termination condition is defined as

$$\|r^k\|_2 \leq \varepsilon, \quad \text{with}\, \varepsilon = \max\{\|b\|_2 \frac{\varepsilon_{\text{start}}}{k^r}, \varepsilon_{\text{min}}\},$$

where k is the ADMM iteration counter, $r > 0$ is a decrease parameter, b is the right-hand side of the reduce KKT system (3.11), and $\varepsilon_{\text{min}}/\varepsilon_{\text{start}}$ are the minimum and starting tolerances respectively. The following parameters have been found to work well for a variety of problems:

$$\begin{aligned} \varepsilon_{\text{start}} &= 50 \\ \varepsilon_{\text{min}} &= 10^{-7} \\ r &= 2.75. \end{aligned}$$

These parameters have been determined by running the OSQP benchmarks (See Appendix A).

Residual strategy This strategy chooses the tolerance, as the name suggests, according to the current ADMM residuals. The termination condition is defined very similarly to the previous strategy. Instead of the 2-norm, the ∞-norm is used to match the norm used for the ADMM residuals. The termination condition is then given as

$$\|r^k\|_\infty \leq \varepsilon,$$

and ε is calculated as

$$\begin{aligned} \varepsilon &\leftarrow \min\{\lambda \sqrt{\|\bar{r}_{\text{prim}}\|_\infty \|\bar{r}_{\text{dual}}\|_\infty}, \varepsilon\} \\ \varepsilon &\leftarrow \max\{\varepsilon, \varepsilon_{\text{min}}\}, \end{aligned}$$

where $\bar{r}_{\text{prim}}$ and $\bar{r}_{\text{dual}}$ are the scaled primal and dual ADMM residuals respectively. The parameter $\lambda > 0$ ensures that the accuracy of the iterative solver is always higher than the geometric mean of the primal and dual residuals. In case the new tolerance value is larger than the old one, the old value is used again. The result is a monotonically decreasing tolerance over the progress of ADMM. Additionally, λ is reduced by $\lambda_{\text{reduction}}$ after the number of ADMM iterations with zero PCG iterations exceeds the threshold $k_{\text{threshold}} \in \mathbb{N}$. Successive ADMM iterations with zero PCG iterations suggest that the solution has already been found or that the chosen tolerance is too large. As before, the following parameters have been found to work well for a variety of problems:

$$\begin{aligned} \lambda &= 0.17 \\ \varepsilon_{\text{min}} &= 10^{-7} \\ k_{\text{threshold}} &= 10 \\ \lambda_{\text{reduction}} &= 2. \end{aligned}$$

The parameters have been determined by running the OSQP benchmarks (See Appendix A).

Furthermore, the number of PCG iterations is restricted to $n_{\text{max}} = 25$ iterations for both strategies.

4.3 Further Acceleration

4.3.1 ADMM Residual Calculation

The calculation of the primal and dual residuals (3.6) and (3.7) requires several SpMVs, which are very expensive operations compared to the rest of ADMM (steps 3-5 of Algorithm 1). Offloading these calculation to the GPU has several benefits. First, it reduces the time to calculate the residuals. Second, it allows to check the termination criterion more often and thus allows an earlier termination. Another benefit is the opportunity to adapt $\bar{\rho}$ more often (See Section 4.3.3).

The GPU implementation of the residual calculation is invoked from the check termination subroutine and performs the following steps:

1. Copy the current iteration $(\bar{x}^k, \bar{y}^k, \bar{z}^k)$ to the GPU.
2. Calculate the scaled primal residual $\bar{r}^k_{\text{prim}} = \bar{A}\bar{x}^k - \bar{z}^k$ and its norm $\|\bar{r}^k_{\text{prim}}\|_\infty$
   (a) In case scaled termination is disabled, calculate the unscaled primal residual $r^k_{\text{prim}} = E^{-1}\bar{r}^k_{\text{prim}}$ and its norm $\|r^k_{\text{prim}}\|_\infty$.
3. Calculate the scaled dual residual $\bar{r}^k_{\text{dual}} = \bar{P}\bar{x}^k + \bar{q} + \bar{A}^T\bar{y}^k$ and its norm $\|\bar{r}^k_{\text{dual}}\|_\infty$
   (a) In case scaled termination is disabled, calculate the unscaled primal residual $r^k_{\text{dual}} = c^{-1}D^{-1}\bar{r}^k_{\text{dual}}$ and its norm $\|r^k_{\text{dual}}\|_\infty$.
4. Copy the results of the matrix-vector products $(\bar{A}\bar{x}^k, \bar{P}\bar{x}^k, \bar{A}^T\bar{y}^k)$ back to the CPU.

The last step is necessary since other subroutines of OSQP require the results of matrix-vector products to work properly.

The diagonal inverse scaling matrices E^{-1} and D^{-1} and the linear cost vector $\bar{q}$ are copied to the GPU during the initialization phase. The two matrices and the vector are updated on the GPU in case the scaling changes.

4.3.2 Scaling / Ruiz Equilibration

The runtime of the modified Ruiz equilibration (Algorithm 2) is determined by the column-norm calculation and the post/pre matrix multiplication as described in Section 3.3. It is thus essential to accelerate these three operations on the GPU to reduce the total runtime considerably.

Accelerating the column norm calculation

The CSC format allows for an efficient way of calculating the column norms of a sparse matrix. The column pointer directly defines segments of the value array that correspond to individual columns. Conversely the CSR format allows for an efficient way of calculating row norms for the same reasons. Since the matrix M defined in (3.12) is symmetric, calculating the row norms is equivalent to calculating column norms. This allows us to use the CSR format to calculate the column norms.

The naive approach would be to have one thread per row calculating the norm of its row. This approach suffers from several inefficiencies. First the work load is poorly distributed among

the threads since one thread can have zero elements in its row and another thread can have n. Secondly, the memory is accessed almost randomly as each thread iterates through its row. Thus, a more systematic solution has to be used.

The problem of calculating the row norms of a CSR matrix can be stated in the form of a segmented reduction, with the segments defined by the row pointer. In the case of the ℓ_∞ norm, the reduction operator $\oplus$ has to be the maximum of the absolute values, *i.e.*

$$x_1 \oplus x_2 \coloneqq \max(|x_1|, |x_2|).$$

Special care has to be taken for segments with only one element. Since there is only one element to reduce, the reduction returns that element without invoking the reduction operator depending on the implementation. Therefore, the absolute value operator has to be applied to each element resulting from the segmented reduction to ensure that each element corresponds to the desired row ℓ_∞ norms.

There are several CUDA implementations of the segmented reduction operation available.

Thrust Thrust [28] offers a reduce-by-key operation, which essentially performs a segmented reduction. It requires one key per value element, and a segment is defined by consecutive identical keys.

$$\begin{aligned} \texttt{value} &= \left[\begin{array}{cc|ccc|ccc} 0 & 4 & 1 & 2 & 1 & 4 & 0 & 2 \end{array}\right] \\ \texttt{key} &= \left[\begin{array}{cc|ccc|ccc} 0 & 0 & 1 & 1 & 1 & 3 & 3 & 3 \end{array}\right]. \end{aligned}$$

The row index of a matrix stored in coordinate format defines one segment per row using the above definition. Thus, the row index is identical for each element within one row and all elements of one row are stored consecutively. This assumes that the matrix is sorted by row. Unfortunately, the CSR format does not store the row index. Therefore, the row index has to be calculated from the row pointer in advance of the segmented reduction. This is one drawback of the reduce-by-key method offered by Thrust. Another drawback is that empty segments, which correspond to empty rows, cannot be defined. This issue is overcome by inserting a zero into the result vector of reduce-by-key at the place of empty rows.

Modern GPU Modern GPU [24] offers a segmented reduction operation that works on CSR-defined segments, where a segment is defined by its start index in the value array and the start index of the next segment. This allows the segmented reduction operation to work directly with matrices defined in CSR. Furthermore, no extra steps are necessary to deal with empty rows since empty segments can be defined easily.

The only downside is that the project is no longer maintained and that the latest code-base is not compatible with Visual Studio 2017 on Windows.

A comparison of the two implementations shows that they perform similarly, but Modern GPU is much more memory-efficient since it does not require an additional key array. This saves $\text{nnz}(A) \cdot \text{sizeof(float)}$ bytes of memory per matrix A. Though, the modern GPU implementations is not used at the moment since it is no longer maintained.

Accelerating the pre/post matrix multiplication

The post-multiplication routine can be adopted from the CPU implementation but the pre-multiplication requires some more effort.

Post-multiplication For the post-multiplication, the column index of each element is used to determine the scaling element in the diagonal matrix.

```
...
    int column = col_ind[i];
    CSR_value[i] *= post_matrix[column];
...
```

The instructions listed above can be performed by many threads concurrently and independently. The memory read and write access to the array `CSR_value` is fully coalesced, *i.e.* all memory accesses can be combined into a larger transaction. The read access to `post_matrix` however can only be partly coalesced, but this does not impact performance too much since it is constant and thus can be cached.

