Efficient Convex Optimization on GPUs for Embedded Model Predictive Control

P.J Avinash,Rakesh Kumar Arwini,Aniket Raj,Vikash Khakhal,Kanhaiya Agarwal

April 2019

1 Abstract

A key component in enabling the application of model predictive control (MPC) in fields such as automotive, aerospace, and factory automation is the availability of low-complexity fast optimization algorithms to solve the MPC finite horizon optimal control problem in architectures with reduced computational capabilities. Graphics processing units(GPUs) are specialized electronic circuits designed primarily for 3D game rendering and alter memory to accelerate the creation of images in a frame buffer intended for output to display devices and are often used in embedded systems,game consoles, personal computers, workstations, and mobile phones. Their highly parallel structure makes them more efficient than general-purpose CPUs in computationally heavy tasks and for algorithms that process large blocks of data in parallel. Because most of these computations involve matrix and vector operations, engineers and scientists have increasingly studied the use of GPUs for non-graphical calculations; they are especially suited to other embarrassingly parallel problems.

Relative to traditional control techniques such as PID, MPC is very computationally demanding. MPC relies on the real-time solution of a convex optimization problem to compute the control input(s) to a system. Quadratic programming algorithms used for solving convex optimization problems generally can be parallelized. Here We investigate two different gradient-based algorithms, the Alternating Direction Method of Multipliers(ADMM) and Parallel Quadratic Programming (PQP), for solving the QP that occurs in a large class of MPC problems.The performance of these Algorithms is guided by matrix operations like multiplications, additions, and matrix-vector products. In this project work, we have implemented the single-threaded CPU version of the PQP and ADMM algorithm which is used for solving the convex optimization MPC problem. We have implemented the unoptimized version of GPU implementation of PQP and ADMM. We have implemented SGEMV (Single-precision General Matrix-Vector Multiplication) and SGEMM (Single-precision General Matrix-Matrix Multiplication) kernels using efficient block and grid dimension, tiling, using shared memory. We have used adaptive kernel fusion with proper synchronization in such a way that the data movement and launching overhead are significantly reduced.

2 Introduction

Model predictive control (MPC) (Rawlings Mayne, 2009) is a powerful framework for optimal control of multivariable constrained systems. MPC is based on the repeated, receding horizon solution of a finite-time optimal control problem formulated from the system dynamics, the constraints on system states, inputs, and outputs, and the cost function describing the control objectives and their relative priority.MPC has several advantages over traditional control techniques like PID(Proportional Integral Derivative) control technique. Although PID control technique is most dominant in terms of usage, there are several other systems that simply cannot be controlled by these traditional techniques. The advantage of MPC is that it can handle MIMO systems(Multiple input-Multiple output). It has been initially developed and used in the process industries in chemical plants and oil refineries since the 1980s. These applications typically require several hundreds of inputs and outputs but slow dynamics. In recent years MPC had seen extensive usage in Robotics(self-driving vehicles, Motion controlled robotic equipment etc), automotive, aerospace, and factory automation. These are typically characterized by a relatively less number of control inputs and outputs but very fast dynamics and high sensitivity to the controller's output. Larger Plant controllers may need to compute a new set of inputs and outputs every few minutes and some embedded MPC applications may require for every few seconds and milliseconds also. The need to solve computationally heavy Mathematical problems at such rates makes it challenging to use MPC in many applications.

In this Project work, we look at the Mathematical formulation of MPC using linear Prediction models with Convex performance Objective functions subject to some linear constraints, implement them on GPU and Analyze performance. These problems are essentially Quadratic Programming subjected to constraints i.e. Constrained Quadratic Programming. There is a good number of Techniques for solving this problem like the Interior Point Method, Active Set Method, Augmented Lagrangian Method, and Gradient-based Iterative Methods. Here we present the implementation of two Gradient Based iterative methods namely Parallel Quadratic Programming(PQP) and Alternating Direction Method of Multipliers (ADMM). Here we make a Serial implementation for the CPU and Compare the performance against parallelized version on GPU. We demonstrate MPC performance with optimized implementation of PQP and ADMM algorithms on GPU.

