MITSUBISHI ELECTRIC RESEARCH LABORATORIES
https://www.merl.com

Efficient Convex Optimization on GPUs for Embedded Model Predictive Control

Yu, Leiming; Goldsmith, Abraham; Di Cairano, Stefano

TR2017-033 February 05, 2017

Abstract

GPU applications have traditionally run on PCs or in larger scale systems. With the introduction of the Tegra line of mobile processors, NVIDIA expanded the types of systems that can exploit the massive parallelism offered by GPU computing architectures. In this paper, we evaluate the suitability of the Tegra X1 processor as a platform for embedded model predictive control. MPC relies on the real time solution of a convex optimization problem to compute the control input(s) to a system. Relative to 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. However, until the introduction of the Tegra, there has never been an off-the-shelf embedded processor that would enable a massively parallel embedded implementation. We investigate two different gradient based algorithms, ADMM and PQP, for solving the QP that occurs in a large class of MPC problems. The performance of these algorithms is dominated by the performance of matrix-matrix and matrix-vector products. Our work focuses on maximizing the performance of these operations for relatively small matrices of 100 to 1000 elements per dimension, which are common in the MPC control implementations found in automotive and factory automation applications. Modern BLAS libraries for CPUs and GPUs are quantitatively evaluated. We create SGEMV kernels that can outperform the state-of-the-art cuBLAS by 2.3x on TX1. Different kernel fusion schemes utilizing concurrent kernel execution and zero copy mechanisms are investigated. For ADMM, our implementation achieves 46.6x speedup over the single threaded CPU version and 2.7x speedup over the optimized OpenBLAS version. For PQP, we achieve 41.2x speedup over the single threaded CPU version and 4.2x speedup over the OpenBLAS version.

Workshop on General Purpose Processing with Graphics Processing Units

Mitsubishi Electric Research Laboratories, Inc.
201 Broadway, Cambridge, Massachusetts 02139

Efficient Convex Optimization on GPUs for Embedded Model Predictive Control

Leiming Yu *
Electrical and Computer Engineering
Northeastern University
Boston, MA, USA 02115
ylm@ece.neu.edu

Abraham Goldsmith, Stefano Di Cairano
Mitsubishi Electric Research Laboratories
201 Broadway
Cambridge, MA, USA 02139
{goldsmith,dicairano}@merl.com

ABSTRACT

GPU applications have traditionally run on PCs or in larger scale systems. With the introduction of the Tegra line of mobile processors, NVIDIA expanded the types of systems that can exploit the massive parallelism offered by GPU computing architectures. In this paper, we evaluate the suitability of the Tegra X1 processor as a platform for embedded model predictive control. MPC relies on the real time solution of a convex optimization problem to compute the control input(s) to a system. Relative to 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. However, until the introduction of the Tegra, there has never been an off-the-shelf embedded processor that would enable a massively parallel embedded implementation.

We investigate two different gradient based algorithms, ADMM and PQP, for solving the QP that occurs in a large class of MPC problems. The performance of these algorithms is dominated by the performance of matrix-matrix and matrix-vector products. Our work focuses on maximizing the performance of these operations for relatively small matrices of 100 to 1000 elements per dimension, which are common in the MPC control implementations found in automotive and factory automation applications. Modern BLAS libraries for CPUs and GPUs are quantitatively evaluated. We create SGEMV kernels that can outperform the state-of-the-art *cuBLAS* by 2.3x on TX1. Different kernel fusion schemes utilizing concurrent kernel execution and zero copy mechanisms are investigated. For ADMM, our implementation achieves 46.6x speedup over the single threaded CPU version and 2.7x speedup over the optimized OpenBLAS version. For PQP, we achieve 41.2x speedup over the single threaded CPU version and 4.2x speedup over the OpenBLAS version.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; G.1.6 [**Numerical Analysis**]: Optimization—*Quadratic programming methods*

General Terms

Algorithms, Performance

Keywords

GPU, Convex Optimization, Model Predictive Control, Tegra X1

*This work was done during the internship at MERL.

1. INTRODUCTION

Model predictive control (MPC) is a technique for controlling dynamical systems subject to input and state constraints. MPC has numerous advantages over traditional control techniques such as proportional integral derivative (PID) control. Primary among these are that MPC can handle multiple inputs and outputs, it enforces constraints, and it can provide improved control performance. However, these advantages come at the cost of high computational complexity.

MPC was initially developed for control of large scale industrial processes such as chemical plants. These applications are characterized by hundreds or more inputs and outputs and slow dynamics. Over the course of the last 20 years, as microprocessor performance has improved, MPC has been applied to a growing number of embedded applications such as automotive, aerospace, factory automation, and robotics [1, 2, 3]. These applications are characterized by relatively small numbers of inputs and outputs but fast dynamics and high sensitivity to controller cost. Whereas large plant controllers may need to compute a new set of outputs every few minutes or every hour, embedded MPC applications may need to compute a new output in milliseconds or microseconds. The need to solve a mathematically complex control problem at such high rates presents a barrier to the deployment of MPC in many applications. Embedded GPU computing is one potential means of overcoming this obstacle.

In this work we are interested in MPC formulations using linear prediction models with convex quadratic performance objectives subject to polyhedral constraints. These types of problems can be formulated as a constrained quadratic program (QP) [4]. Many powerful methods exist for solving QPs including active set methods [5], interior point methods [4], and gradient-based iterative methods [6, 7, 8]. We study two gradient based methods: Parallel Quadratic Programming (PQP) and Alternating Direction Method of Multipliers (ADMM). First, we create single threaded versions of the two algorithms for the ARM A57 CPU in the TX1. Next, we create CPU and GPU versions that leverage the openBLAS and NVIDIA cuBLAS basic linear algebra subroutine (BLAS) libraries. We then use a number of techniques to create GPU implementations that outperform the cuBLAS versions. Finally, we demonstrate the performance of MPC using our optimized implementations of PQP and ADMM on the Tegra X1 GPU.

The major contributions of this paper include:

- Analysis of the performance of two state of the art BLAS libraries.
- Efficient SGEMV kernels for small matrix sizes that achieve 2.3x speedup over *cuBLAS* on TX1.
- Investigation several kernel fusion schemes, including concurrent kernel execution and zero copy for both solvers.

- We demonstrate that for problem sizes greater than approximately 300 variables, GPU accelerated MPC achieves a higher control frequency than CPU implementations.

This remainder of the paper is organized as follows. In Section 2, we introduce Model Predictive Control, the PQP and ADMM algorithms, and GPU computing. In Section 3, we present the experimental systems and strategies. Performance optimization is discussed in Section 4. Related work is presented in Section 5. We present our conclusions in Section 6.

2. BACKGROUND

In this section we will briefly describe MPC, the PQP and ADMM algorithms, as well as GPU computing.