Pre-multiplication The pre-multiplication seems to be conceptually easier to implement, since it multiplies each element in a row with the same scaling value.

```
...
    CSR_value[i] *= pre_matrix[row];
...
```

The difficulty lies in the fact that it is not obvious how to determine the row index from the index `i` of the value array. One solution is to compute the row index of the matrix from the row pointer in advance, but there is a more elegant solution that does not require any extra memory. The idea is to calculate the row index from the row pointer on the fly. This can be done with a modified binary search as described in Algorithm 4. The goal of the algorithm is to find the index `k` of the segment/row containing the index `i` that is used to index the value array.

$$\texttt{row\_ptr[k]} < \texttt{i} < \texttt{row\_ptr[k+1]} \tag{4.5}$$

Algorithm 4 Binary search

given `m`, `row_ptr`, `i`

1: **initialize** `l = 0`, `u = m`, `row` $= \left\lfloor \frac{\texttt{l + u}}{2} \right\rfloor$
2: **while** `u - l != 1` **do**
3: **if** `i >= row_ptr[row]` **then**
4: `l = row` ▷ Update lower bound
5: **else**
6: `u = row` ▷ Update upper bound
7: **end if**
8: `row` $= \left\lfloor \frac{\texttt{l + u}}{2} \right\rfloor$ ▷ Choose mid point of new bounds
9: **end while**
10: **return** `row`

The idea is to to start with an interval $[\texttt{l}, \texttt{u})$ that contains `k` and then half the interval in each iteration until the interval only contains `k`. The algorithm starts with interval $[\texttt{0}, \texttt{m})$ which is guaranteed to contain `k`.

Assuming $\texttt{k} \in [\texttt{l}, \texttt{u})$, then from condition (4.5) and the fact that `row_ptr` is monotonically increasing, it follows that

$$\texttt{row\_ptr[l]} <= \texttt{i} < \texttt{row\_ptr[u]}. \tag{4.6}$$

In each iteration of Algorithm 4, the index `i` is compared against the value of the row pointer in the middle of the current interval. If `i` is larger or equal, the lower bound is updated to the midpoint. Otherwise, the upper bound is updated to the midpoint. This ensures that condition (4.6) is also true for the reduced interval. The size of the interval is halved in each iteration, thus fulfilling the condition `u - l = 1` in at most $\lceil \log_2(\mathtt{m}) \rceil$ steps.

4.3.3 Choice of Parameters

This section discusses the choice of several ADMM/OSQP related parameters with an indirect linear system solver in mind.

Choice of ρ

The choice of the step-size ρ/R has a large influence on the performance of ADMM. By default, OSQP chooses a larger step-size for the dual variables associated with equality than with inequality constraints (See Section 3.2.2). However, numerical testing with the PCG method as an indirect solver suggests that the convergence of the PCG can be improved by using the same step-size ρ for all constraints, *i.e.* replacing the diagonal matrix R with a scalar ρ.

This observation can further be motivated by looking at the effect of R on reduced KKT matrix (3.11). The diagonal matrix R appears in the term $A^T R A$, where it has the effect of scaling the rows of A by different amounts. This effectively increases the condition number of the linear system and slows down the convergence of PCG.

Adapting ρ

OSQP has demonstrated that adapting the step-size ρ can substantially increase the rate of convergence of ADMM. In OSQP, updating ρ is a trade-off between the convergence speed-up and the re-factorization cost. This constraint is inherent to all direct solution procedures of the KKT system (3.10). However, this does not apply for indirect solution methods, such as the PCG method, where updating ρ amounts to a simple update of a vector.

Thus, ρ can be updated more frequently with virtually no additional cost when using the PCG solver. The only costs are copying the new value of ρ to the GPU and updating the Jacobi preconditioner. Both of these operations take much less time than a single PCG iteration.

Numerical benchmarks suggest that the convergence of ADMM benefits by updating ρ as often as possible. However, they also show that updating ρ more often than every 5 iterations can lead to much worse convergence rates for some problem instances. The following parameters have been found to work well across the benchmark set (See Appendix A).

`rho_eq_over_rho_ineq`	1
`check_termination`	5
`adaptive_rho_interval`	10
`adaptive_rho_tolerance`	1

Table 4.1: OSPQ parameters that are different from their default values for the indirect solver.

Chapter 5

Numerical Results

5.1 The Benchmark

We use a benchmark to compare the performance of the GPU-accelerated OSQP variant (OSQP-GPU) with nominal OSQP. In a first step we compare the performance of OSQP to the indirect PCG solver on the GPU. We refer to this case as OSQP-GPU-PCG. In a second step we incorporate the GPU implementation of the matrix equilibration and residual evaluation into OSQP-GPU-PCG and compare it to OSQP. We call this final version OSQP-GPU. Finally we put the performance of OSQP-GPU into perspective by also comparing it to OSQP's multi-core performance by using the Pardiso solver from Intel Math Kernel Library (MKL). The OSQP solver's single core performance in the benchmark (*i.e.* using QDLDL as the linear system solver) is used as a reference value throughout this chapter if not stated otherwise.

We also evaluate the performance of the matrix equilibration and the residual evaluation on the GPU. Furthermore, we also look at the influence of the floating-point precision on the performance.

All benchmarks are performed with the default parameters of the respective solver. In the case of OSQP, we use the parameters proposed in [27]. For the OSQP-GPU solver we use the values given in Table 4.1. Furthermore, the default absolute and relative tolerances levels of OSQP (*i.e.* $\varepsilon = 10^{-3}$) are used for all solver variants.

All the numerical results were computed on a Linux-based system with an i9-9900K @ 5Ghz (8 cores) processor and 64 GB of DDR4 3200Mhz RAM. As a GPU the RTX 2080Ti with 11 GB of VRAM was used.

5.1.1 Benchmark Problems

The benchmark includes 7 different problem classes which range from random QPs to applications in optimal control, linear regression, and machine learning. For each problem class, 10 different instances for 15 problem dimensions are generated giving a total of 1050 instances. Each problem instance is run 5 times and the median runtime is then reported. The generation of problem instances is described in Appendix A. The problem dimension n ranges from 10^2 to 10^5 and m ranges from 10^3 to 10^6. The number of non-zeros N ranges from 10^4 to 10^9.

5.1.2 Evaluation Criteria

As a performance metric, the benchmark uses the total runtime reported by OSQP. This time includes the setup time of the solver, the time used for scaling, the update time of ρ, and the solve time. The results in this chapter all show average runtime, *i.e.* the average runtime across the 10 different problem instances of the same size. As a reminder, the problem size is defined as

$$N = \text{nnz}(A) + \text{nnz}(P).$$

5.2 Results

5.2.1 Replacing the Linear System Solver

In a first step we have replaced the direct solver of OSQP with the indirect PCG solver implemented on the GPU. Figure 5.1 compares the total computation times for OSQP and OSQP-GPU-PCG for the problem classes Lasso and Huber. It clearly shows that QDLDL is superior over the CG linear system solver for small problem sizes. However, for large problem instances the GPU solver is significantly faster. Furthermore, the slope of the runtime vs problem size of OSQP is approximately constant, whereas the slope of the GPU solver is flatter at the beginning and starts to increase towards larger problems. This behaviour is expected since smaller problems cannot fully utilize the GPU and data transfer latency is predominant.

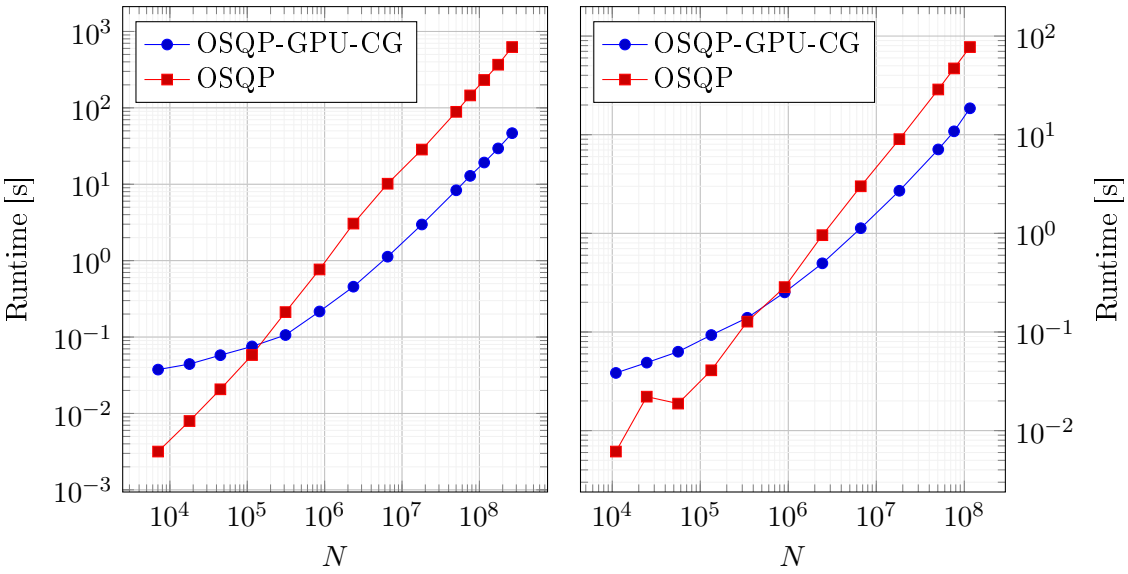

Figure 5.1: Average runtime vs the problem size N for OSQP and OSQP-GPU-PCG for problem class: (left) Lasso, (right) Huber

This can also be seen from Figure 5.2 which shows the achieved speedup in runtime vs problem size of the GPU solver over OSQP for all problem classes. The general trend is that the larger the problem size the larger the speedup. Already 5 out of 7 problem classes achieve a speedup of more than one order of magnitude for at least one problem size.

