# Project 7: Efficient Convex Optimization on GPUs for Model Predictive Control

### Problem Statement

Model predictive control (MPC) is an advanced control technique for complex multivariable control problems. It is widely used in industries like chemical plants, oil refineries, power system balancing models, power electronics etc. MPC relies on the real-time solution of a convex optimization problem to compute the control input(s) to a system. Comparing with traditional control techniques such as PID, MPC is very computationally demanding. Quadratic programming algorithms for the solution of convex optimization problems generally lend themselves to parallelization. There are several powerful algorithms for solving the QP that occur in MPC problems that need to be solved online at each sampling time. We will focus on two gradient-based algorithms - Parallel Quadratic Programming (PQP) and Alternating Direction Method of Multipliers (ADMM). The performance of these algorithms is dominated by the performance of matrix-matrix and matrix-vector products. There is a huge scope of optimization of these algorithms implemented for GPU. The overall performance can be further optimized by coarsening and fusion. The aim of this project is to design an efficient GPU implementation to solve Quadratic Programming in MPC such that the overall execution time is minimized.

* Implement efficient GPU version of Parallel Quadratic Programming (PQP) for solving QP in MPC using the following optimization techniques by efficient implementation of
  + SGEMM (Single-precision General Matrix-Matrix Multiplication) kernels and SGEMV (Single-precision General Matrix-Vector Multiplication).(3 students)
  + Thread level and/or block-level coarsening across x and y dimension for better utilization of GPU (3 students)
  + Using adaptive kernel fusion with proper synchronization in such a way that the data movement and launching overhead is significantly reduced (3 students)
* Write a script to generate the fused version of the code given the kernels you need to fuse and the type of fusion and the launching parameters of the unfused kernels.
* Write a script to generate the coarsened version of the code given the kernels you need to coarsen, type of coarsening, the coarsening factor and the launching parameters of the un-coarsened kernel.
* Evaluate the runtime of the optimized PQP algorithm on a real-world MPC example (to be provided). For the example plant, you need to vary the MPC prediction horizon and check the difference in running time between non-optimized and optimized parallel implementations.

### Introduction

Convex optimization is a set of mathematical techniques implemented to solve convex functions over convex sets. A subgroup of this is, Quadratic Programming (QP), which is widely used for solving MPC type problems. QP algorithms are highly suitable when it comes to parallelization on GPUs and can accelerate such high computationally demanding jobs.

The proposed method intends to design an efficient GPU implementation to solve Quadratic

Programming in MPC such that the overall execution time is minimized. Quadratic programming

algorithms that solve convex optimization problems usually lend themselves to parallelization and provides a linear convergence rate. In our implementation, we have focused on the gradient-based algorithm – parallel quadratic programming whose performance is dominated by the performance of matrix-matrix and matrix-vector products. But it has a huge scope of optimization to implement efficient algorithms for GPU. We make GPU utilization in a better way by using coarsening across x and y block dimensions of launching kernel on unoptimized code. Then adaptive kernel fusion is used on the implemented code to proper thread synchronization. So that data movement and overhead launching are reduced considerably and overall performance is optimized significantly.

The application of PQP to MPC problem would require at every step the formulation of the dual QP, its solution, and the generation of the primal solution. However, the structure of the QP problem of MPC can be exploited to perform part of the calculations ofﬂine and to synthesize a controller embedding the optimization algorithm. The algorithm for this optimized solution as proposed by Di Cairano, S. and Brand, M., in 2013 in the research paper “On a multiplicative update dual optimization algorithm for constrained linear MPC. In 52nd IEEE Conference on Decision and Control (pp. 1696-1701). IEEE” is shown in Fig 1.

**Notation**: R, R0+, R+, Z, Z0+, Z+ denote real, nonnegative real, positive real, integer, nonnegative integer, and positive integer numbers, respectively, and Z[a,b] , {z ∈ Z : a ≤ z ≤ b}. For a vector φ ∈ Rn, [φ]i denotes the ith component, for a matrix Φ ∈ Rn×m, [Φ]ij denotes the element at the ith row and jth column. We denote a vector of size m entirely composed of ones by 1m, the identity matrix of size m by Im, and the matrix entirely composed of zeros by 0m, where subscripts are dropped when clear from the context. For vectors, absolute value, maximum, and inequalities are intended componentwise, while for a symmetric matrix Q, Q > 0 (Q ≥ 0) indicates positive (semi)deﬁniteness. For a vector x and a matrix Q ≥ 0, ||x||2Q= x′Qx. For a signal a sampled with period Ts, ak is the value at the kth sampling instant, i.e., at time kTs, and ai|k denotes the predicted value of a at step k + i, based on data at step k. Given Φ ∈ Rn×m, we deﬁne Φ+,Φ− ∈ Rn×m where [Φ+]ij = max(0,[Φ]ij) and [Φ−]ij = max(0,−[Φ]ij). For the optimization problem minz∈Z J(z), the optimum is J∗ and the optimal solution is z∗, i.e., J∗ = J(z∗).