2.1 Model Predictive Control

Linear MPC is based on the prediction model

$$x_{k+1} = Ax_k + Bu_k \tag{1a}$$
$$y_k = Cx_k + Du_k, \tag{1b}$$

where $x \in \mathbb{R}^n$ is the state vector, $u \in \mathbb{R}^m$, is the input vector, $y \in \mathbb{R}^p$ is the output vector, $A \in \mathbb{R}^n \times \mathbb{R}^n$ is the state matrix, $B \in \mathbb{R}^n \times \mathbb{R}^m$ is the input matrix, $C \in \mathbb{R}^p \times \mathbb{R}^n$ is the state constraint matrix, $D \in \mathbb{R}^p \times \mathbb{R}^n$ is the input constraint matrix, subject to constraints

$$x_{\min} \leq x_k \leq x_{\max}, \tag{2a}$$
$$u_{\min} \leq u_k \leq u_{\max}, \tag{2b}$$
$$y_{\min} \leq y_k \leq y_{\max}, \tag{2c}$$

where $x_{\min}, x_{\max} \in \mathbb{R}^n$, $u_{\min}, u_{\max} \in \mathbb{R}^m$, and $y_{\min}, y_{\max} \in \mathbb{R}^p$ are lower and upper bounds on state, input, and output vectors, respectively. At step $t \in \mathbb{Z}_{0+}$, MPC solves the finite-horizon optimal control problem

$$\min_{U_t} \quad x'_{N|t}Px_{N|t} + \sum_{k=0}^{N-1} z'_{k|t}Qz_{k|t} + u'_{k|t}Ru_{k|t} \tag{3a}$$
$$\text{s.t.} \quad x_{k+1|t} = Ax_{k|t} + Bu_{k|t} \tag{3b}$$
$$y_{k|t} = Cx_{k|t} + Du_{k|t} \tag{3c}$$
$$z_{k|t} = Ex_{k|t} \tag{3d}$$
$$y_{min} \leq y_{k|t} \leq y_{max},\ k \in \mathbb{Z}_{[N_0,N_{cy}]} \tag{3e}$$
$$x_{min} \leq x_{k|t} \leq x_{max},\ k \in \mathbb{Z}_{[1,N_{cy}]} \tag{3f}$$
$$u_{min} \leq u_{k|t} \leq u_{max},\ k \in \mathbb{Z}_{[0,N_{cu}-1]} \tag{3g}$$
$$u_{k|t} = K_N x_{k|t},\ k \in \mathbb{Z}_{[N_u,N-1]} \tag{3h}$$
$$S_N x_{N|t} \leq T_N \tag{3i}$$
$$x_{0|t} = x(t), \tag{3j}$$

where $N \in \mathbb{Z}_{0+}$ is the prediction horizon, $z \in \mathbb{R}^q$ is the performance output vector, $E \in \mathbb{R}^q \times \mathbb{R}^n$ is the performance matrix,

$Q \in \mathbb{R}^q \times \mathbb{R}^q$, $Q \geq 0$ is the performance cost weight, $R \in \mathbb{R}^m \times \mathbb{R}^m$, $R > 0$ is the input cost weight, $P \in \mathbb{R}^n \times \mathbb{R}^n$, $P \geq 0$ is the terminal state cost weight, $K_N \in \mathbb{R}^m \times \mathbb{R}^n$, is the terminal controller, $S_N \in \mathbb{R}^{c_N} \times \mathbb{R}^n$, is the terminal constraint matrix, $T_N \in \mathbb{R}^{c_N}$, is the terminal constraint vector, $N_0 \in \{0, 1\}$ is the beginning of the constraint horizon, $N_u \in \mathbb{Z}_{[0,N-1]}$ is the control horizon, $N_c \in \mathbb{Z}_{[N_0,N-1]}$ is the constraint horizon, and $N_{cu} \in \mathbb{Z}_{[0,N-1]}$ is the input constraint horizon.

2.2 Parallel Quadratic Programming

PQP is an iterative, gradient based algorithm for solving convex optimization problems with quadratic objective functions. Given x_t, (3) is formulated as the QP

$$\min_U \quad J_p(U) = \frac{1}{2}U'Q_pU + F'_pU + \frac{1}{2}M_p \tag{4a}$$
$$\text{s.t.} \quad G_pU \leq K_p, \tag{4b}$$

where $U = U_k$, $Q_p \in \mathbb{R}^{n_u \times n_u}$, $n_u = Nm$, $Q_p > 0$, $G_p \in \mathbb{R}^{n_q \times n_u}$, $K_p \in \mathbb{R}^{n_q}$, $M_p \in \mathbb{R}_{0+}$.

In order to solve 4, the PQP algorithm requires that it be converted into dual form. The dual problem of (4) is

$$\min_Y \quad J_d(Y) = \frac{1}{2}Y'Q_dY + F'_dY + \frac{1}{2}M_d \tag{5a}$$
$$\text{s.t.} \quad Y \geq 0, \tag{5b}$$

where $Q_d = G_pQ_p^{-1}G'_p$, $F_d = (K_p + G_pQ_p^{-1}F_p)$, and $Y \in \mathbb{R}^{n_q}$, i.e., the number of variables in (5) is equal to the number of constraints in (4).

The actual mathematical basis of the algorithm is beyond the scope of this paper. For a detailed treatment please see [9]. Henceforth we concern ourselves primarily with the types of matrix operations that occur at each stage of the PQP algorithm and the number of FLOPs required to perform those operations as a function of the problem dimensions.

The PQP algorithm, shown in Figure 1, is separated into two parts: initialization and iteration. In later sections we will refer to these as *PQP-init* and *PQP-iter* respectively. At every sample time k, the problem matrices are initialized using a combination of matrix-matrix, matrix-vector, and inner products. Overall, the initialization step requires $2n_u^2n_q + 2n_q^2n_u + n_u^2 + 7n_qn_u - 1$ FLOPs, where n_u is the number of variables in the primal QP (4), and n_q is the number of variable in the dual QP (5).

The iterative part of the PQP algorithm is itself divided into two parts: update and acceleration. The update step is the normal execution path during iteration. The update step iteratively improves the solution, Y, to the dual QP (5), in the direction of the optimal solution Y^* according to

$$[Y_{(k+1)}]_i = \frac{[(Q_d^- + \phi)Y_{(k)} + F_d^-]_i}{[(Q_d^+ + \phi)Y_{(k)} + F_d^+]_i}[Y_{(k)}]_i \tag{6}$$

(6) is composed of two matrix vector products, two vector sums, an element-wise division, and an element-wise multiplication, for a total of $2n_un_q + 5n_q^2 + 7n_q - n_u$ FLOPs.

The purpose of the acceleration step is to improve the rate of convergence when certain numerical conditions arise. The acceleration rate, or what percentage of PQP iterations use acceleration, is problem dependent but usually falls between 0% and 10%. The number of FLOPs required varies depending on the numerical condition of the problem but in the worst case it is $4n_q^2 + 6n_q - 1$.

When the solution Y has converged to within some threshold of the global optimum Y^*, the algorithm terminates. Testing for this convergence requires $2n_qn_u + 4n_q^2 + 8n_q$ FLOPs in the worst case. In practice, testing for convergence after every iteration is wasteful. In some implementations convergence is only tested every p iterations, where p can be as high as 100 or even 1000 depending on the problem, and in other implementations the algorithm is always allowed to run for the same fixed number of iterations, such that the convergence only needs to checked once, after the final iteration.

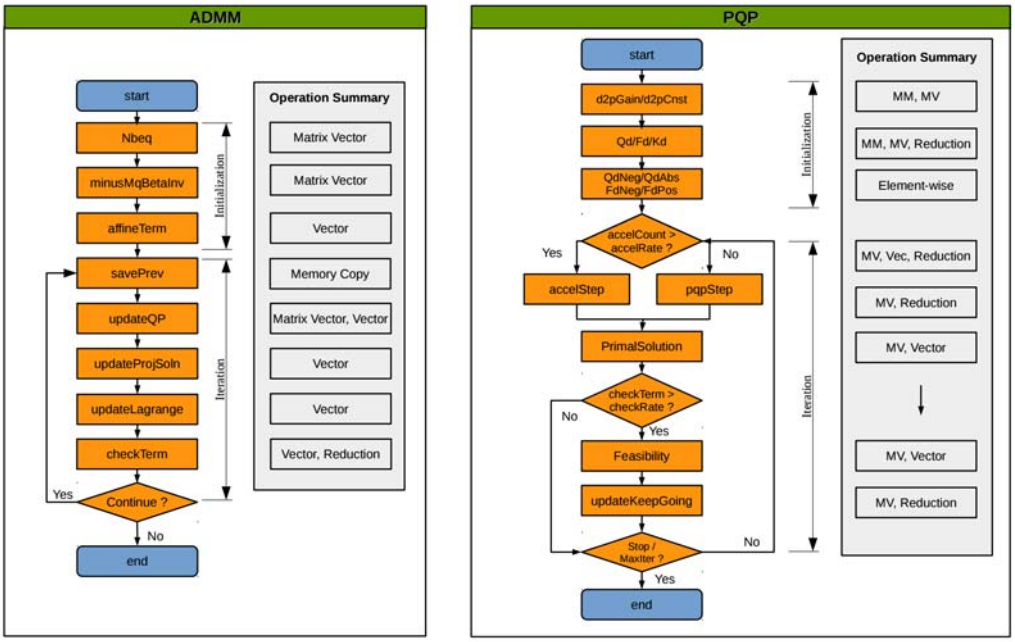

Figure 1: Block diagram of the *ADMM* and *PQP* algorithms, showing the linear algebra operations that occur at each step. MM, MV, and Vector stand for Matrix-Matrix multiplication, Matrix-Vector multiplication and Vector operations (such as addition and division), respectively.