3 Theory

Here in this section We will have a brief look at a control system. Generally, A control system deals with Modifying inputs to the system in order to generate desired outputs. The most general case is that a control system may have Multiple inputs and Multiple outputs (Typically called A MIMO system) and the size of input and output vectors here depends on how complex the control plant or system is. The size of input /output vectors may vary from 3 to several hundred. In General, a control system can be Non-linear. But for the sake of Modelling and analysing, Most systems are modelled with state-space modelling and linearized about the operating point.

The most general state-space representation of a linear system with p inputs, q outputs and n state variables is written in the following form

$$\dot{\mathbf{x}}(t) = A(t)\mathbf{x}(t) + B(t)\mathbf{u}(t)$$

$$\mathbf{y}(t) = C(t)\mathbf{x}(t) + D(t)\mathbf{u}(t)$$

Here $\mathbf{x}(\cdot)$is called the "state vector",$\mathbf{x}(t) \in R^n$;
$\mathbf{y}(\cdot)$ is called the "output vector", $\mathbf{y}(t) \in R^q$;
$\mathbf{u}(\cdot)$ is called the "input (or control) vector", $\mathbf{u}(t) \in R^p$;

In a control system problem we are interested in finding appropriate $\mathbf{u}(t)$ in order to achieve desired output.

3.1 Model Predictive Control

Model Predictive Control (MPC) is a multivariable control algorithm that uses:
1.Internal dynamic model of the process
2.a history of past control moves and
3.An optimization cost function J over the receding prediction horizon.

The main advantage of MPC is the fact that it allows the current timeslot to be optimized while keeping future time slots in account. This is achieved by optimizing a finite time horizon, but only implementing the current time slot and then optimizing again.

For a linear MPC, We have the following equations

$$x_{k+1} = Ax_k + Bu_k$$
$$y_k = Cx_k + Du_k$$

where x_k is state vector, u_k is the input vector and y_k is the output vector. subjected to the following constraints

$$x_{min} \leq x_k \leq x_{max}$$
$$u_{min} \leq u_k \leq u_{max}$$
$$y_{min} \leq y_k \leq y_{max}$$

and A, B, C, and D are state matrix, input matrix, state constraint Matrix, and output constraint matrix respectively.$x \in \mathrm{R}^n$,$y \in \mathrm{R}^p$ and $u \in \mathrm{R}^m$
At step t $\in Z_{0+}$, MPC solves a finite Horizon Optimal Control Problem.

$$\min_{U_t} \quad x_{N|t}^T P x_{N|t} + \sum_{k=0}^{N-1} z_{k|t}^T Q z_{N|t} + u_{k|t}^T R u_{k|t}$$

The above optimization Problem can be solved using many different methods. Here we look at Two gradient-based iterative algorithms Namely Parallel Quadratic Programming (PQP) and Alternating Direction Method of Multipliers(ADMM)

3.2 Parallel Quadratic Programming

This iterative Gradient based algorithm is formulated in primal form as below.

$$\min_{U} \quad J_P(U) = \frac{1}{2}U^T Q_P U + F_P^T U + \frac{1}{2}M_P$$

such that

$$G_P U \leq K_P$$

In order to solve the above equation subjected to a constraint, we solve its dual problem formulated as below.

$$\min_{Y} \quad J_D(Y) = \frac{1}{2}Y^T Q_D Y + F_D^T Y + \frac{1}{2}M_D$$

such that

$$Y \geq 0$$

Given Q_P,F_P,G_P,K_P We have to find Y where $Q_D = G_P Q_P^{-1} G_P^T$, $F_D = K_P + G_P Q_P^{-1} F_P$, and Y$\in$R$_q^n$, so number of variables in second equation is same as number of constraints in first problem.