Linear MPC is based on the prediction model:

xk+1 = Axk + Buk  and yk = Cxk + Duk

where x ∈ Rn, u ∈ Rm, y ∈ Rp are the state, input, and output vectors subject to constraints xmin ≤ xk ≤ xmax , umin ≤ uk ≤ umax  and ymin ≤ yk ≤ ymax

The objective function is to minimize Jp(U) = ½ U’QpU + F’pU + ½ Mp  with constraints GpU ≤ Kp

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// type of QP ½\*U'QpU + Fp'U + ½Mp

// Gp'U <= Kp

read Qp, Fp, Mp, Gp, Kp

// convert to dual form ½Y'QdY + Fd'Y + ½Md

// Y > 0

compute Qd, Fd, Md using Qp, Fp, Mp, Gp, Kp

Qd = GpQp-1Gp'

Fd = (Kp + GpQp-1Fp)

Md = F′pQp−1Fp − Mp

h=1

while not converged:

update Y using update rule: Yh+1(i) = [[(Qd- + theta)Yh + F- ] ÷ [(Qd+ + theta)Yh + F+]]×Yh(i)

// where Qd-(i)(j) = max(0,-Qd(i)(j))

// where Qd+(i)(j) = max(0, Qd(i)(j))

// theta is a matrix such that theta(i)(i) >= Qd-(i)(i)

h = h+1

compute U\* using Y\* according to the rule U\* = -Qp-1(Fp + Gp'Y\*)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

convergence check:- GpU <= Kp + max(erc\*|Kp|, eac[**1**])

Jp + Jd <= eaJ

(Jp + Jd)/abs(Jd) <= eaJ

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Fig 1: The algorithm for PQP

### Execution

The purpose of this methodology is to estimate the execution time of the optimized PQP algorithm for a real-world MPC. We have worked out this problem in four phases:

* Phase 1: Implement a CPU version code of PQP algorithm

We have implemented a C program to run on the CPU. First we have created all the matrices needed for the algorithm to run and populated them with input. Sequentially, we have performed the following modules:

**Compute Fp** : This is a PQP utility function for computing Fp from Fp1, Fp2, Fp3. Parameters are (pointer of Fp, pointer of Fp1, pointer of Fp2, pointer of Fp3, pointer of D, pointer of x) dimension of (n X m matrix). The result is stored in Fp.

The formula for Fp generation: Fp = Fp1\*D+Fp2\*x-Fp3

This function uses the utility function matrixMultiply(), matrixAdd().

**Compute Mp :** This is a PQP utility function for computing Mp from Mp1, Mp2, Mp3, Mp4, Mp5, Mp6, D, X. Parameters are (pointer of Mp, pointer of Mp1, pointer of Mp2, pointer of Mp3, pointer of Mp4, pointer of Mp5, pointer of Mp6, pointer of D, pointer of x).

The result is stored in Mp.

The formula for Mp generation: Mp= 0.5 .\* x'\*Mp1\*x + D'\*Mp2\*x+ 0.5 .\*D'\*Mp3\*D - 0.5 .\*Mp4\*x - 0.5.\*Mp5\*D+0.5\*Mp6

[ Where, '=> transpose operation and .\* => element wise multiplication]

This function uses the utility function matrixMultiply(), matrixAdd().

**Convert to dual :** This is a PQP utility function for converting primal to dual form of PQP.

The parameters are (pointer of Qd,pointer of Fd,pointer of Md,pointer of Qp\_inv, pointer of Gp,pointer of Kp,pointer of Fp,pointer of Mp,int N, int M)

This function uses utility function matrixMultiply(), matrixAdd(), computeQd(), computeFd(), computeMd().

Temp variables are used to keep the intermediate result.

**Solve Quadratic Dual :** This function takes the matrices of dual form of QP as input and performs update operation on the vector Y iteratively till convergence. This function also takes input of the primal form of QP to check for convergence.