2.3 Alternating Direction Method of Multipliers

ADMM [10] is another iterative algorithm for solving convex optimization problems with quadratic objective functions. For ADMM, the finite horizon optimal control problem in (3) is formulated as the following QP:

$$\min_{\xi} \quad J_p(\xi) = \frac{1}{2}\xi' Q_p \xi \tag{7a}$$
$$\text{s.t.} \quad G_p\xi = K_p \tag{7b}$$
$$\underline{\xi} \le \xi \le \overline{\xi} \tag{7c}$$

where $\xi = [X_t' \; U_t' \; Y_t' \; S_t']'$, $X_t = [x_{1|t}' \dots x_{N|t}']'$, $Y_t = [y_{0|t}' \dots y_{N-1|t}']'$, $S_t = Sx_{N|t}$, so that $\xi \in \mathbb{R}^{n_u}$ now with $n_u = N(m+p+n)$, and $G_p \in \mathbb{R}^{n_q \times n_u}$, $K_p \in \mathbb{R}^{n_q}$, with n_q usually different from that of (4). In order to simplify the problem, a "copy" ζ of the optimization vector ξ is used to enforce bound constraints

$$\min_{\xi,\zeta} \quad J_p(\xi) = \frac{1}{2}\xi' Q_p \xi \tag{8a}$$
$$\text{s.t.} \quad G_p\xi = K_p \tag{8b}$$
$$\underline{\xi} \le \zeta \le \overline{\xi} \tag{8c}$$
$$\zeta = \xi \tag{8d}$$

The equality constraint is dualized by the augmented Lagrangian form with a vector of Lagrange multipliers $\lambda \in \mathbb{R}^{n_u}$,

$$\min_{\xi,\zeta} \quad J_p(\xi) = \frac{1}{2}\xi' Q_p \xi + \frac{\beta}{2}\|\xi - \zeta - \lambda\|^2 \tag{9a}$$
$$\text{s.t.} \quad G_p\xi = K_p \tag{9b}$$
$$\underline{\xi} \le \zeta \le \overline{\xi} \tag{9c}$$

where β is a stepsize parameter. The ADMM the algorithm iteratively adjusts λ to seek the values of ξ, ζ that solve (9) and such that at optimum $\zeta = \xi$. It was shown in [11, 12] that this corresponds to evaluating the iterations

$$\xi^{(k+1)} = \mathcal{M}(\zeta^{(k)} + \lambda^{(k)}) + \mathcal{N}K_E \tag{10a}$$
$$\zeta^{(k+1)} = \text{proj}_{\zeta \in [\underline{\xi}, \overline{\xi}]} \quad \xi^{(k+1)} - \lambda^{(k)} \tag{10b}$$
$$\lambda^{(k+1)} = \lambda^{(k)} + \zeta^{(k+1)} - \xi^{(k+1)} \tag{10c}$$

where $\mathcal{M}, \mathcal{N}$ are matrices computed from the matrices in (9), and proj denotes the projection, i.e, in this case clipping within the box determined by $\underline{\xi}, \overline{\xi}$. The iterations in (10) update the values of ξ, ζ, and λ, respectively, while keeping the remaining variables fixed.

A flowchart for the ADMM algorithm is presented in Figure 1 along with the matrix operations that occur at each step. The ADMM algorithm is divided into an initialization phase and an iteration phase. In later sections we will refer to these as *ADMM-init* and *ADMM-iter* respectively. The initialization phase occurs once per call to the solver. The initialization computations require two matrix vector products and several vector and scalar operations for a total of $2n_u^2 + 2n_u n_q$ FLOPs. The iteration part of the algorithm consists of a single matrix vector product and several vector operations for a total of $2n_u^2 + 5n_u$ FLOPs. Similar to PQP, in order to save processor cycles, the ADMM algorithm does not check for convergence every iteration. Rather, the algorithm checks for convergence every p iterations, where p is problem dependant. The convergence check itself requires only vector and reduction operations totalling $9n_u + 1$ FLOPs.

2.4 GPU Computing

Modern graphics processing units (GPUs) feature hundreds or thousands of parallel processing cores that support general purpose processing as well as traditional graphics applications [13][14]. GPUs can sustain a much higher arithmetic computing throughput and streaming memory bandwidth (e.g. 8873 GFLOPs and 320 GB/sec for GP104) than their CPU counterparts (e.g., 354 GFLOPs and 68 GB/sec for Intel i7 5960X) [15]. More and more computationally intensive applications, such as molecular dynamics, deep

Table 1: CPU Specifications

CPU	Intel Core i7-4790K	64-bit ARM A57
Lithography	22 nm	16 nm
Cores	4	4
Threads	8	4
Core Clock	4.0 GHz	1.9 GHz
Memory Bandwidth	25.6 GB/s	25.6 GB/s
Cache	8 MB	2 MB
Memory Size	32 GB	4 GB
Memory Types	DDR3-1333	LPDDR4
TDP	88 W	10 W

learning, and cryptography, are being ported to GPUs and GPUs are deployed in 68 of the Top500 supercomputer systems [16].

GPUs execute programs in the *Single Instruction Multiple Thread* (SIMT) style. Hundreds or thousands of threads can be launched and concurrently executed. These parallel threads are organized into grids of blocks that get scheduled to the streaming multiprocessors in the device[1]. Each block executes instructions in groups of 32 threads, also known as *warps*, in the *SIMD* fashion. A warp scheduler dispatches two independent instructions to increase the *instruction level parallelism*. The high latency of memory loads and stores can be tolerated by warps running independent arithmetic operations. A brief overview of NVIDIA's Maxwell architecture is shown in Figure 2.

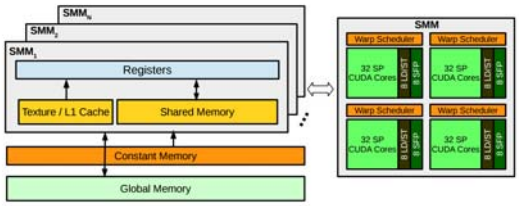

Figure 2: An overview of NVIDIA's Maxwell architecture.

3. METHODOLOGY

3.1 Experimental Systems

In this work we utilize two seperate experimental setups. The first is an Intel Core i7 4790K with an NVIDIA GTX 970 graphics card. The second is the NVIDIA Jetson TX1 development kit which includes the Tegra X1 mobile processor. The Tegra X1 processor includes 4 ARM Cortex A57 cores, 4 ARM Cortex A53 cores, and a 256 core Maxwell GPU. We use the ARM A57 CPUs inside the X1 as a reference processor that we will benchmark our GPU implementations against. This choice represents a compromise. Although the A57 is a RISC machine like most processors used in embedded controllers, it has significantly higher performance than is typical in such systems. However, using the A57 as the reference processor allows us to perform all of our work using a common platform, toolchain and code base. Although we are primarily interested in the performance of ADMM and PQP on the TX1 CPU and GPU, we make use of the Intel i7 / GTX970 system as well. This is because we want to explore the performance differences between the Maxwell GPU in the GTX 970 and the Maxwell GPU in the TX1. All of the development platforms run Ubuntu 14.04 and CUDA 8.0. The detailed specifications for these platforms are shown in Tables 1 and 2.