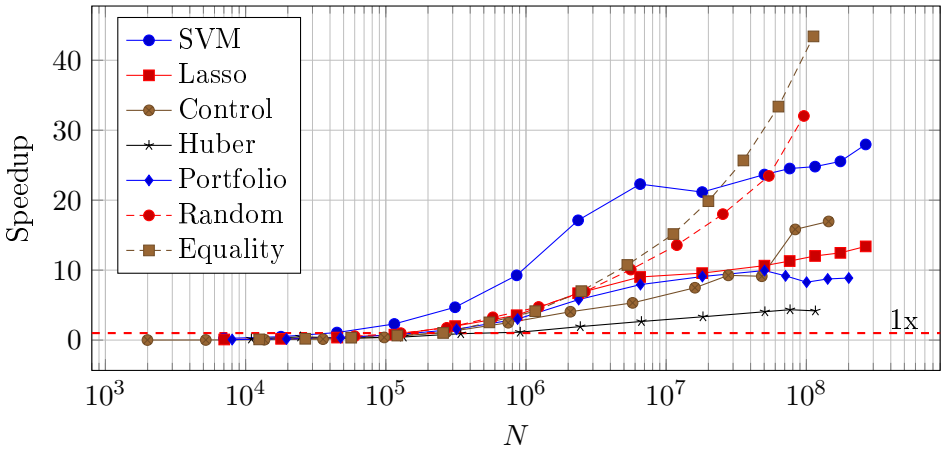

Figure 5.2: Achieved total runtime speedup by replacing the direct solver with the PCG method on the GPU.

5.2.2 Performance of the Residual Evaluation on GPU

Figure 5.3 (left) shows the average computation time of the residuals including the three SpMVs Px, Ax, and A^Ty and the data transfer from and to the GPU for the Lasso class. The computation time on the GPU is almost constant for small problem sizes showing the constant communication overhead. The computation times are very similar across all problem classes, hence only one is shown.

Figure 5.3 (right) shows a clear trend for the speedup, the larger the problem the larger the speedup. This trend continues until the very large problems, where the speedup starts to level off at approximately 44x.

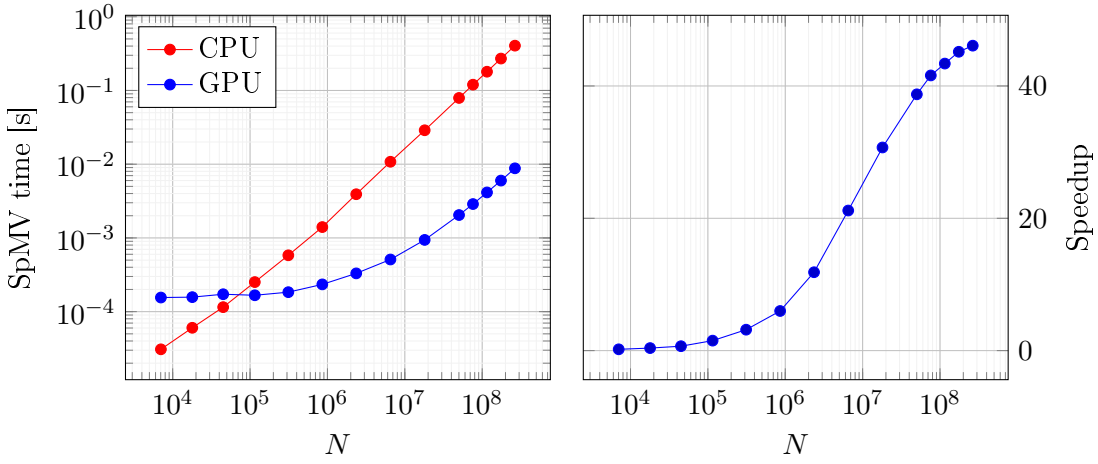

Figure 5.3: (left) Comparison of the average time for checking termination criteria on the CPU or GPU vs problem size N, (right) achieved speedup on the GPU.

5.2.3 Performance of the GPU Matrix Equilibration

Figure 5.3 (left) compares the computation times of the GPU matrix equilibration implementation described in Section 4.3 and the OSQP implementation. The computation time is proportional to the problem size for the OSQP implementation.

Figure 5.4 (right) shows a very similar trend than Figure 5.3 (right) for the residual evaluation. There is very little speedup for small problems followed by a steep increase. However, the speedup starts to level off earlier, at around 20 million non-zeros with a speedup of 45x.

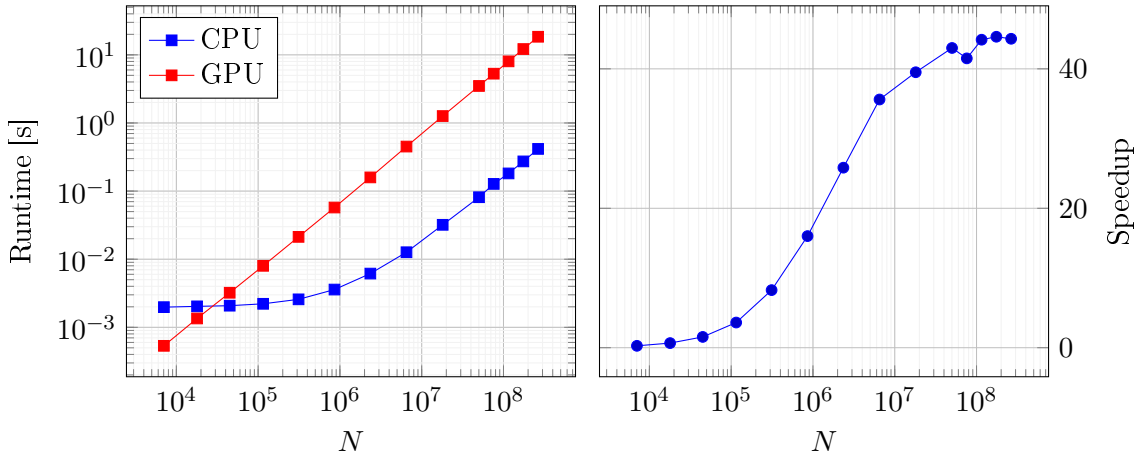

Figure 5.4: (left) Comparison of the matrix equilibration runtime as a function of the problem size N for the CPU and the GPU implementation, (right) achieved speedup on the GPU compared to the CPU version.

5.2.4 Total Runtime with Scaling and Residuals on the GPU

In Section 5.2.1 we have presented the obtained speedups of the PCG-solver on the GPU compared to the direct solver in OSQP. In this section we present the achieved speedup of the GPU solver that includes the GPU implementation of the matrix equilibration and residual calculation (*i.e.* OSQP-GPU).

Figures 5.5–5.11 compare the average runtime vs problem size for each problem class between OSQP and OSQP-GPU. The absolute runtimes are shown on the left subplot and the achieved speedups on the right.

The maximum speedups for each problem class range from 13x up to 160x. The largest reduction in absolute runtime is achieved for the SVM class shown in Figure 5.5. For the largest problem instance OSQP-GPU reduces the runtime from 14 min 30 sec down to mere 8.6 sec. The second largest reduction in runtime is achieved for the Random class with a reduction from 4 min 45 sec down to 1.85 sec shown in Figure 5.10.

The hardest problem class for OSQP-GPU to accelerate is the Huber fitting with the maximum speedup of 13x shown in Figure 5.7. One of the reasons is that OSQP achieves relatively short runtimes compared to other problem classes of the same size. Moreover, the Huber problem class does not benefit from updating ρ more often. In fact, the shortest runtimes are achieved by never updating ρ.

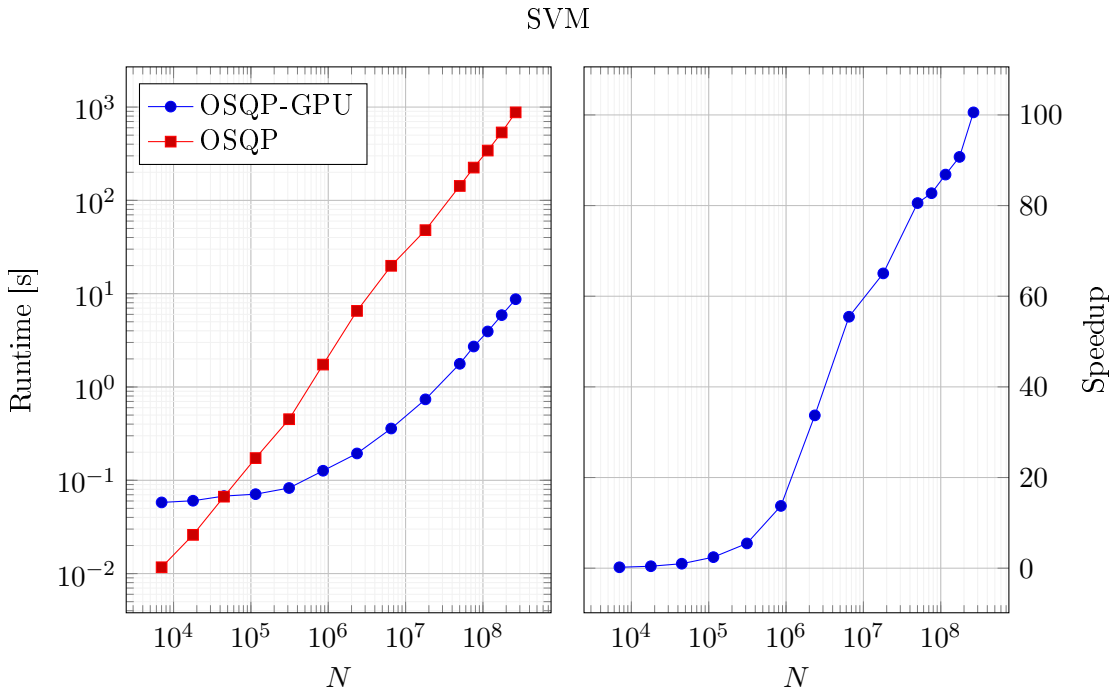

Figure 5.5: Problem class: SVM (left) Comparison of the average runtime of OSQP and OSQP-GPU per problem size N. (right) Achieved speedup in total runtime on the GPU vs the CPU.