Since, we have to solve the model predictive control optimization problem, so the above primal form will be changed as shown below.

$$\min_{U} \quad J_P(U) = \frac{1}{2}U^T Q_P U + x^T C_P^T U + x^T del_P x$$

such that

$$G_P U \leq S_P x + W_P$$

In order to solve the above equation subjected to a constraint, we solve its dual problem formulated as below.

$$\min_{Y} \quad J_D(Y) = \frac{1}{2}Y^T Q_D Y + x^T S_D^T Y + W_D^T Y + \frac{1}{2}x^T del_D x$$

such that

$$Y \geq 0$$

Given Q_P,C_P,G_P,S_P,W_P We have to find Y where $Q_D = G_P Q_P^{-1} G_P^T$, $S_D = S_P + G_P Q_P^{-1} C_P$, $W_D = W_P$, $del_D = C_P^T Q_P^{-1} C_P - del_P$, and Y$\in$R$_q^n$,so the number of variables in the second equation is same as the number of constraints in the first problem.
Furthermore, the primal optimal solution can be computed from the dual optimal solution via

$$U(Y^*) = tau_D x + E_D Y^*$$

where $tau_D = -Q_P^{-1} G_P^T$, $E_D = -Q_P^{-1} C_P$.

In this project we are primarily concerned with computational complexity due to matrix operations at each stage of PQP, the Number of FLOPs(floating point operations per second), and efficient parallelization of the algorithm.

PQP Algorithm is divided into two parts: Initialization and Iteration.(PQP-init and PQP-iter).At every sample time K the problem matrices are initialized using Matrix-Matrix, Matrix-vector, and vector-vector(inner) products. The total number of flops required is $2n_u^2 n_q + 2n_q^2 n_u + n_u^2 + 7n_q n_u - 1$ FLOPs where $n_u (= Nm)$ is numbers of variables in primal (m is the dimension of input vector U)and n_q is the number of variables in dual i.e number of constraint equations in Primal. The iterative part of PQP is again divided into two parts: update and acceleration. The update is a normal execution path during iteration and the acceleration part is to increase the rate of convergence to the actual solution. updating after each iteration will make our solution Y move towards the actual solutionY^*. Y after each step-k will be updated by following expression

$$Y_{k+1} = \frac{[(Q_D^- + \phi)Y_k + F_D^-]}{[(Q_D^+ + \phi)Y_k + F_D^+]} Y_k$$

The above step is composed of two matrix-vector products, two vector sums, an element-wise division, and multiplication totally costing $2n_q n_u + 5n_q^2 + 7n_q - n_u$ steps. The acceleration step is problem-dependent, so the number of FLOPS may vary with numerical conditions but the worst case is $4n_u^2 + 6n_q - 1$. The algorithm terminates when solution Y has converged within some threshold of Y^*, but its verification requires $2n_u n_q + 4n_q^2 + 8n_q$ FLOPS in the worst case. Testing for convergence at every iteration is a waste of computation. In general, testing for convergence is done for every N iteration where N will be in the order of 100s depending on the problem. In some implementations, convergence is only checked after the final iteration, and the number of iterations is fixed in this case. But sometimes we may perform more iterations than is required for convergence.

3.3 Alternating Direction Method of Multipliers

ADMM is another Gradient-based iterative algorithm. In ADMM finite horizon problem is formulated as below.

$$\min_{\xi} \quad J_p(\xi) = \frac{1}{2}\xi^T Q_p \xi$$

$$s.t \quad G_P \xi = K_P$$

$$\xi_{min} \leq \xi \leq \xi_{max}$$

where $\xi = [X_t^{'} U_t^{'} Y_t^{'} S_t^{'}]^{'}$, $X_t = [x_{1|t}^{'} ... x_{N|t}^{'}]$, $Y_t = [y_{0|t}^{'} ... y_{N-1|t}^{'}]$ and $S_t = SX_{N|t}$. Number of variables $n_u = N(p+m+n)$ To simplify the problem, a New copy of ξ(optimization vector) denoted by ζ is used to enforce boundary constraints.

$$\xi = \zeta$$

This equality constraint is dualized by augmented lagrangian form by using vector Lagrange multiplier $\lambda \in \mathrm{R}_u^n$ as shown below where β is step size parameter.

$$\min_{\xi} \quad J_p(\xi) = \frac{1}{2}\xi^T Q_p \xi + \frac{\beta}{2}||\xi - \zeta - \lambda||^2$$

$$s.t \quad G_P\xi = K_P$$

$$\xi_{min} \leq \xi \leq \xi_{max}$$

AMMD iteratively adjusts λ to seek values of ξ and ζ to solve the above QP. At optimum $\xi = \zeta$. The values of ξ, λ and ζ are updated iteratively using following relations.