3.2 Implementation Strategy

Our baseline for evaluating the performance of the PQP and ADMM algorithms is a single threaded implementation for the ARM

[1]The CUDA terminology is applied since NVIDIA GPUs are used for the work.

Table 2: GPU Specifications

GPU	GeForce GTX 970	NVIDIA Tegra X1
Architecture	Maxwell	Maxwell
CUDA Cores	1664	256
Multiprocessors	13	2
CUDA Cores / SP	128	128
Core Clock	1317 MHz	1000 MHz
Memory Clock	3505 MHz	1600 MHz
Memory Bus Width	256-bit	64-bit
Global Memory	4095 MB	3994 MB
L2 Cache	1792 KB	256 KB
Constant Memory	64 KB	64 KB
Shared Memory per Block	48 KB	48 KB
Register per Block	64 K	32 K
copy engines	2	1
Integrated GPU sharing Host Memory	No	Yes

A57 core of the TX1. We will refer to this version as *cpu-1*. We utilize g++ 4.8.4 with the $-O3$ and $-march = native - mcpu = native$ compiler flags set to turn on optimization. To investigate how well the algorithms mapped to a traditional parallel programming paradigm we created a 4 threaded version for the A57 using GNU OpenMP, which we refer to as *cpu-4*. Because we know that both of these algorithms are fundamentally composed of elementary linear algebra operations, it seems natural to begin our performance optimization by evaluating some standard BLAS libraries. We create BLAS optimized versions for both the CPU and GPU because comparing the performance of a BLAS optimized GPU version to that of a non-BLAS CPU version would not be informative. For the ARM CPU we evaluate ATLAS and openBLAS and then choose the one with the best performance for our application and architecture. This version is referred to as *cpu-openblas* We then implement initial versions of the algorithms for the TX1 GPU using Nividia's cuBLAS library. This version is referred to as *gpu-cublas*. Finally, we tested a variety of custom CUDA kernels in order to find the best method for achieving optimal performance for each algorithm. The final version of each algorithm is referred to as *mpcGPU*.

4. PERFORMANCE OPTIMIZATIONS

4.1 Baseline

For the CPU implementation, we compare the performance of single-threaded and four-threaded versions. Four threads can leverage all the physical CPU cores available. As shown in Figure 1, many operations for the solvers can be expressed in linear algebra subroutines. To leverage the full potential of the CPU, we optimize the implementations with BLAS libraries on the CPU. The *ATLAS* and *OpenBLAS* are two of the most popular CPU BLAS libraries [17][18].

- The *ATLAS* is a BLAS library which can be tuned automatically at the compilation time for the target CPU architecture. It provides interfaces to both BLAS and *LAPACK* routines [19].
- The *OpenBLAS* library supports optimized linear algebra kernels on a variety of CPU architectures such as *MIPS64*, *SPARC*, *POWER*, *x86/x86-64*, *ARM*.

We utilize *ATLAS-3.8.4-arm* becuase it supports ARM architectures. The performance of *ATLAS* and *OpenBLAS* on the ARM A57 is illustrated in Figure 3. On the ARM CPU, for *SGEMV*, *ATLAS* is faster than *OpenBLAS* when the matrix is smaller than 300x300. For *SGEMM*, *OpenBLAS* outperforms *ATLAS* for all the test cases. On average, *OpenBLAS* is 1.2x and 2.9x faster than *ATLAS* for *SGEMV* and *SGEMM*, respectively.

For the GPU implementation, we compare two widely used GPU BLAS libraries, *cuBLAS* [20] and *MAGMA*(Ver 2.1) [21]. Their performance on *SGEMV* and *SGEMM* are plotted in Figure 4. On

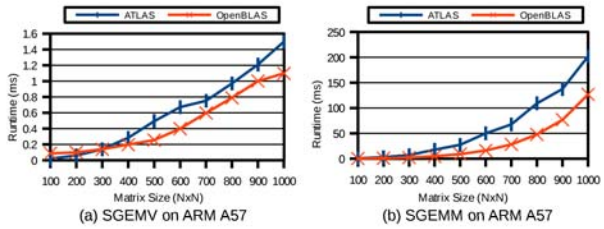

Figure 3: Performance of the *ATLAS* and *OpenBLAS* libraries on ARM A57. The SGEMV uses *CblasRowMajor* and *CblasTrans*. The SGEMM uses *CblasRowMajor* and *CblasNoTrans*. Smaller the runtime indicates better performance.

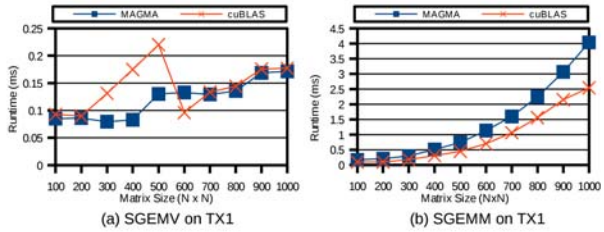

Figure 4: Performance of GPU BLAS libraries, *MAGMA* and *cuBLAS* on TX1. No transposition is applied to the column major matrices. Smaller the runtime indicates better performance.

the TX1, *MAGMA* achieves 1.2x better performance, on average, than *cuBLAS* for *SGEMV*. On *SGEMM*, *cuBLAS* is 1.6x better than *MAGMA* on average.

We selected *OpenBLAS* becuase it consistantly outperformed *ATLAS* in our testing. The results for the GPU were not as clear cut. We ultimately selected *cuBLAS* because of its superiour performance on the SGEMM operation. The performance of our baseline implementations on the TX1 is illustrated in Figure 5 for a 500x500 matrix.

4.1.1 Baseline ADMM Performance

On the CPU, running four threads (*cpu-4*) only results in a 30% improvement over the single-threaded version (*cpu-1*) during the *ADMM-Init* step, whereas no performance benefit is achieved for the *ADMM-Iter*. Since the input matrix (1MB) fits into the CPU cache, running more threads doesn't improve performance beause of the fork-and-join overhead. After adopting *OpenBLAS*, more than 2.6x speedup is obtained for the preparation and prediction step, relative to the single threaded CPU version.

On the GPU, the *cuBLAS* version (*gpu-cublas*) performs slightly better than *cpu-1* with 1.2x improvement for *ADMM-Init*. It performs worse than *cpu-1* for *ADMM-Iter*. There are three important factors contributing to the poor performance:

- Not every step in the ADMM algorithm can be mapped to a *BLAS* subroutine.
- Breaking down the computation into several kernel calls introduces more kernel launch overhead.
- Data movement is expensive the between CPU and GPU, especially for small data sizes.

For instance, Equation (11) is used for the *updateQp* during the iterative computation of ADMM.

$$M * (w\_prev - lam\_prev * betaInv) + affineTerm \quad (11)$$

where M is a square matrix, w_prev, lam_prev and $affineTerm$ are vectors, and $betaInv$ is a scalar. To implement this equation in *cuBLAS*, four kernels calls are needed, including two *cublasScopy*, one *cublasSaxpy* and one *cublasSgemv*.

4.1.2 Baseline PQP Performance

On the CPU, *cpu-4* runs 1.9x faster than *cpu-1* for *PQP-Init* and 1.6x for *PQP-Iter*. The *OpenBLAS* implementation improves the initialization steps by 34.5x and the iteration steps by 1.9x. Matrix multiplication is the most intensive computation for *PQP-Init*. On the GPU, the *cuBLAS* version of *PQP-Init* has achieved 248.2x and 7.2x speedup over *cpu-1* and *cpu-openblas*, respectively. This is due to the highly efficient *SGEMM* implementation on the GPU. However, during the iteration step which is less computationally intensive, the improvement after using *cuBLAS* is less than 10%.

4.1.3 Summary

As Figure 5 illustrates, the *Nbeq* step in *ADMM-Init* and the *updateQP* in *ADMM-Iter* are the largest contributors to overall runtime in the *ADMM* algorithm. For *PQP*, the matrix multiplication required to compute *d2pGain* and *Qd* are the largest contributors to runtime, while the *pqp* and *accel* steps limit the performance during the iteration steps.