**Compute U from Y :** This is PQP utility function for compute U(solution of primal QP) from Y(solution of dual QP). The parameters are (pointer of U vector,pointer of Y vector, pointer of Fp, pointer of Gp, pointer of Qp\_inv,int N, int M) dimension of (n X m matrix)

The result is stored in Vector U.

* Phase 2: Naive implementation of the PQP algorithm for GPU and then its optimization using tiling and shared memory

**We have designed and deployed the kernels to run on GPU:**

* void initMat(float \*mat, float val, int N);
* void copyMatrix(float \*output, float \*mat, int a, int b);
* void transpose(float \*odata, float \*idata, int n, int m);
* void matrixMultiply(float \*output, float \*mat1, int transpose1, float \*mat2, int transpose2, int a, int b, int c);
* void matrixAdd(float \*A, float \*B, float sign, int a, int b);
* void negateMatrix(float \*mat, int n, int m) ;
* void matrixPos(float \*mat1, float \*mat2, int n, int m);
* void matrixNeg(float \*mat1, float \*mat2, int n, int m);
* void diagonalAdd(float \*theta, float \*tmp, int N);
* void compare(float \*GpU, float \*Kp, int \*re, int N);
* void updY(float \*Y\_next, float \*numerator, float \*denominator, float \*Y, int N)
* Phase 3: Implement Coarsening

Parallel execution sometimes requires doing redundant memory accesses and/or calculations. Coarsening is a technique to merge multiple threads so each resulting thread calculates multiple output elements

– Perform the redundant work once and save result into registers

– Use register result for calculating all output elements

Merged kernel code will use more registers

– May reduce the number of threads allowed on an SM

– Increased efficiency may outweigh reduced parallelism, especially if ample for a given hardware.

In order to improve resource utilization per kernel launch, a thread based coarsening was implemented.

Example: The following function has been coarsened to perform the kernel operation with half the number of threads.

\_global\_\_ void copyMatrixCuda(float \*output, float \*mat, int a, int b)

{

int blockNum = blockIdx.z \* (gridDim.x \* gridDim.y) + blockIdx.y \* gridDim.x + blockIdx.x;

int threadNum = threadIdx.z \* (blockDim.x \* blockDim.y) + threadIdx.y \* (blockDim.x) + threadIdx.x;

int id = blockNum \* (blockDim.x \* blockDim.y \* blockDim.z) + threadNum;

if(id<a\*b)

{

int id1=2\*id+0;

int id2=2\*id+1;

output[id1] = mat[id1];

output[id2] = mat[id2];

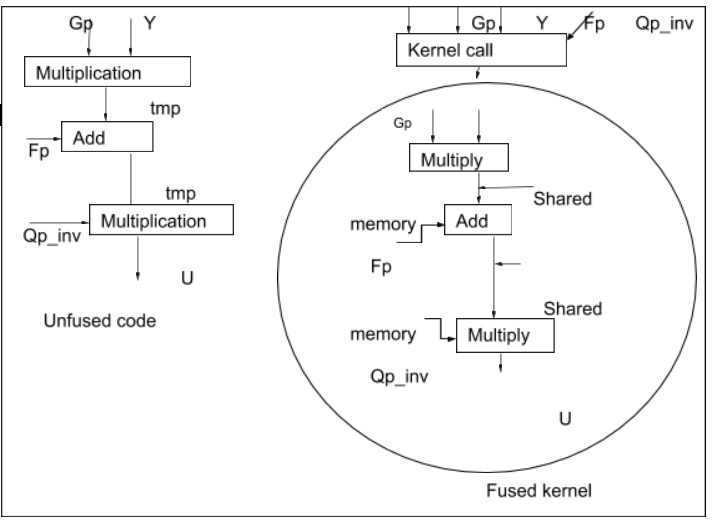
}

}

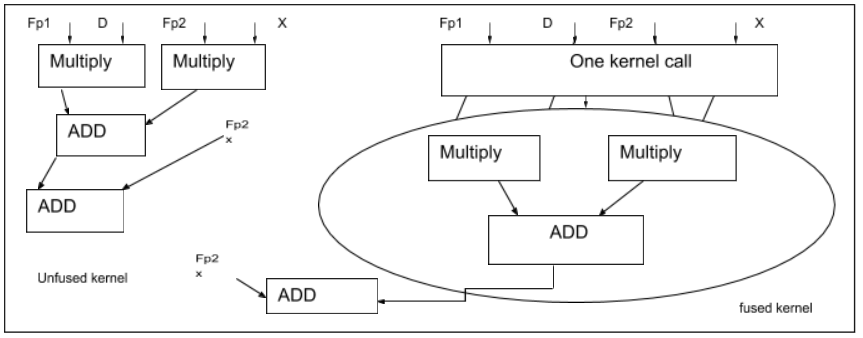
The function copies the value stored in \*mat to an output matrix designated with \*output. Coarsening is implemented by means of doubling the thread load by reducing the number of threads launched by a factor of 2.