$$\xi_{k+1} = M(+\xi_k) + NK_E$$

$$\zeta_{k+1} = proj_{\zeta \in [\xi_{min}, \xi_{max}]} \quad \xi_{k+1} - \lambda_k$$

$$\lambda_{k+1} = \lambda_k + \zeta_{k+1} - \xi_{k+1}$$

proj denotes projection and M,N are matrices obtained from above mentioned dualized augmented lagrangian form. ADMM is also divided into two phases. Namely initialization Phase and iteration Phase(ADMM-init, ADMM-iter). initialization occurs once at the beginning and requires $2n_u^2 + 2n_u n_q$ FLOPs.The iteration part of the algorithm contains a single matrix-vector product and several vector operations and a total of $2n_u^2 + 5n_u$ FLOPs. Similar to the PQP algorithm, convergence is tested for several Steps at once which is problem dependent. Testing convergence takes $9n_u + 1$ FLOPs.

4 GPU computing

Table 1: CPU specifications

CPU	
Lithography	
cores	
Threads	
Core Clock	
Memory Bandwidth	
Cache	
Memory size	
Memory types	
TDP	

Table 2: GPU specifications

GPU	
Architecture	
CUDA cores	
Multiprocessors	
CUDA cores/SP	
Core clock	
Memory clock	
Memory Bus Width	
Global Memory	
L2 Cache	
Constant Memory	
Saturated Memory per block	
Shared Memory per block	
Register per Block	
copy engines	
Integrated GPU Sharing host Memory	

4.1 Kernels and Implementation

A kernel is the unit of work that the main program running on the host computer offloads to the GPU for computation on that device. In CUDA, launching a kernel requires specifying three things: 1)The dimensions of the grid)The dimensions of the blocks)The kernel function to run on the device. Kernel functions are specified by declaring them ”global” in the code, and a special syntax is used in the code to launch these functions on the GPU while specifying the block and grid dimensions. These kernel functions serve as the entry points for the GPU computation, much the same way main() serves as the entry point in an ordinary C program. The main kernels which are important for our problem are the MATRIX-MATRIX Multiplication kernel and MATRIX-VECTOR Multiplication kernels since every term in our equations are either one of the above computation kernels(vector and matrix generations).

4.2 MATRIX-MATRIX Multiplication kernel

Let’s say the function for matrix multiplication is MatMul(). So the basic working is as follows. MatMul(Matrix A, Matrix B, Matrix C) This function takes two matrices A and B as input and fills the entries of matrix C with the product. We first allocate memory on the device for matrix A, and copies A onto the device’s global memory. Then we do the same for matrix B. We allocate space on the device for C, the matrix product, computes the number of blocks needed to cover the product matrix, and then launch a kernel with that many blocks. When the kernel is done, we read matrix C off of the device, and free the global memory.