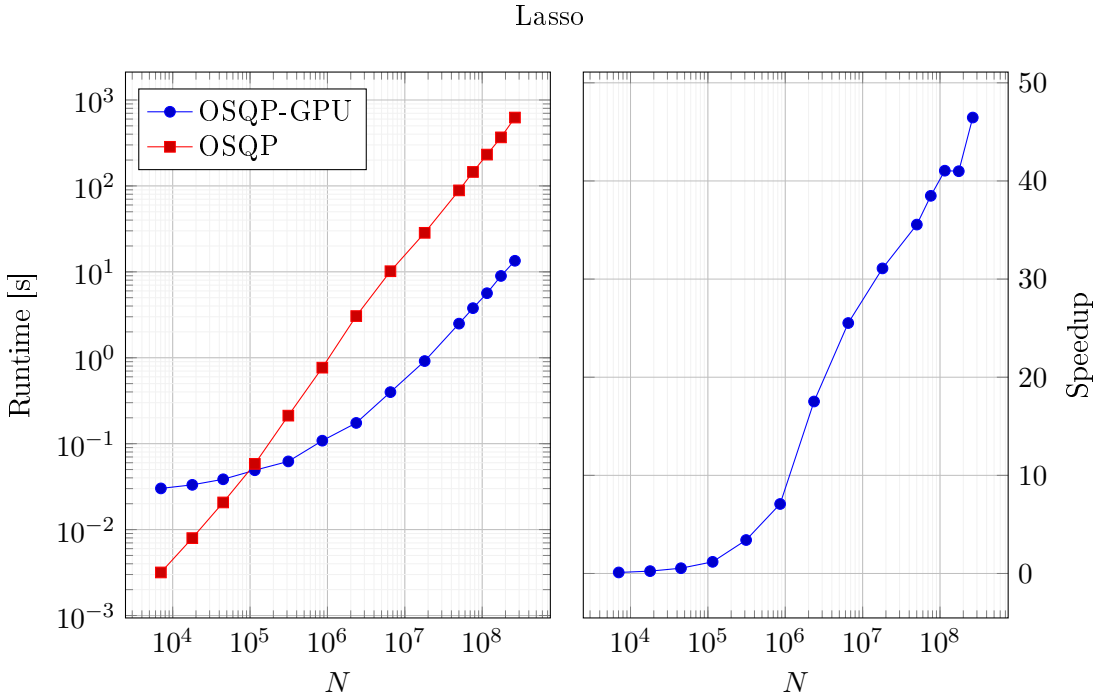

Figure 5.6: Problem class: Lasso (left) Comparison of the average runtime of OSQP and OSQP-GPU per problem size N. (right) Achieved speedup in total runtime on the GPU vs the CPU.

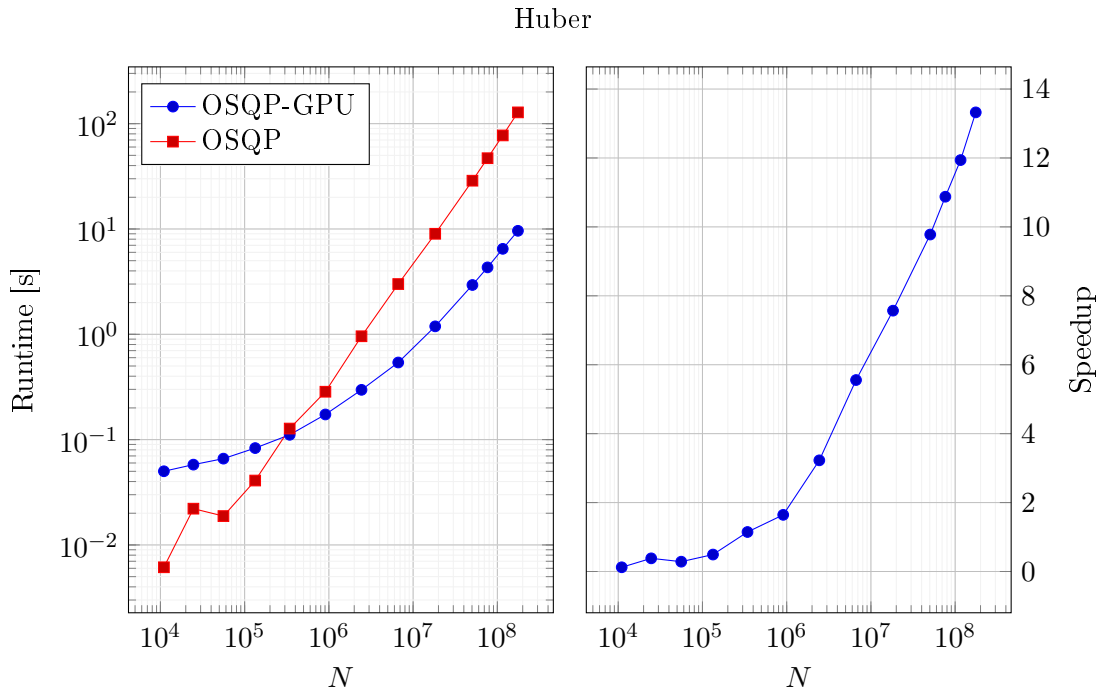

Figure 5.7: Problem class: Huber (left) Comparison of the average runtime of OSQP and OSQP-GPU per problem size N. (right) Achieved speedup in total runtime on the GPU vs the CPU.

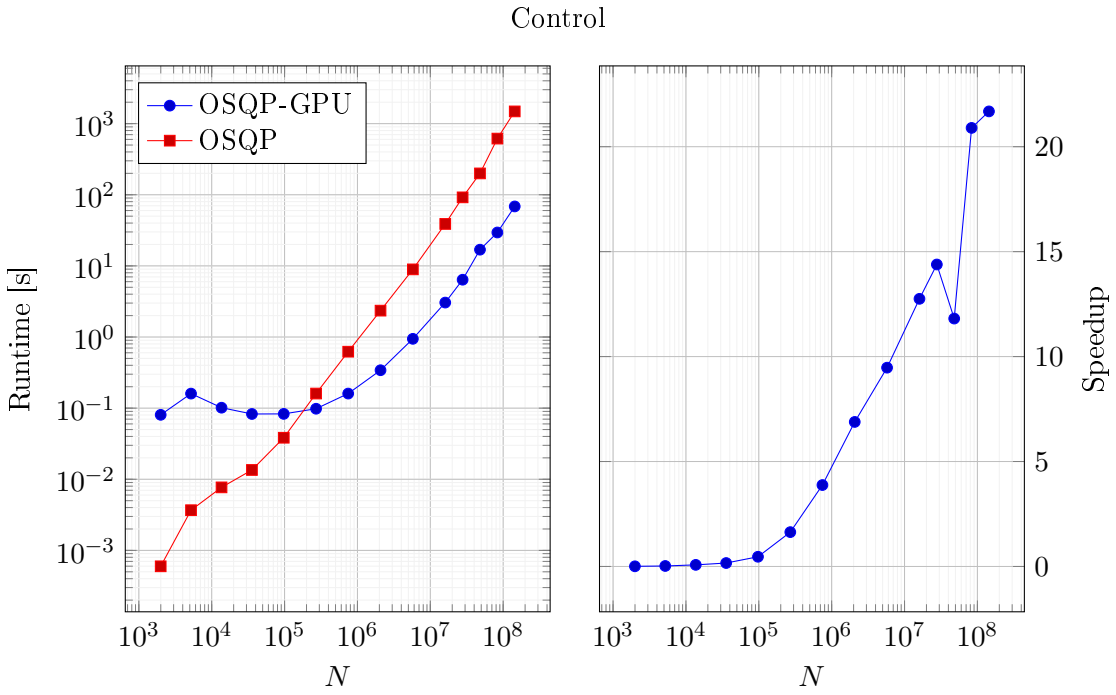

Figure 5.8: Problem class: Control (left) Comparison of the average runtime of OSQP and OSQP-GPU per problem size N. (right) Achieved speedup in total runtime on the GPU vs the CPU.

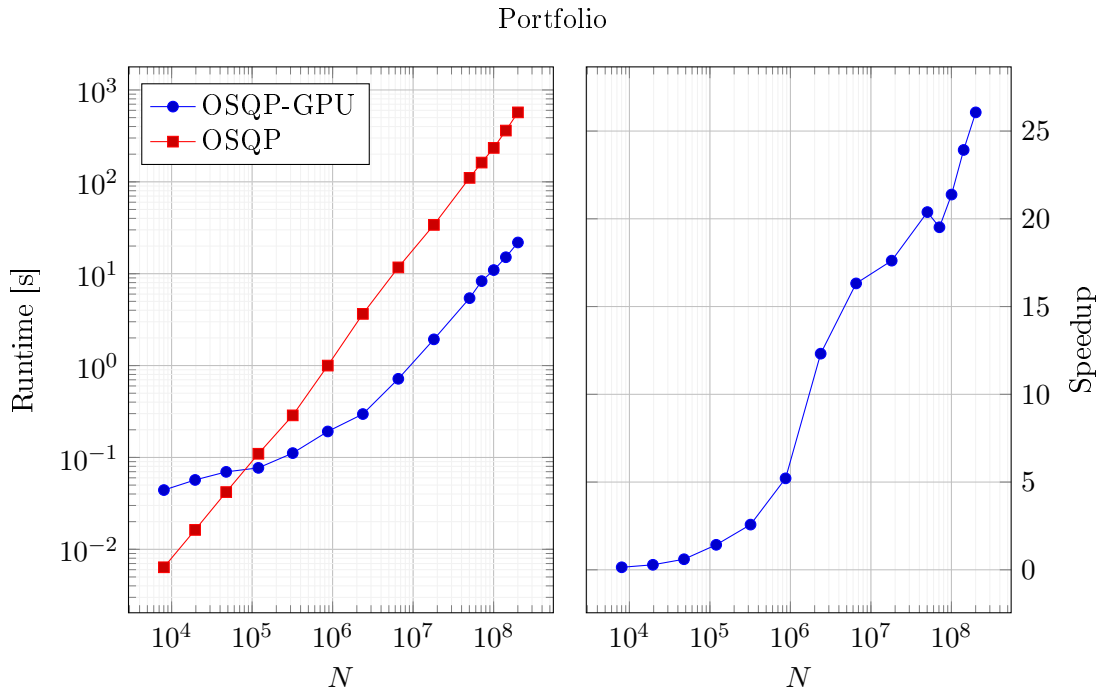

Figure 5.9: Problem class: Portfolio (left) Comparison of the average runtime of OSQP and OSQP-GPU per problem size N. (right) Achieved speedup in total runtime on the GPU vs the CPU.