4.2 Efficient SGEMV on TX1

Both *ADMM* and *PQP* use the iterative scheme to make prediction. The most intensive routine inside the loop is matrix-vector multiplication. Therefore, we focused most of our effort on developing a more efficient SGEMV kernel that can outperform the *cuBLAS* library on the TX1 GPU.

4.2.1 Implementation

We propose three different implementations for matrix-vector multiplication, *2d-bs128*, *2d-bs1024*, and *1d-bs1024*. The first two versions use a 2D grid for the kernel and the third uses a 1D grid.

For *2d-bs128*, the block dimension is set as (32, 4, 1), where the x dimension, mapped to the columns of the input matrix, uses 32 threads (the size of a warp) and the y dimension, mapped to the rows of the input matrix, uses 4 threads. In this case, the total number of threads per block, 128, matches the number of CUDA cores per streaming multiprocessor for the NVIDIA Maxwell GPU architecture (see Figure 2). To increase the instruction level parallelism, each warp quadruples the work along the x dimension and double the work along the y dimension. Hence, we have a batch size of 128 along the column and 8 along the row of the input matrix, as illustrated in Figure 6 (a).

For *2d-bs1024*, the block dimension is set as (1024, 1, 1), which means launching a 1024-thread block for each row of the input matrix. Each block is responsible for the consecutive 8 rows, as shown in Figure 6 (b). In this case, the workload for each thread has been increased 8x. This technique has been used previously for tuning the dense linear algebra kernels [22].

For *1d-bs1024*, a block of 1024 threads is launched to execute the entire *SGEMV* computation. These threads are grouped as a 32x32 2D grid, as presented in Figure 6 (c). After thread-mapping, each warp is responsible for one row of the input matrix and iterates along the column. These 32 warps of the block marshal along the row of the input matrix, in a batch size of 32.

In these implementations, the *SHFL* instructions are used to exchange data within the warp [23]. They can reduce the demand on shared memory and limit the usage of local barriers without neutralizing the performance. The read-only cache is enabled for global memory access. Using the constant memory or texture memory slows down the performance since moving data to the large cache is expensive for small data sizes.

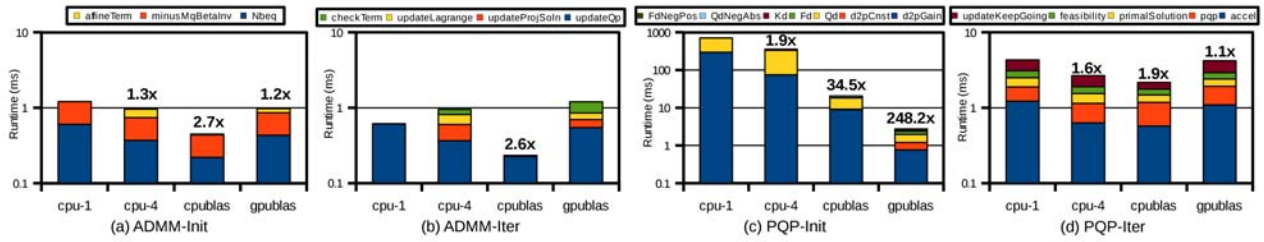

Figure 5: The runtime for baseline implementations, *cpu-1, cpu-4, cpublas (OpenBLAS), gpublas (cuBLAS)* on Jetson TX1. The speedup over single-threaded (*cpu-1*) version is shown in bold text. The y-axis is in logarithmic scale.

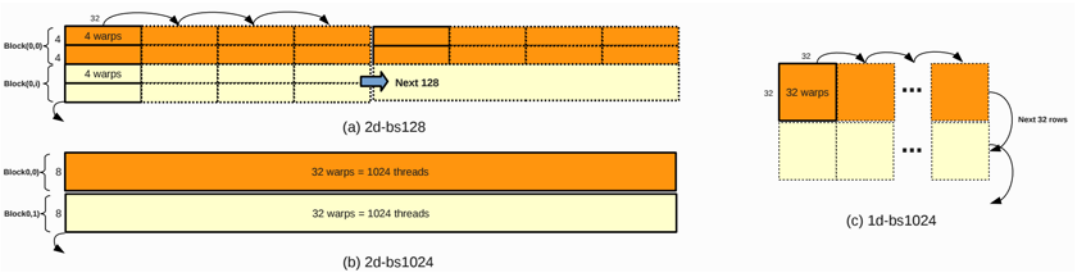

Figure 6: The implementation diagrams for the *mpcGpu* SGEMV kernels.

4.2.2 *Analysis*

To help with tuning our kernels, four performance counters are extracted using the NVIDIA's command line profiler *nvprof*. These performance counters include occupancy, instructions per cycle (ipc), global memory load throughput (gld_throughput) and global memory store throughput (gst_throughput) [24]. Two sampling points (N=500, 1000) are studied since the *cuBLAS* switches kernels depending on the data size. The statistics are illustrated in Figure 7. We noticed that *cuBLAS* achieves better performance with a lower occupancy level. The $2d{-}bs128$ always achieves a higher occupancy level than the other three. Although the *ipc* for the $2d{-}bs1024$ is 4.5x higher than the $2d{-}bs128$, its throughput is 5% less than $2d{-}bs128$ on average. In the authors' opinion, *ipc* is not a good metric for evaluating the GPU performance. Since *SGEMV* is a memory-bound kernel, its performance is tightly correlated to the throughput of the global memory. The combined throughput, $gld\_throughput$ and $gst\_throughput$, of $2d{-}bs128$ is the highest for N=500. For N=1000, the combine throughput of $2d{-}bs2014$ is the highest among all the kernels.

In addition to the throughput-oriented metrics, we investigated the top-3 stalling reasons for these implementations (see Table 3). Increasing the workload per thread in *cuBLAS* introduces more stalls due to the instruction latency, as indicated by $stall\_exe$. With a larger matrix, the primary stalling reason for $2d{-}bs1024$ changes from local synchronization ($stall\_sync$) to memory latency ($stall\_mem$). The performance of $1d{-}bs1024$ heavily relies on memory latency. For $2d{-}bs128$, the stalling reasons are evenly balanced among the different factors.

As shown in Figure 7, *cuBLAS* boosts the *SGEMV* throughput from 1.14 to 3.76 Gflop/s, when N increases from 500 to 600. This is due to a more efficient high-performance, low-occupancy implementation. It reaches a peak of 5.65 Glop/s when N=1000. For the *cuBLAS* SGEMV kernel, no transposition is applied to the input matrix. When the row number N is no larger than 900, the $2d{-}bs128$ achieves the best performance, with an average of 2.3x speedup compared to *cuBLAS*. $2d{-}bs1024$ outperforms $2d{-}bs128$ afterwards, by a margin of 7%. $2d{-}bs1024$ outperforms *cuBLAS* by 2.0x on average. For N=1000, the $2d{-}bs1024$ achieves 7.45 Gflop/s throughput, which is 1.3x higher than *cuBLAS*. The 1D version of matrix-vector multiplication $1d{-}bs1024$ achieves 1.9x speedup on average. Our motivation for developing a 1D *SGEMV* kernel will be explained in the next section.

Table 3: Top-3 stalling factors for the SGEMV kernels.