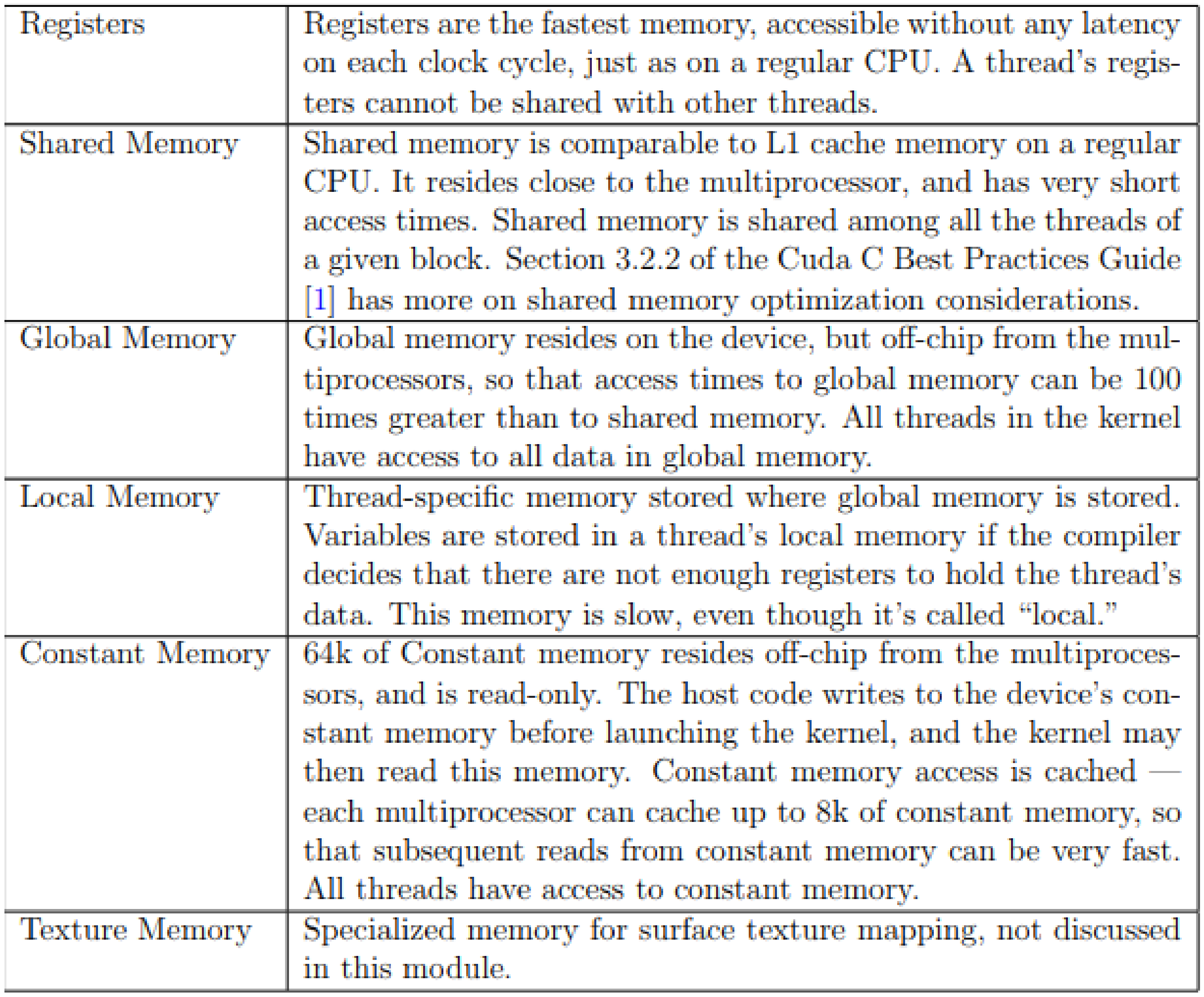
The code for the Matmul kernel is as follows.

```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
  // Each thread computes one element of C
  // by accumulating results into Cvalue
  float Cvalue = 0;
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  if(row > A.height || col > B.width) return;
  for (int e = 0; e < A.width; ++e)
    Cvalue += A.elements[row * A.width + e] *
                             B.elements[e * B.width + col];
  C.elements[row * C.width + col] = Cvalue;
}
```

Figure 1: Matmul kernel

Similarly for the matrix vector multiplication kernel is also written.

But there is scope for optimization in the above kernels. For this we need to know the memory hierarchy. These are the kinds of memory that are available on a GPU.



Registers	Registers are the fastest memory, accessible without any latency on each clock cycle, just as on a regular CPU. A thread's registers cannot be shared with other threads.
Shared Memory	Shared memory is comparable to L1 cache memory on a regular CPU. It resides close to the multiprocessor, and has very short access times. Shared memory is shared among all the threads of a given block. Section 3.2.2 of the Cuda C Best Practices Guide [1] has more on shared memory optimization considerations.
Global Memory	Global memory resides on the device, but off-chip from the multiprocessors, so that access times to global memory can be 100 times greater than to shared memory. All threads in the kernel have access to all data in global memory.
Local Memory	Thread-specific memory stored where global memory is stored. Variables are stored in a thread's local memory if the compiler decides that there are not enough registers to hold the thread's data. This memory is slow, even though it's called "local."
Constant Memory	64k of Constant memory resides off-chip from the multiprocessors, and is read-only. The host code writes to the device's constant memory before launching the kernel, and the kernel may then read this memory. Constant memory access is cached — each multiprocessor can cache up to 8k of constant memory, so that subsequent reads from constant memory can be very fast. All threads have access to constant memory.
Texture Memory	Specialized memory for surface texture mapping, not discussed in this module.