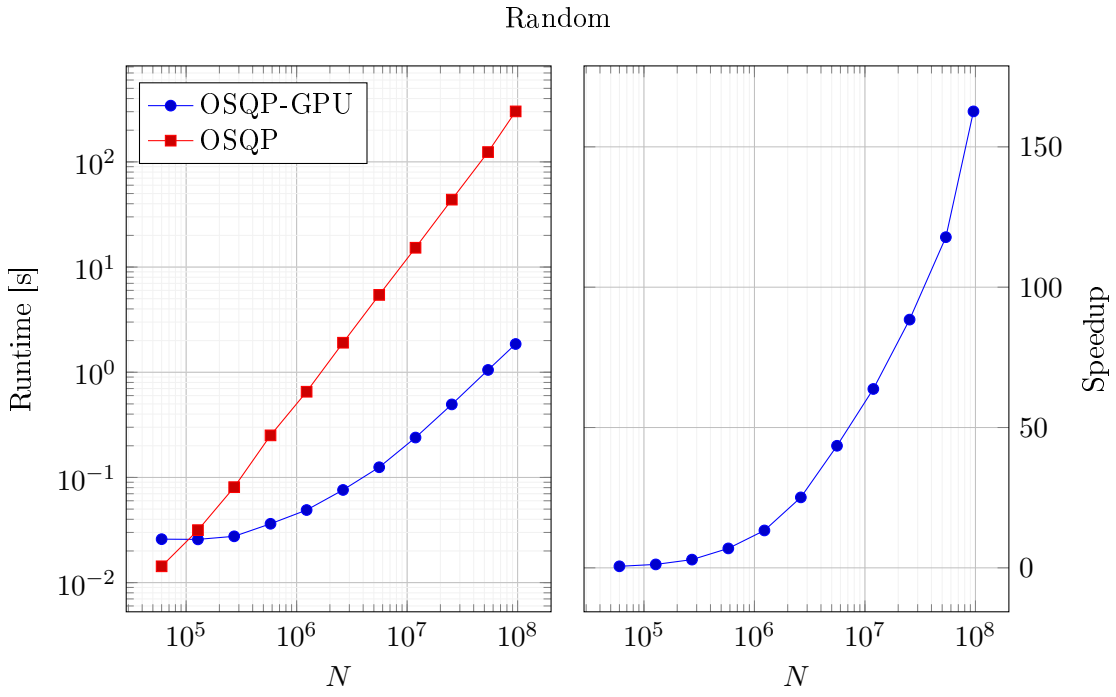

Figure 5.10: Problem class: Random (left) Comparison of the average runtime of OSQP and OSQP-GPU per problem size N. (right) Achieved speedup in total runtime on the GPU vs the CPU.

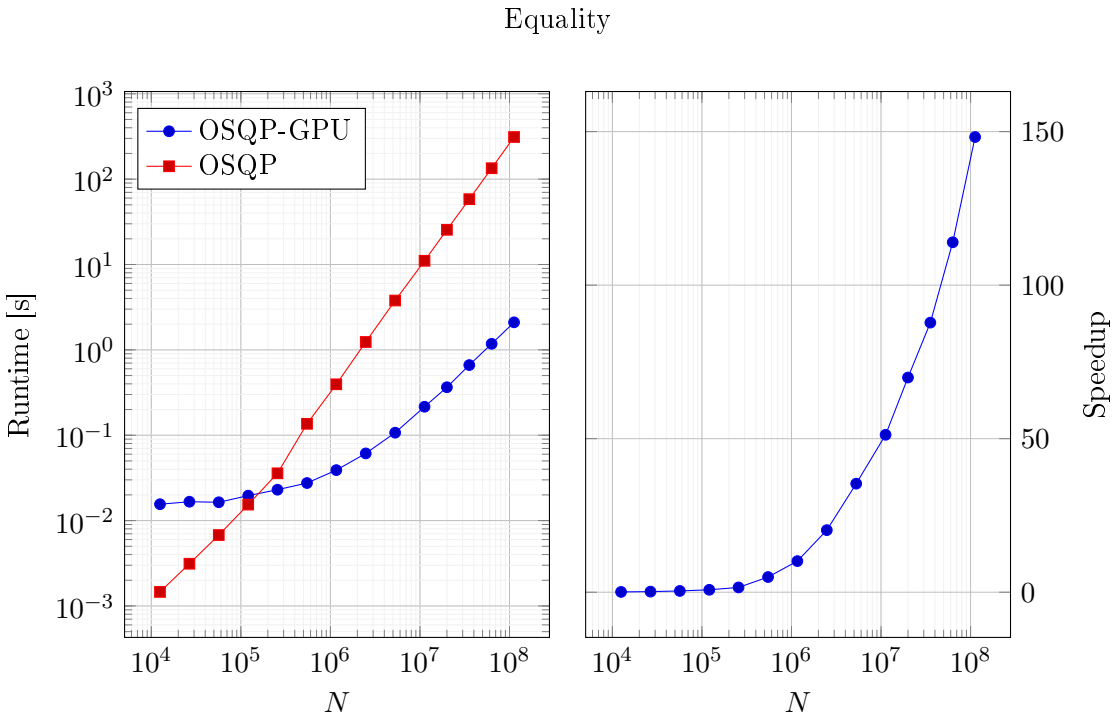

Figure 5.11: Problem class: Equality (left) Comparison of the average runtime of OSQP and OSQP-GPU per problem size N. (right) Achieved speedup in total runtime on the GPU vs the CPU.

5.2.5 Parallelizing OSQP with MKL

This section presents the results of parallelizing OSQP by using the parallel direct solver Pardiso from MKL. Figure 5.12 shows that OSQP with the Pardiso solver is actually slower than the single-threaded OSQP for most problems of the Huber class. Only for the very large problem instances it manages to achieve a speedup larger than 1x. OSQP-GPU is clearly much faster for most of the problem sizes. For the Portfolio class shown in Figure 5.13 though, the Pardiso solver is faster than OSQP for most problem sizes and achieves a maximal speedup of 4.5x. However, there is still a gap of almost 6x between the OSQP-GPU and OSQP-MKL.

The two problem classes, Huber and Portfolio are chosen to present the worst and best performance of OSQP-MKL respectively. The runtime speedups with respect to OSQP of the rest of the problem classes lies between the two extrema presented above.

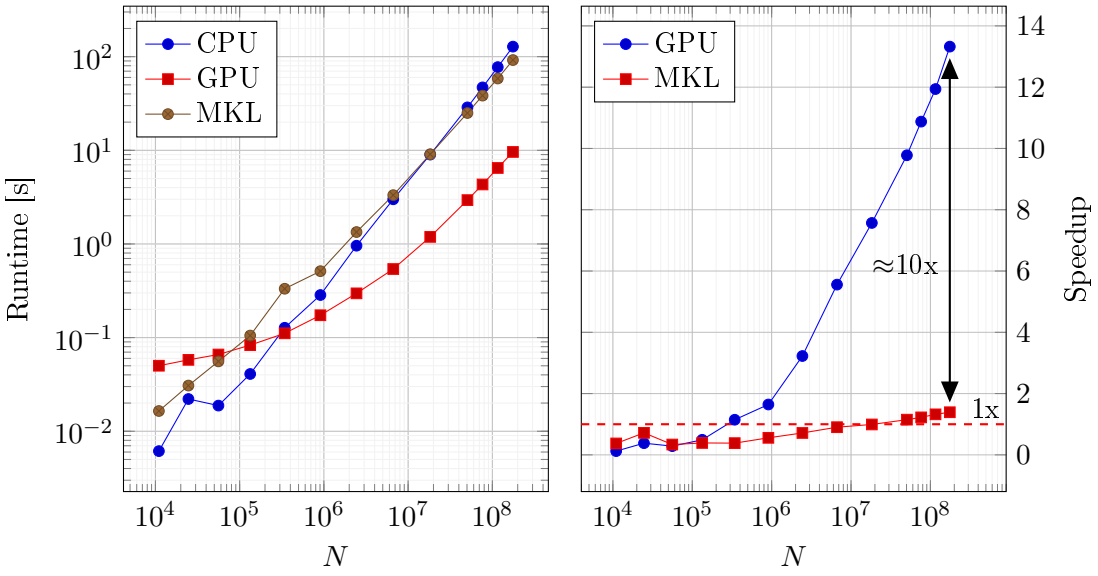

Figure 5.12: Problem class: Huber (left) Comparison of the average runtime of OSQP, MKL and OSQP-GPU per problem size N. (right) Achieved speedup in total runtime of MKL and GPU vs OSQP singe threaded.