Matrix size 500 x 500

	cuBLAS	2d-bs128	2d-bs1024	1d-bs1024
stall_exec	**37.61%**	-	24.84%	9.49%
stall_memory	33.20%	25.58%	-	**72.44%**
stall_texture	-	31.32%	-	-
stall_sync	16.37%	-	**26.36%**	7.07%
stall_other	-	**36.72%**	20.12%	-

Matrix size 1000 x 1000

	cuBLAS	2d-bs128	2d-bs1024	1d-bs1024
stall_exec	**46.27%**	-	23.59%	8.78%
stall_memory	41.66%	26.98%	**37.48%**	**77.01%**
stall_texture	-	31.67%	-	-
stall_sync	-	-	17.93%	4.29%
stall_other	4.81%	**36.52%**	-	-

4.3 Kernel Fusion

For each step of the QP solvers, multiple arithmetic operations are performed. Some of them are called frequently in each iteration, such as matrix-vector multiplication. Depending on the convergence speed, the number of iterations can be significantly large. This is not an issue for the CPU BLAS subroutines since the calling overhead is low. However, it is a potential bottleneck for the GPU implementation. Therefore, for most of the benchmarked cases shown in Figure 5, the *cuBLAS* implementation is inferior to the *OpenBLAS* implementation on the CPU.

To reduce the kernel launch overhead, multiple BLAS routines are merged and computed within a single kernel. This technique is known as *kernel fusion*[25]. However, the data dependency between

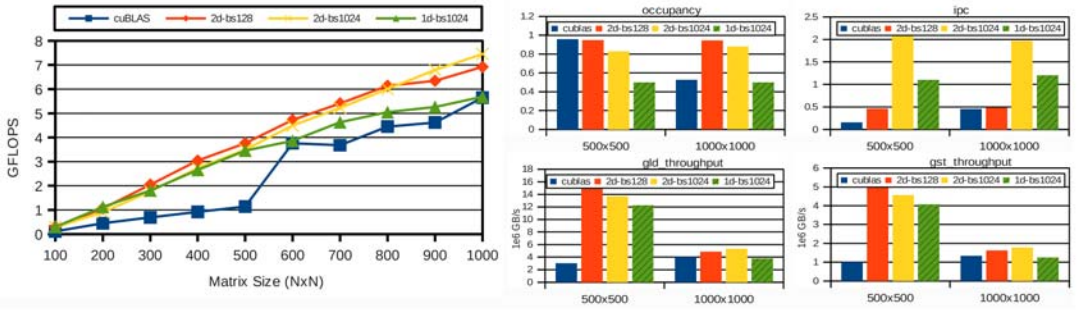

Figure 7: Comparison of the custom kernels with *cuBLAS* kernels for SGEMV on Jetson TX1.

steps will introduce the need for explicit or implicit thread synchronization. One possible solution is using a local barrier instead of global synchronization. In this case, one single block is launched to execute the fused computation. The aforementioned 1D SGEMV kernel is applied under such circumstances. For instance, the dot product of two vectors after matrix-vector multiplication can be merged into a single kernel call. Ideally, all the subroutines can be fused in such a way that the data movement and memory footprint is significantly reduced. In Table 4, we show the number CUDA API calls before and after applying *kernel fusion*. The API calls include the BLAS routines, memory copy, and custom kernels.

Table 4: The CUDA API calls before and after fusing kernels.

	cuBLAS	**mpcGpu**
ADMM-Init	4	1
ADMM-Iter	16	1
PQP-Init	9	9
PQP-Iter	24	1

4.3.1 ADMM Performance Optimization

For the *ADMM* solver, both the initialization and iteration steps can be merged into one GPU kernel. For *ADMM-Init*, since there are no parallel reduction operations, the 2D SGEMV kernels are applied for better performance. For *ADMM-Iter*, multiple schemes have been investigated.

- *K1*: 1D SGEMV kernel.
- *K2*: 2D SGEMV kernel + parallel reduction + data transfer
- *K3*: *K2* using persistent thread block (ptb) method [26]
- *K4*: *K1* with loops inside the kernel

We consider *K1* a *greedy-fusion* scheme since it aggressively merges all the steps into a single kernel. *K2* uses the *adaptive-fusion* scheme where separate kernels are called for different types of operations. In Table 5, a real world example with a matrix size of 325x325 is benchmarked. There are a total of 2282 iterations. By separating the reduction operation from other kernel calls, *K2* achieves the lowest combined average runtime per call at *36.2us*. The persistent thread block method in *K3* did not have much effect. Launching 1 monolithic block in *K1* attains 59.9% occupancy compared to 90.0% for *K2*. Running the iteration loop inside the kernel slows down the performance significantly for *K4*. Compared to *K1*, *K4* brings about 5.3x instructions per warp, 4.9x executed control-flow instructions and 12.4x miscellaneous instructions executed by non-predicated threads. Based on these observations, the *adaptive-fusion* scheme used in *K2*, appears to be the most effective.

Table 5: Profiling summary for ADMM iteration.

	Kernel Type	Calls	Time(%)	Avg / Call
K1	1d sgemv	2282	98.9%	83.8us
K2	2d sgemv	2282	88.1%	32.8us
	reduction	2282	9.2%	3.4us
K3	2d + ptb	2282	88.5%	34.1us
	reduction	2282	9.0%	3.5us
K4	1d + loop	1	100.0%	8.4s

4.3.2 PQP Performance Optimization

The performance of the initialization stage of the *PQP* solver is dominated by the *SGEMM* routine. *cuBLAS* provides significant performance improvement over the CPU versions, as shown in Figure 5 (c). For *PQP-Init*, *cuBLAS* is applied to leverage the GPU computing power on TX1.

There are seven computing steps during the initialization stage of *PQP* (see Figure 1). These steps are not all data-dependent. We can take advantage of *Concurrent Kernel Execution* (CKE) to parallelize the computation of the independent steps [27, 28, 29]. After adopting CKE using CUDA streams, we observed 2-15% improvement on GTX 970, as illustrated in Figure 8 (a). On TX1, however, 2-37% performance degradation is observed in Figure 8 (b). Concurrent kernel execution performance depends on the availability of device resources. The TX1 has 2 streaming multiprocessors compared to 13 on the GTX 970. The launched CUDA streams are serialized on TX1 with a significant launching overhead. Therefore, we use the default stream for all the *PQP-Init* steps.

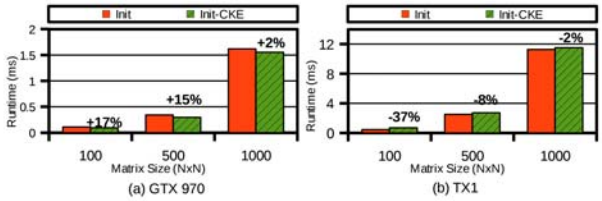

Figure 8: Concurrent kernel execution performance on GTX 970 and TX1 for *PQP-Init*.

For the iteration step of the *PQP* solver, there are four different execution paths as listed below.

- *PQP-Iter-Accel*: Run the acceleration step.
- *PQP-Iter-Accel-CheckTerm*: Run the acceleration step and check termination.
- *PQP-Iter-PQP*: Run the pqp step.

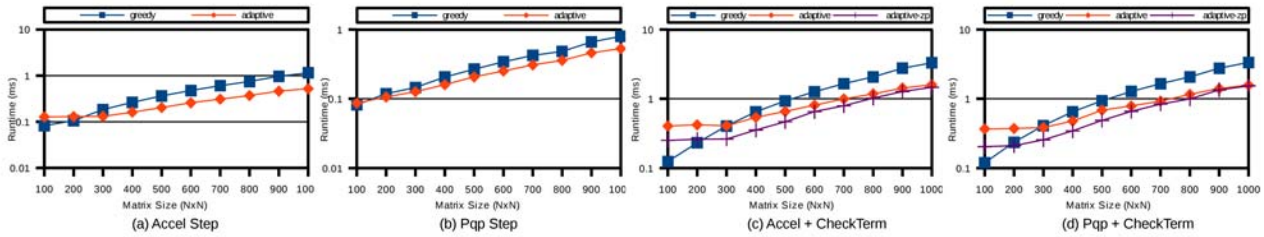

Figure 9: Performance of the *greedy-fusion* and *adaptive-fusion* scheme for *PQP-iter*. Zero Copy is applied to the *adaptive-fusion* during the checkTerm step.

- *PQP-Iter-PQP-CheckTerm*: Run the pqp step and check termination.