Figure 2: Memory Hierarchy in GPU

In light of the memory hierarchy described above, let us see what optimizations we might consider. Looking at the loop in the kernel code, we notice that each thread loads (2 x A.width) elements from global memory — two for each iteration through the loop, one from matrix A and one from matrix B. Since accesses to global memory are relatively slow, this can bog down the kernel, leaving the threads idle for hundreds of clock cycles, for each access. One way to reduce the number of accesses to global memory is to have the threads load portions of matrices A and B into shared memory, where we can access them much more quickly. Ideally we would load both matrices entirely into shared memory, but unfortunately, shared memory is a rather scarce resource, and won't hold two large matrices. One way of doing this is suggested by figure below

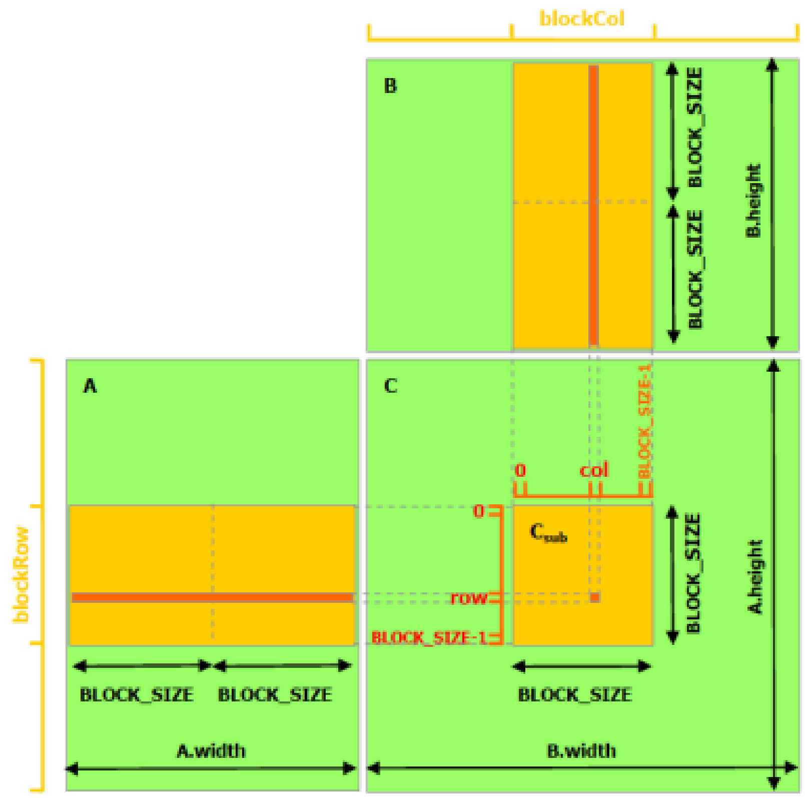
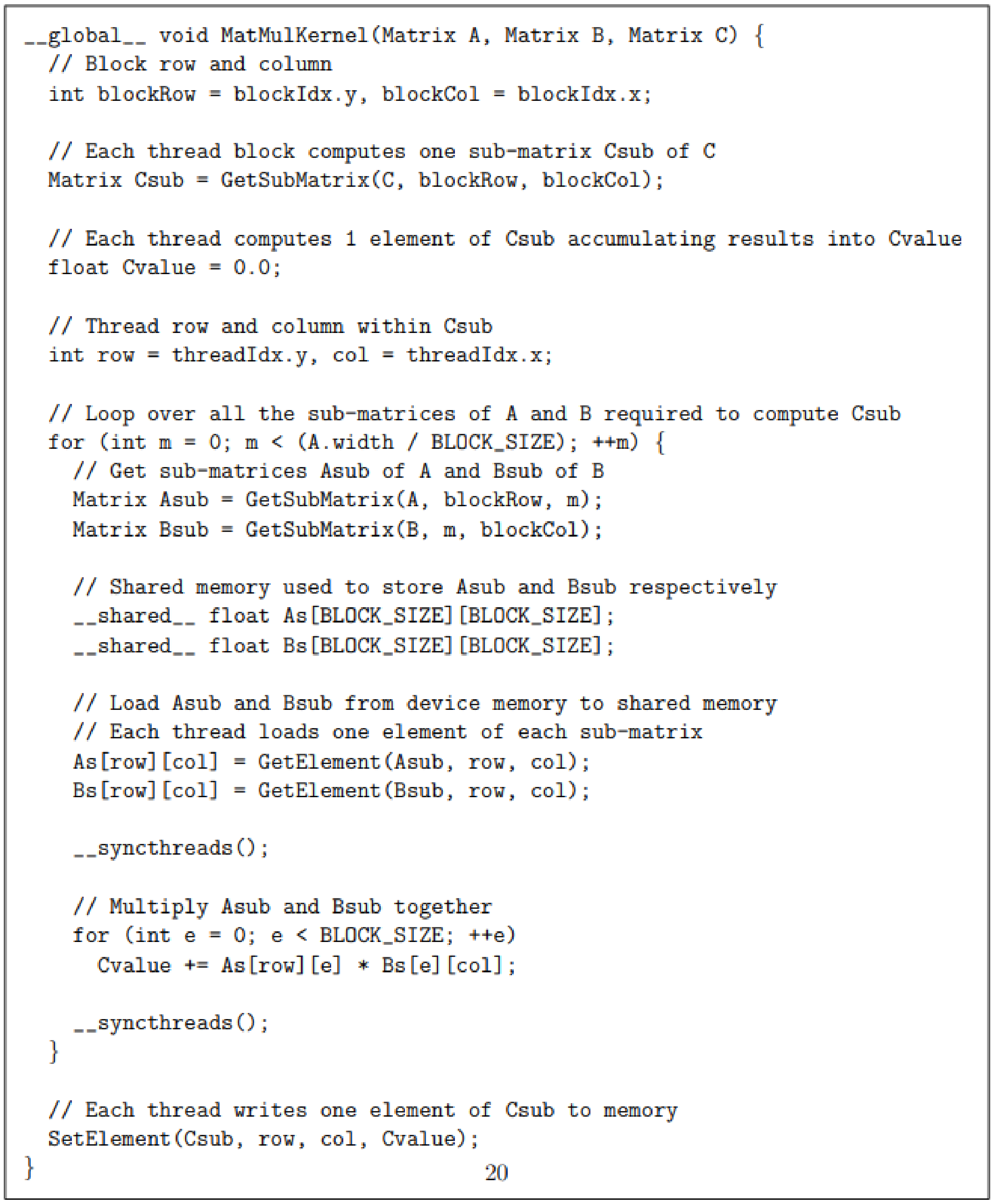