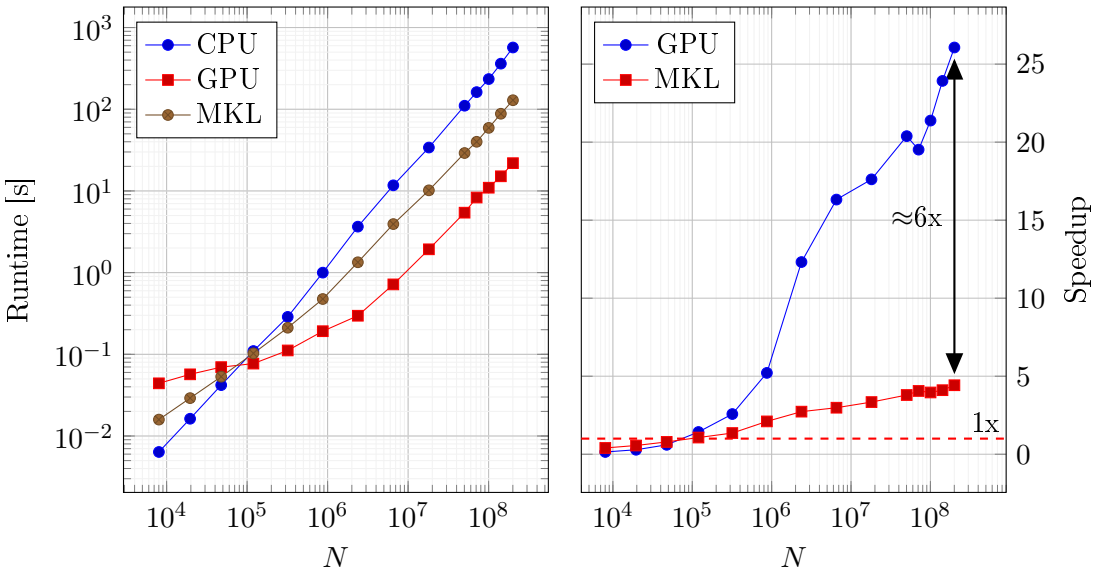

Figure 5.13: Problem class: Portfolio (left) Comparison of the average runtime of OSQP, MKL and OSQP-GPU per problem size N. (right) Achieved speedup in total runtime of MKL and GPU vs OSQP singe threaded.

5.2.6 Single vs Double Precision

Figure 5.14 shows the average runtime of the SVM class vs the problem size for single- and double-precision floating-point values. The penalty in runtime of using double over single precision is less than 1.6x over all problem sizes. This is counter-intuitive at first, since the GPU used

in the benchmark has a 32x higher single precision floating point performance than in double precision. However, most of the GPU routines, especially the SpMV operation, is a memory bound operation. Thus, computation times are limited by the memory bandwidth.

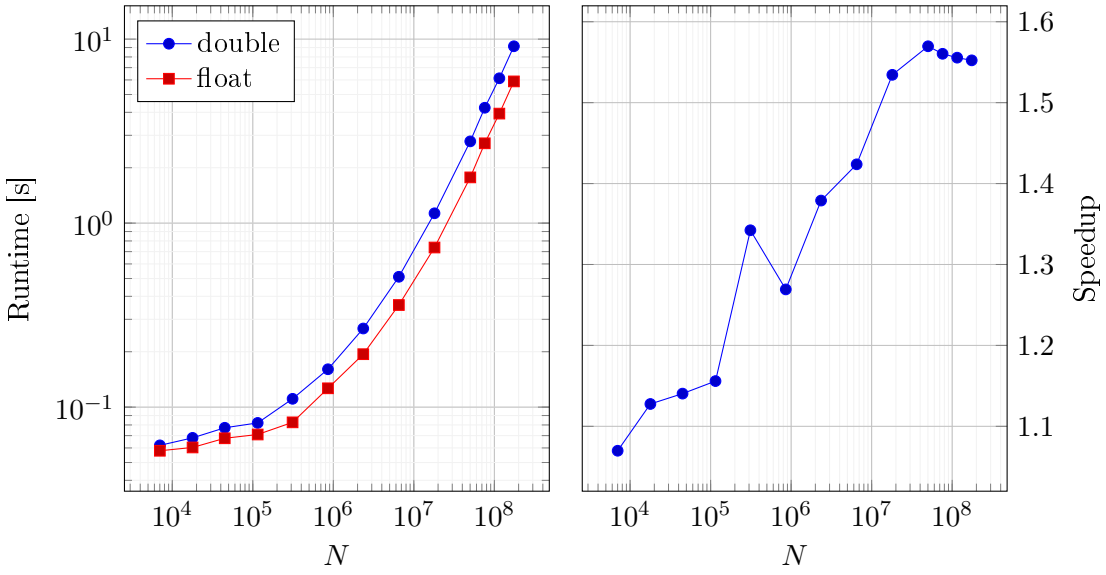

Figure 5.14: Problem class: SVM. (left) Comparison of the average runtime vs problem size N for single and double precision. (right) Achieved speedup in total runtime by using single instead of double precision.

5.2.7 Profiling

Figure 5.15 compares the fraction of the solve time that runs on the GPU for 7 different problem classes. It shows that roughly 75% - 95% of the time is spent on the GPU during the solve phase. This means that implementing the rest of the CPU routines on the GPU would yield a speed up of at most 5 % to 30 % by Amdahl's Law [15]

$$S_N = \frac{1}{p + (1-p)/N},$$

where p is the fraction of the time spent on the serial part of the program, N is the number of parallel processors, and S_N is the speedup. An upper limit of the speedup is given by the inverse of the serial fraction p. Comparing the total time with the time of the solve phase shows a similar picture. Figure 5.16 illustrates this further, more than 90 % of the total runtime is spent during the solve phase. Except for the two problem classes Random and Equality, where the setup phase takes longer due to the rather dense structure of P and A.

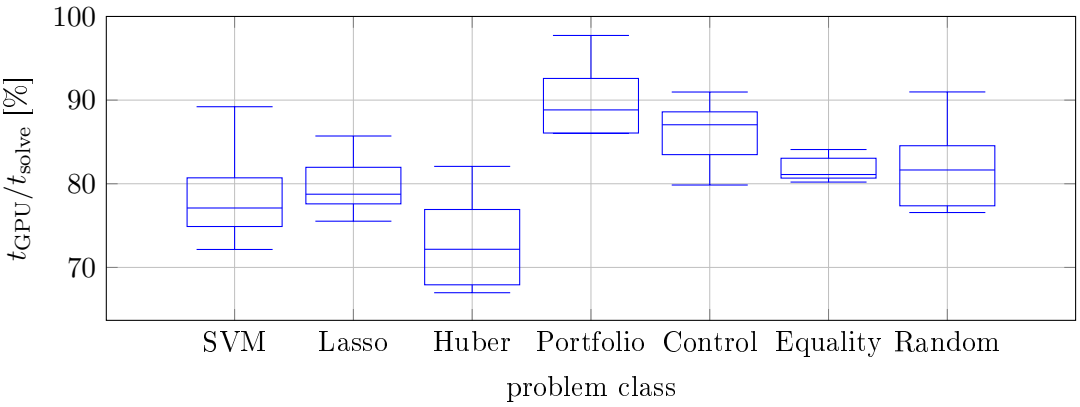

Figure 5.15: Distribution of the relative time spent on the GPU with respect to the total solve time for different problem classes. The distribution for each class is generated from problem instances with N ranging from 10^5 to 10^7.

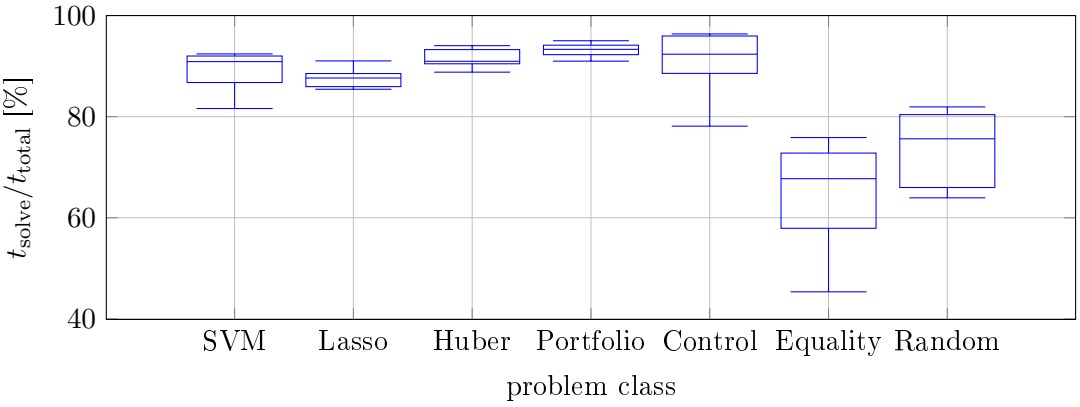

Figure 5.16: Distribution of the solve time compared to total runtime for different problem classes. The distribution for each class is generated from problem instances with N ranging from 10^5 to 10^7.

Chapter 6

Conclusion

We have demonstrated the huge potential of GPUs for solving large-scale QPs with hundreds of millions non-zeros in less than 10 seconds. First, we showed that replacing a direct linear system solver of OSQP with a PCG method implemented on the GPU can achieve a speedup of one order of magnitude in total runtime. Moreover, we investigated other ways to parallelize OSQP to reduce the runtime even further by replacing serial CPU routines with parallel GPU implementations.