The computation for each execution path can be merged into one single kernel call. The CUDA memory copy to update the predicted outcome is avoided by switching the data buffers between two consecutive iterations. On the TX1, *Zero Copy* [30] is supported with the integrated GPU sharing the host memory (see Table 2). Leveraging this feature, we can reduce the data communication in the checkTerm step.

In Figure 9, we present the results after adopting the *greedy-fusion* and *adaptive-fusion* schemes for these execution paths. Iteration inside the kernel is avoided due to the significant performance impact. For *PQP-Iter-Accel* in Figure 9(a), merging all the steps used by the *greedy-fusion* scheme shows 50% better performance than the *adaptive-fusion* scheme when N=100. As the matrix size increases, *adaptive-fusion* achieves better performance than *greedy-fusion*. On average, a speedup of 1.6x and 1.3x is obtained by adopting *adaptive-fusion* for the *Accel* and *Pqp* step, respectively.

Zero Copy is adopted during the checkTerm step, as shown in Figure 9 (c) (d). For *PQP-Iter-Accel-CheckTerm*, the *adaptive-fusion* scheme with Zero Copy achieves an average 1.7x speedup over the *greedy-fusion* scheme. The benefit of the *Zero Copy* decreases as the matrix size increases. For *PQP-Iter-PQP-CheckTerm*, an average of 1.7x speedup over the *greedy-fusion* scheme is also observed by *adaptive-zp* (adaptive-fusion with zero copy).

4.4 Performance Consolidation

The combined performance of all optimization techniques is presented in Figure 10 in terms of runtime per iteration. For *ADMM-Init*, the *mpcGpu* exceeds the single-threaded *cpu-1* after N=100, with an average 5.8x speedup. The *cpu-openblas* is most efficient when $N\leq$200,after which *mpcGpu* is superior. On average, it achieves 2.2x and 2.4x speedup compared to *cpu-openblas* and *cuBLAS*, respectively.

As illustrated in Figure 10 (b), The CPU *OpenBLAS* implementation is the fastest when $N\leq$300. For $N\geq$300 *mpcGpu* obtains an average of 3.8x speedup over *cpu-1*, 1.4x speedup over *cpu-openblas*, and 4.6x speedup over *cuBLAS*.

For *PQP-Init*, *mpcGpu* utilizes *cuBLAS* and achieves 501.6x speedup over *cpu-1* and 19.3x speedup over *cpu-openblas* on average. For *PQP-Iter-Accel*, *mpcGpu* is 3.1x faster than *cpu-1*, 1.1x faster than *cpu-openblas* and 2.6x faster than *cuBLAS*. With the check termination step, *mpcGpu* outperforms *cpu-1*, *cpu-openblas* and *cuBLAS* by 3.7x, 1.3x and 3.0x, respectively. The performance impact on *PQP-Iter-PQP* using the 1D and 2D SGEMV kernels are shown in Figure 10 (f) and (g). The 2D version is 1.3x faster than the 1D version. For *PQP-Iter-PQP*, our implementation attains an average of 3.0x speedup over *cpu-1*, 2.1x speedup over *cpu-openblas* and 2.5x speedup over *cuBLAS*. For *PQP-Iter-PQP-CheckTerm*, our *mpcGpu* performs 2.7x better than *cpu-1*, 1.4x better than *cpu-openblas* and 2.7x better than *cuBLAS*. It is observed that, when $N>$200, the *mpcGpu* runs faster than the CPU counterparts.

Finally, we evaluated the runtime of the optimized PQP and ADMM algorithms on 10 real world MPC examples ranging in size from 70 to 980. The problem sizes are odd because it is not always possible to formulate a real MPC problem that results in a particular problem size. In order to make a fair comparison, the number of iterations was fixed at 1000 for all cases. The results are summarized in Figure 11. The OpenBLAS CPU version is superior when the problem size is smaller than about 300 for ADMM and 200 for PQP. After that, the GPU version has the shortest runtime. In the best case, for *ADMM*, *mpcGpu* achieves 46.6x speedup over the single-threaded CPU version (*cpu-1*), and 2.7x over the optimized CPU OpenBLAS version. For *PQP*, *mpcGpu* attains a maximum speedup of 41.2x relative to *cpu-1* and 4.2x relative to *openBLAS*.

Figure 12 illustrates the same data in terms of achievable MPC controller frequency. Over the range of problem sizes, *mpcGPU* achieves an update rate between 10 Hz and 1.7 Hz for PQP and 14 Hz and 3.4 Hz for ADMM.

5. RELATED WORK

Model Predictive Control is a widely applied control method for anticipating future events for a short time horizon [31]. ADMM and PQP are two popular approaches to solving the convex optimization problem at the heart of MPC [10, 32, 9]. The PQP algorithm proposed by Brand *et al.*obtained a speedup of 5-10x against the Matlab quadratic Programming solver *quadprog* on the Intel Core^{TM}2 CPU [33]. Frison and Jørgensen proved that the ARM Cortex CPU are suitable for embedded MPC [34]. A FPGA architecture with reduced precision fix-point arithmetic for ADMM is designed by Jerez *et al.* [35]. With the advent of general purpose computing on GPUs, GPU-accelerated interior point methods for convex optimization problems have been proposed by Gade-Nielsen *et al.* [36]. On the NVIDIA Tesla C2050, their GPU kernels achieve 6x speedup compared to the corresponding Matlab version running on the Intel Core i7 930 CPU. Regarding to optimizing dense matrix-vector multiplication on GPUs, KBLAS reports an 50%-60% impromentment on the state-of-the-art GPU BLAS libraries [37]. Optimizing GPU kernels for these iterative solvers is non-trivial. Several optimization principles for iterative solvers are proposed by Tarjan *et al.* [38]. Applying large kernels, memory reuse, kernel execution overlapping and persistent thread blocks, their GPU implementation achieves more than 200x speedup over Matlab for the MGVF application.

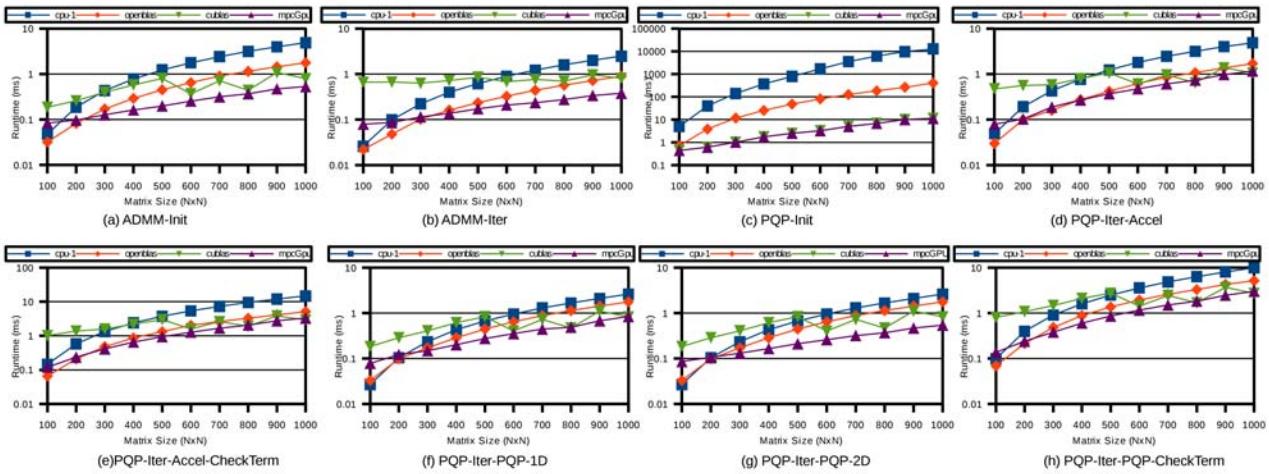

Figure 10: Performance after adopting efficient *SGEMV* and kernel fusion on Jeston TX1.