* Phase 4: Implement Fusion.

In our implementation, we have done some thread-level kernel optimization. A group of kernels can be combined together to form a fused kernel. It is a challenging task to determine fused kernel if there is any data dependency as kernel fusion does not always help to improve the performance. In our work, we have selected two specific functions void computeUfromY() and void computeFp() where multiple kernel call performed to execute the function code. We have applied inner thread fusion in those functions as data size and grid and block dimension the same for each kernel call in each of these particular functions. In void computeUfromY() there has three kernel call two matrix- vector multiplication and one vector-vector addition. We have fused these three kernels in one kernel which is described in given below diagram.



In void computeFp(),we have also used two matrix-vector multiplication kernel call and two vector-vector addition kernel. Here we optimized the kernel using fusion of three kernels call which is shown in given below figure:



Performance Analysis

### CPU Version

Flat profile:

Each sample counts as 0.01 seconds.

% cumulative self self total

time seconds seconds calls ms/call ms/call name

100.06 0.36 0.36 330011 0.00 0.00 matrixMultiply

0.00 0.36 0.00 842408 0.00 0.00 max

0.00 0.36 0.00 510048 0.00 0.00 initMat

0.00 0.36 0.00 510046 0.00 0.00 newMatrix

0.00 0.36 0.00 360009 0.00 0.00 copyMatrix

0.00 0.36 0.00 90001 0.00 0.00 matrixAdd

0.00 0.36 0.00 60000 0.00 0.00 computeCost

0.00 0.36 0.00 30000 0.00 0.00 computeUfromY

0.00 0.36 0.00 30000 0.00 0.00 negateMatrix

0.00 0.36 0.00 29999 0.00 0.00 checkFeas

0.00 0.36 0.00 29999 0.00 0.00 compare

0.00 0.36 0.00 29999 0.00 0.01 terminate

0.00 0.36 0.00 29998 0.00 0.00 updY

0.00 0.36 0.00 29998 0.00 0.00 updateY2

0.00 0.36 0.00 3 0.00 0.00 matrixNeg

0.00 0.36 0.00 2 0.00 0.00 matrixPos

0.00 0.36 0.00 1 0.00 0.00 Gauss\_Jordan

0.00 0.36 0.00 1 0.00 0.00 computeFd

0.00 0.36 0.00 1 0.00 0.00 computeFp

0.00 0.36 0.00 1 0.00 0.00 computeMd

0.00 0.36 0.00 1 0.00 0.01 computeMp

0.00 0.36 0.00 1 0.00 0.00 computeQd

0.00 0.36 0.00 1 0.00 0.00 computeQdn\_theta

0.00 0.36 0.00 1 0.00 0.00 computeQdp\_theta

0.00 0.36 0.00 1 0.00 0.00 computeTheta

0.00 0.36 0.00 1 0.00 0.01 convertToDual

0.00 0.36 0.00 1 0.00 0.00 diagonalAdd

0.00 0.36 0.00 1 0.00 0.00 input

0.00 0.36 0.00 1 0.00 360.20 solveQuadraticDual

% the percentage of the total running time of the

time program used by this function.

cumulative a running sum of the number of seconds accounted

seconds for by this function and those listed above it.

self the number of seconds accounted for by this

seconds function alone. This is the major sort for this

listing.

calls the number of times this function was invoked, if

this function is profiled, else blank.

self the average number of milliseconds spent in this

ms/call function per call, if this function is profiled,

else blank.

total the average number of milliseconds spent in this

ms/call function and its descendents per call, if this

function is profiled, else blank.

**Analysis**

**N = 1000, M = 500**

**CPU -**

real 0m4.651s

user 0m4.651s

sys 0m0.004s

**GPU\_unoptimized -**

real 0m4.651s

user 0m4.651s

sys 0m0.004s

**GPU\_optimized-**

real 0m1.674s

user 0m1.453s

sys 0m0.213s