Combined with the tweaked parameters of the OSQP algorithm, we were able to reduce the runtime by up to two orders of magnitude. We also established that GPUs are not suited for small-scale problems for which CPU implementations are generally much faster.

We have implemented the PCG solver on the GPU, written in CUDA C, that is interfaced with the open-source solver OSQP. Furthermore, we have also implemented the matrix equilibration and residual calculation of OSQP in CUDA C on the GPU. The complete implementation of the OSQP-GPU solver is cross-platform, and has been tested on both Linux and Windows.

6.1 Future Work

There are several ways to improve upon our results.

An obvious next step is to extend the implementation of the PCG to a multi-GPU setup. This opens up the potential to solve problems beyond one billion non-zero elements. There are many things to consider, such as how to split up the work-load across multiple GPUs, how to ensure synchronization, etc. Other factors to investigate is the communication and latency between the GPUs.

It would be interesting to investigate the merits of a concept called unified memory space, which merges the system memory with the GPU memory and automatically transfers data on demand between the two memory spaces. This increases the available memory to the amount of system memory. Furthermore, a compariosn to the multi-GPU setup could be made.

We assumed that the matrix equilibration performed by OSQP for the KKT system is also a good heuristic for the reduced KKT system solved by the indirect solver. It would be interesting to investigate other heuristic approaches to scale the problem data such that it is a good preconditioner for the reduced system.

Using a different iterative method instead of PCG, such as the minimum residual (MINRES)

method, could make use of the KKT system directly. This would eliminate the need to form the reduced system and would open up possibilities to use different preconditioners, such as the incomplete Cholesky or incomplete LU preconditioner.

Appendix A

Benchmark Problems

This chapter describes the problem classes used in the OSQP benchmark. The formulations of the problem instances are adopted from [27], if not stated otherwise. To distinguish between the dimensions of the original problem and the reformulated one, we introduce the new variables $\tilde{n}$ and $\tilde{m}$ to refer to dimensions of a QP in the form (3.1).

If not stated otherwise, n is the parameter that is adapted to change the dimension and size of the problem instance.

A.1 Random QP

The Random QP problem class is of the form

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}x^T P x + q^T x \\ \text{subject to} & l \leq Ax \leq u. \end{array}$$

Problem structure The number of variables is n and the number of constraints is fixed as $m = 10n$. P is generated as $P = M^T M$, where $M \in \mathbb{R}^{n \times n}$ with 50% non-zero elements $m_{ij} \sim \mathcal{N}(0, 1)$. A is generated similarly with 50 % non-zero elements $a_{ij} \sim \mathcal{N}(0, 1)$, q is also normally distributed, *i.e.* $q_i \sim \mathcal{N}(0, 1)$. The upper and lower bounds are generated from a uniform distribution as $l_i \sim \mathcal{U}(-1, 0)$ and $u_i \sim \mathcal{U}(0, 1)$

The resulting QP has the following dimensions

$$\begin{aligned} \tilde{n} &= n \\ \tilde{m} &= m \end{aligned}$$

A.2 Equality Constrained QP

Equality constrained QP problem class is of the form

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}x^T P x + q^T x \\ \text{subject to} & Ax = b. \end{array}$$

Problem structure The number of variables is n and the number of constraints is fixed as $m = \lfloor n/2 \rfloor$. P is generated as $P = M^T M$, where $M \in \mathbb{R}^{n \times n}$ with 50% non-zero elements $m_{ij} \sim \mathcal{N}(0, 1)$. A is generated similarly with 50 % non-zero elements $a_{ij} \sim \mathcal{N}(0, 1)$. The vectors q and b are normally distributed, *i.e.* $q_i, b_i \sim \mathcal{N}(0, 1)$.

The resulting QP has the following dimensions

$$\begin{aligned} \tilde{n} &= n \\ \tilde{m} &= m \end{aligned}$$

A.3 Optimal Control

The optimal control problem class is of the form

$$\begin{aligned} \text{minimize} \quad & x_T^T P x_T + \sum_{t=0}^{T-1} x_t^T Q x_t + u_t^T R u_t \\ \text{subject to} \quad & x_0 = x_{\text{init}} \\ & x_{t+1} = A x_t + B u_t \\ & -\bar{x} \leq x_t \leq \bar{x} \\ & -\bar{u} \leq u_t \leq \bar{u}, \end{aligned}$$

where $x_t \in \mathbb{R}^{n_x}$ and $u_t \in \mathbb{R}^{n_u}$ are the state and input to the system at time t, which are constrained to boxes bounded by $\bar{x} \in \mathbb{R}^{n_x}$ and $\bar{u} \in \mathbb{R}^{n_u}$. The prediction horizon is T and x_{init} is the initial state.

Problem structure The dimensions of the dynamical system are defined as $n_x = n$ states and $n_u = 0.5n$ inputs and the prediction horizon is set to $T = 10$. The bounds of the boxes are generated as $\bar{x}_i \sim \mathcal{U}(1, 2)$ and $\bar{u}_i \sim \mathcal{U}(0, 0.1)$. The initial state x_{init} is drawn from a uniform distribution, *i.e.* $x_{\text{init}} \sim \mathcal{U}(-0.5\bar{x}, 0.5\bar{x})$. The system dynamics are generated as $A = I + \Delta$ with $\Delta_{ij} \sim \mathcal{N}(0, 0.1)$ and $B_{ij} \sim \mathcal{N}(0, 1)$. Only stable systems are considered, thus the eigenvalues of A are forced to be less than 1.

The state cost $Q = \text{diag}(q)$ is uniformly generated with 70% none-zero elements as $q_i \sim \mathcal{U}(0, 10)$. The input cost R is chosen to be the scaled identity $R = 0.1I$. Finally, the terminal cost P is chosen as the infinite horizon cost of the LQR problem defined by A, B, Q, R.

The resulting QP has the following dimensions

$$\begin{aligned} \tilde{n} &= (T + 1)n_x + T n_u \\ \tilde{m} &= 2(T + 1)n_x + T n_u \end{aligned}$$

A.4 Portfolio Optimization

The Portfolio optimization problem class can be formulated as a QP of the form

$$\begin{aligned} \text{minimize} \quad & x^T D x + y^T y - \tfrac{1}{\gamma} \mu^T x \\ \text{subject to} \quad & y = F^T x \\ & \mathbf{1}^T x = 1 \\ & x \geq 0, \end{aligned}$$

where $x \in \mathbb{R}^n$ represents the choice of assets and $y \in \mathbb{R}^k$ is an auxiliary variable.

Problem structure The number of assets n is determined by the number of factors k as $n = 100k$. The factor loading matrix $F \in \mathbb{R}^{n \times k}$ is generated from a normal distribution, $F_{ij} \sim \mathcal{N}(0,1)$ with 50% non-zero elements. The diagonal matrix $D \in \mathbb{R}^{n \times n}$ is chosen from $d_{ii} \sim \mathcal{U}(0, \sqrt{k})$. The mean return vector $\mu \in \mathbb{R}^n$ is given by $\mu_i \sim \mathcal{N}(0,1)$ and γ is set to 1.

The resulting QP has the following dimensions

$$\begin{aligned} \tilde{n} &= n + k = 101k \\ \tilde{m} &= n + 1 + k = 101k + 1 \end{aligned}$$

Note that the parameter k is used instead of n to adapt the instance size.

A.5 Lasso

The Lasso problem class formulated as a QP has the following form

$$\begin{aligned} \text{minimize} \quad & y^T y + \lambda \mathbf{1}^T t \\ \text{subject to} \quad & y = Ax - b \\ & -t \leq x \leq t, \end{aligned}$$

where $x \in \mathbb{R}^n$ is the vector of parameters and $y \in \mathbb{R}^m$ and $t \in \mathbb{R}^n$ are auxiliary variables.

Problem structure The number of data points is chosen as $m = 100n$, where n is the number of features. The data matrix $A \in \mathbb{R}^{m \times n}$ has 50% non-zero elements and is drawn as $a_{ij} \sim \mathcal{N}(0,1)$. Vector b is constructed from a true vector v and additive noise ε as $b = Av + \varepsilon$. The vector v has 50% non-zero elements and is generated as $v_i \sim \mathcal{N}(0, 1/n)$, and the noise is generated from a normal distribution $\varepsilon_i \sim \mathcal{N}(0,1)$. The relaxation parameter λ is set to $\frac{1}{5}\|A^T b\|_\infty$.

The resulting QP has the following dimensions

$$\begin{aligned} \tilde{n} &= 2n + m \\ \tilde{m} &= 2n + m \end{aligned}$$

A.6 Huber

The Huber fitting problem's QP formulation is adopted from [21, Eq. 24] and is defined as

$$\begin{aligned} \text{minimize} \quad & \tfrac{1}{2} z^T z + \mathbf{1}^T (r + s) \\ \text{subject to} \quad & Ax - b - z = r - s \\ & r, s \geq 0, \end{aligned}$$

where $z \in \mathbb{R}^n$, $s \in \mathbb{R}^m$, and $t \in \mathbb{R}^m$ are auxiliary variables and $x \in \mathbb{R}^n$ is the feature vector.

Problem structure The number of data points is chosen to be $m = 100n$, where n is the number of features. The problem data $A \in \mathbb{R}^{m \times n}$ is generated with 50% non-zero elements as $A_{ij} \sim \mathcal{N}(0,1)$. The vector $b \in \mathbb{R}^m$ is generated from a vector $v \in \mathbb{R}^n$ as $v_i \sim \mathcal{N}(0, 1/n)$ and a noise vector $\varepsilon \in \mathbb{R}^m$ whose elements are defined as

$$\varepsilon \sim \begin{cases} \mathcal{N}(0,1) & \text{with probability } p = 0.95 \\ \mathcal{U}(0,10) & \text{else} \end{cases}$$

Then b is set to $Av + \varepsilon$.