Figure 3: shared memory

Matrix A is shown on the left and matrix B is shown at the top, with matrix C, their product, on the bottom right. This is a nice way to lay out the matrices visually since each element of C is the product of the row to its left in A, and the column above it in B. The row and column sort of "aim" for their product in C. In that figure, and in our sample code, we will use square thread blocks of dimension BLOCKSIZE BLOCKSIZE and will assume that the dimensions of A and B are all multiples of BLOCKSIZE. Again, each thread will be responsible for computing one element of the product matrix C. For reference, consider the item highlighted in red in the matrix C, in the above figure (Note that the yellow square in matrix C represents one thread-block's-worth of elements, whereas the red box inside the yellow square represents a single entry in C, and hence a single thread.) Our thread computes this element in C by multiplying together the red row shown in A, and the red column shown in B, but it will do it in pieces, as we will now discuss. We may decompose matrices A and B into non-overlapping submatrices of size BLOCKSIZE BLOCKSIZE. If we look at our red row and red column, they will pass through the same number of these submatrices, since they are of equal length. If we load the left-most of those submatrices of matrix A into shared memory, and the top-most of those submatrices of matrix B into shared memory, then we can compute the first BLOCKSIZE products and add them together just by reading the shared memory. But here is the benefit: as long as we have those submatrices in shared memory, every thread in our thread block (computing the BLOCKSIZEBLOCKSIZE submatrix of C) can compute that portion of their sum as well from the same data in shared memory. When each thread has computed this sum, we can load the next BLOCKSIZE BLOCKSIZE submatrices from A and B, and continue adding the term-by-term products to our value in C. And after all of the submatrices have been processed, we will have computed our entries in C. So the code for matrix multiplication using shared memory is



```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
  // Block row and column
  int blockRow = blockIdx.y, blockCol = blockIdx.x;

  // Each thread block computes one sub-matrix Csub of C
  Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

  // Each thread computes 1 element of Csub accumulating results into Cvalue
  float Cvalue = 0.0;

  // Thread row and column within Csub
  int row = threadIdx.y, col = threadIdx.x;

  // Loop over all the sub-matrices of A and B required to compute Csub
  for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
    // Get sub-matrices Asub of A and Bsub of B
    Matrix Asub = GetSubMatrix(A, blockRow, m);
    Matrix Bsub = GetSubMatrix(B, m, blockCol);

    // Shared memory used to store Asub and Bsub respectively
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    // Load Asub and Bsub from device memory to shared memory
    // Each thread loads one element of each sub-matrix
    As[row][col] = GetElement(Asub, row, col);
    Bs[row][col] = GetElement(Bsub, row, col);

    __syncthreads();

    // Multiply Asub and Bsub together
    for (int e = 0; e < BLOCK_SIZE; ++e)
      Cvalue += As[row][e] * Bs[e][col];

    __syncthreads();
  }

  // Each thread writes one element of Csub to memory
  SetElement(Csub, row, col, Cvalue);
}
```

20

Figure 4: shared memory

Similarly, this shared memory concept is also used for Matrix-Vector Multiplication.

The code that takes advantage of shared memory in this way is modestly more complex, and it is reasonable to ask whether the benefits outweigh the costs. A good way to analyze this is to consider a single element of matrix A, and ask how many times it is loaded from global memory. In our first program, which did not take advantage of shared memory, an entry is loaded by each thread that needs it for its computation. A glance at the above figure (ignoring the blocks for now) shows that we load each entry of A once for each column of B. But in our shared memory code, we load this entry only when we load the block of A containing that entry, and that will happen once for each block in the product matrix C in the same row as that entry. That is, it will be loaded (width of B)/(BLOCKSIZE) times. If BLOCKSIZE equals 16, then we load this entry from global memory 1/16th as many times as we do without making use of shared memory. In our timings below, we will show that this translates into real-time savings.

4.3 Performance Analysis

PQP				
Matrix Dimension	CPU	GPU naive	GPU naive shared	GPU optimized
1024 x 1024	136.37	4.01	3.82	3.82
512 x 512	34.9	1.18	1.07	1.07
256 x 256	7.95	0.28	0.23	0.23
128 x 128	1.25	0.13	0.089	0.089

ADMM			
Matrix Dimension	CPU	GPU naive	GPU optimized
1024 x 1024	43.462	21.39	20.52
512 x 512	9.738	4.06	3.83
256 x 256	2.156	0.7484	0.7236
128 x 128	0.483	0.1175	0.0.089

4.4 Kernel Fusion

5 SUMMARY

In this project, we have implemented the single-threaded CPU implementation of PQP and ADMM of the model predictive control for convex optimization. We have implemented the unoptimized GPU implementation of PQP and ADMM. We have experimented with the global memory and shared memory and understand how it is affecting the run time of the code. We reported improvement in performance if when we are using shared memory over global memory. We have implemented the SGEMV (single-precision general matrix-vector multiplication) and SGEMM (Single-precision General Matrix-Matrix Multiplication) kernels using efficient block and grid dimension, tiling, using shared memory. Because of the time constraint, we cannot implement the kernel fusion in the update and check function of PQP and ADMM implementation. Since we are measuring the time of each update and check steps, so when we implemented the optimized GPU implementation of PQP and ADMM, we did not get the performance improvement, because it is not affecting the update and check function of the ADMM and PQP. If you will measure the whole time of each iteration then you will observe the improvement in performance.

6 References