The resulting QP has the following dimensions

$$\begin{aligned} \tilde{n} &= 3m + n \\ \tilde{m} &= 3m \end{aligned}$$

A.7 Support Vector Machine

The Support Vector Machine problem class can be formulated as a QP of the following form

$$\begin{aligned} \text{minimize} \quad & x^T x + \lambda \mathbf{1}^T t \\ \text{subject to} \quad & t \geq \operatorname{diag}(b) A x + \mathbf{1} \\ & t \geq 0, \end{aligned}$$

where $t \in \mathbb{R}^m$ is an auxiliary variable and $x \in \mathbb{R}^n$ is the normal vector of the separating hyperplane.

Problem structure The number of data points is chosen as $m = 100n$, where n is the number of features. The vector b is chosen according to

$$b_i = \begin{cases} +1 & i \leq m/2 \\ -1 & \text{otherwise,} \end{cases}$$

and the problem data $A \in \mathbb{R}^{m \times n}$ with 50% non-zero elements as

$$A_{ij} \sim \begin{cases} \mathcal{N}(1/n, 1/n) & i \leq m/2 \\ \mathcal{N}(-1/n, 1/n) & \text{otherwise,} \end{cases}$$

and $\lambda = 1$.

The resulting QP has the following dimensions

$$\begin{aligned} \tilde{n} &= n + m \\ \tilde{m} &= 2m \end{aligned}$$

Notation

Sets

$\mathbb{N}$	the set of natural numbers
$\mathbb{R}$	the set of real numbers
$\mathbb{R}_{++}$	the set of positive real numbers
$\mathbb{R}^n$	the set of real valued vectors of dimension n
$\mathbb{R}^{m \times n}$	the set of m-by-n real matrices
$\mathbb{S}^n$	the set of n-by-n symmetric matrices
$\mathbb{S}^n_+$	the set of n-by-n symmetric positive semi-definite matrices
$\mathbb{S}^n_{++}$	the set of n-by-n symmetric positive definite matrices
$[a, b)$	the set of integers between a and b excluding b, with $a, b \in \mathbb{N}$

Matrices and Vectors

$\mathbf{1}$	the vector of ones of appropriate dimension
$x_+ (x_-)$	the vector obtained by setting negative (positive) elements of x to zero
I	the identity matrix of appropriate dimension
A^T	the transpose of the matrix A
A_i	the i-th column of the matrix A
$(Ax)_i$	the i-th value of the matrix-vector product Ax
$\mathrm{diag}(x)$	maps a vector to a diagonal matrix
$\mathrm{diag}(A)$	maps the diagonal of a matrix to a vector
$\mathrm{nnz}(A)$	the number of non-zero elements of the sparse matrix A

Norms

$\|\cdot\|$	a vector norm
$\|x\|_\infty$	the infinity-norm of x: $\|x\|_\infty = \max_i(\|x_i\|)$
$\|x\|_2$	the 2-norm of x: $\|x\|_2 = \sqrt{x^T x}$
$\|x\|_K$	the K-norm of x: $\|x\|_K = \sqrt{x^T K x}, \quad K \in \mathbb{S}^n_{++}$

Other notation

$a \leq b$	element-wise inequality between a and b
$\lfloor x \rfloor$	the largest integer less than or equal to $x \in \mathbb{R}$
$\lceil x \rceil$	the smallest integer larger than or equal to $x \in \mathbb{R}$

Acronyms

ADMM	Alternating Direction Method Of Multipliers
API	Application Programming Interface
BLAS	Basic Linear Algebra Subprograms
CG	Conjugate Gradient
COO	Coordinate
CSC	Compressed Sparse Column
CSR	Compressed Sparse Row
DDR	Double Data Rate
FLOPS	Floating Point Operations Per Second
GPU	Graphics Processing Unit
LP	Linear Program
MHE	Moving Horizon Estimation
MINRES	Minimum Residual
MIP	Mixed Integer Programming
MKL	Math Kernel Library
MPC	Model Predictive Control
PCG	Preconditioned Conjugate Gradient
QP	Quadratic Program
SDP	Semi Definite Program
SM	Streaming Multiprocessor
SOCP	Second-order Cone Program
SpMV	Sparse Matrix-vector Multiplication
SQD	Symmetric Quasi-definite
SQP	Sequential Quadratic Programming
STL	Standard Template Library
SVM	Support Vector Machines

Bibliography

[1] F. Allgöwer, T. Badgwell, J. Qin, J. Rawlings, and S. Wright. "Nonlinear predictive control and moving horizon estimation – an introductory overview". In: *Advances in Control*. Springer London, 1999, pp. 391–449.

[2] S. Ashby, T. Manteuffel, and J. Otto. "A comparison of adaptive Chebyshev and least squares polynomial preconditioning for Hermitian positive definite linear systems". In: *SIAM Journal on Scientific and Statistical Computing* 13.1 (1992), pp. 1–29.

[3] G. Banjac, P. Goulart, B. Stellato, and S. Boyd. "Infeasibility detection in the alternating direction method of multipliers for convex optimization". In: *Journal of Optimization Theory and Applications (to appear)* (2019).

[4] N. Bell and M. Garland. *Efficient sparse matrix-vector multiplication on CUDA*. NVIDIA Technical Report NVR-2008-004. NVIDIA Corporation, 2008.

[5] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. "Distributed optimization and statistical learning via the alternating direction method of multipliers". In: *Foundations and Trends in Machine Learning* 3.1 (2011), pp. 1–122.

[6] S. Boyd, E. Busseti, S. Diamond, R. Kahn, P. Koh K. Nystrup, and J. Speth. "Multi-period trading via convex optimization". In: *Foundations and Trends in Optimization* 3.1 (2017), pp. 1–76.

[7] C. Cortes and V. Vapnik. "Support-vector networks". In: *Machine Learning* 20.3 (1995), pp. 273–297.

[8] *cuBLAS Toolkit Documentation*. `https://docs.nvidia.com/cuda/cublas/index.html`. Version 10.1.168.

[9] *CUDA C Programming Guide*. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`. Version 10.1.168.

[10] *cuSPARSE Toolkit Documentation*. `https://docs.nvidia.com/cuda/cusparse/index.html`. Version 10.1.168.

[11] X. Feng, H. Jin, R. Zheng, K. Hu, J. Zeng, and Z. Shao. "Optimization of sparse matrix-vector multiplication with variant CSR on GPUs". In: *IEEE International Conference on Parallel and Distributed Systems*. 2011.

[12] C. García, D. Prett, and M. Morari. "Model predictive control: theory and practice – a survey". In: *Automatica* 25.3 (1989), pp. 335 –348.

[13] A. George, K. Ikramov, and A. Kucherov. "Some properties of symmetric quasi-definite matrices". In: *SIAM Journal on Matrix Analysis and Applications* 21.4 (2000), pp. 1318–1323.

[14] G. H. Golub and C. F. Van Loan. *Matrix Computations*. 4th. Baltimore, MD: The Johns Hopkins University Press, 2013.

[15] M. Hill and M. Marty. "Amdahl's law in the multicore era". In: *Computer* 41.7 (2008), pp. 33–38.

[16] A. Krizhevsky, I. Sutskever, and G. Hinton. "ImageNet classification with deep convolutional neural networks". In: *Communications of the ACM* 60.6 (2017), pp. 84–90.

[17] R. Lämmel. "Google's MapReduce programming model – revisited". In: *Science of Computer Programming* 70.1 (2008), pp. 1–30.

[18] C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, and W. Shi. "Photo-realistic single image super-resolution using a generative adversarial network". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017.

[19] V. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU". In: *International Symposium on Computer Architecture*. 2010.

[20] S. Liu, J. Tang, Z. Zhang, and J. Gaudiot. "Computer architectures for autonomous driving". In: *Computer* 50.8 (2017), pp. 18–25.

[21] O. Mangasarian and D. Musicant. "Robust linear and support vector regression". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22.9 (2000), pp. 950–955.

[22] H. Markowitz. "Portfolio selection". In: *The Journal of Finance* 7.1 (1952), pp. 77–91.

[23] D. Merrill and M. Garland. "Merge-based parallel sparse matrix-vector multiplication". In: *International Conference for High Performance Computing, Networking, Storage and Analysis*. 2016.

[24] *Modern GPU, a CUDA C++ template library*. `https://github.com/moderngpu/moderngpu`. Version 2.11, Accessed 2019.

[25] J. Nocedal and S. Wright. *Numerical Optimization*. Springer Science & Business Media, 2006.

[26] B. O'Donoghue, E. Chu, N. Parikh, and S. Boyd. "Conic optimization via operator splitting and homogeneous self-dual embedding". In: *Journal of Optimization Theory and Applications* 169.3 (2016), pp. 1042–1068.

[27] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd. "OSQP: an operator splitting solver for quadratic programs". In: *arXiv:1711.08013* (2018).

[28] *Thrust Documentation*. `https://docs.nvidia.com/cuda/thrust/index.html`. Version 10.1.168.

[29] R. Tibshirani. "Regression shrinkage and selection via the lasso". In: *Journal of the Royal Statistical Society: Series B* 58.1 (1996), pp. 267–288.

[30] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringarpure. "On the limits of GPU acceleration". In: *USENIX Conference on Hot Topics in Parallelism*. 2010.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

GPU Acceleration of ADMM for Large-Scale Convex Optimization

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Schubiger	Michel Philippe

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 9.8.2019	M. Schubiger

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.