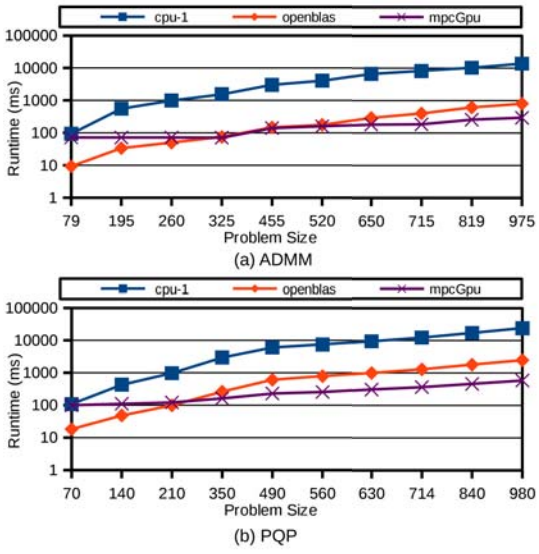

Figure 11: The performance of *mpcGpu* on Jetson TX1 for the *ADMM* and *PQP* solver. The runtime is in logarithmic scale.

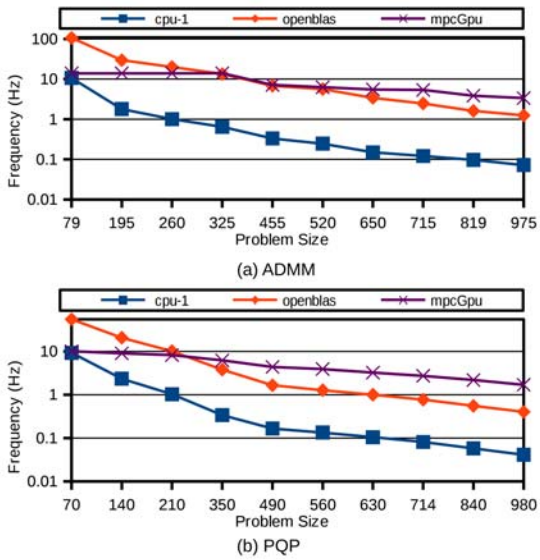

Figure 12: Achievable MPC controller frequency for the *mpcGpu* implementations of ADMM and PQP.

6. CONCLUSION AND FUTURE WORK

This work demonstrates that GPU computing can be leveraged to achieve higher performance in embedded MPC applications. We have optimized the SEGMV operation for small matrix sizes by applying a combination of advanced CUDA programming techniques.

Our GPU implementations of PQP and ADMM have achieved 41.2x-46.6x speedups relative to single threaded CPU implementations and 2.7x-4.2x speedups relative to optimized OpenBLAS implementations. As we stated in the introduction, advanced MPC applications require QP solutions in milliseconds or microseconds. Clearly, the TX1 can not achieve those update rates. However, there may be some MPC applications that are better suited to GPU acceleration. These may include systems with slower dynamics and or systems with larger problem sizes. In future work, we plan to develop a performance model to tune MPC solvers on different architectures and adaptively select the best runtime version according to the problem size. In addition, significant work on deterministic scheduling of real-time GPU applications will be required before GPUs can be leveraged in real-time systems.

7. REFERENCES

[1] S. Di Cairano, "An industry perspective on MPC in large volumes applications: Potential Benefits and Open Challenges," in *Proc. 4th IFAC Nonlinear Model Predictive Control Conference*, (Noordwijkerhout, The Netherlands), pp. 52–59, 2012.

[2] S. Di Cairano, D. Yanakiev, A. Bemporad, I. Kolmanovsky, and D. Hrovat, "Model predictive idle speed control: Design, analysis, and experimental evaluation," *IEEE Tr. Contr. Sys. Technology*, vol. 20, no. 1, pp. 84 –97, 2012.

[3] S. Di Cairano, H. Park, and I. Kolmanovsky, "Model predictive control approach for guidance of spacecraft

rendezvous and proximity maneuvering," *Int. J. Rob. Nonlinear Control*, 2012.

[4] S. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge University Press, 2004.

[5] J. Nocedal and S. Wright, *Numerical optimization*. Springer verlag, 1999.

[6] S. Richter, C. Jones, and M. Morari, "Real-time input-constrained mpc using fast gradient methods," in *Proc. 48th IEEE Conf. on Dec. and Control*, (Shangai, China), pp. 7387–7393, 2009.

[7] A. Bemporad and P. Patrinos, "Simple and certifiable quadratic programming algorithms for embedded control," in *Proc. 4th IFAC Nonlinear Model Predictive Control Conference*, (Noordwijkerhout, The Netherlands), 2012.

[8] M. Kögel and R. Findeisen, "Fast predictive control of linear systems combining Nesterov's gradient method and the method of multipliers," in *Proc. 51st IEEE Conf. on Dec. and Control*, (Orlando, FL), pp. 501–506, 2011.

[9] S. Di Cairano and M. Brand, "On a multiplicative update dual optimization algorithm for constrained linear mpc," in *52nd IEEE Conference on Decision and Control*, pp. 1696–1701, IEEE, 2013.

[10] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends® in Machine Learning*, vol. 3, no. 1, pp. 1–122, 2011.

[11] A. U. Raghunathan and S. Di Cairano, "Optimal step-size selection in alternating direction method of multipliers for convex quadratic programs and model predictive control," in *Proceedings of Symposium on Mathematical Theory of Networks and Systems*, pp. 807–814, 2014.

[12] A. U. Raghunathan and S. Di Cairano, "Infeasibility detection in alternating direction method of multipliers for convex quadratic programs," in *53rd IEEE Conference on Decision and Control*, pp. 5819–5824, IEEE, 2014.

[13] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[14] J. Nickolls and W. J. Dally, "The GPU Computing Era," *Micro, IEEE*, vol. 30, no. 2, pp. 56–69, 2010.

[15] NVIDIA, "NVIDIA GeForce GTX 1080 Whitepaper," 2016.

[16] Top500, "List of top 500 supercomputers." `https://www.top500.org/lists/2016/06/`, 2016.

[17] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pp. 1–27, IEEE Computer Society, 1998.

[18] Z. Xianyi, W. Qian, and Z. Yunquan, "Model-driven level 3 blas performance optimization on loongson 3a processor," in *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, pp. 684–691, IEEE, 2012.

[19] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen, "Lapack: A portable linear algebra library for high-performance computers," in *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pp. 2–11, IEEE Computer Society Press, 1990.

[20] NVIDIA, "Basic Linear Algebra Subroutines (cuBLAS)," 2016.

[21] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki, "Accelerating numerical dense linear algebra calculations with gpus," *Numerical Computations with GPUs*, pp. 1–26, 2014.

[22] V. Volkov, "Better performance at lower occupancy," in *Proceedings of the GPU Technology Conference, GTC*, vol. 10, 2010.

[23] J. Demouth, "Shuffle: Tips and Tricks," *GPU Technology Conference*, 2013.

[24] NVIDIA, "Command-line Profiler," 2016.

[25] J. Filipovič, M. Madzin, J. Fousek, and L. Matyska, "Optimizing cuda code by kernel fusion: application on blas," *The Journal of Supercomputing*, vol. 71, no. 10, pp. 3934–3957, 2015.

[26] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style gpu programming for gpgpu workloads," in *Innovative Parallel Computing (InPar), 2012*, pp. 1–14, IEEE, 2012.

[27] S. Rennich, "Cuda C/C++ Streams and Concurrency," *NVIDIA GPU Computing Webinars*, 2012.

[28] F. Wende, T. Steinke, and F. Cordes, "Multi-threaded Kernel Offloading to GPGPU Using Hyper-Q on Kepler Architecture," Tech. Rep. ZIB-Report 14-19, Zuse Institute Berlin, Zuse Institute Berlin, Takustrasse 7, D-14195 Berlin, Germany, Jun 2014.

[29] L. Yu, Y. Ukidave, and D. Kaeli, "GPU-accelerated HMM for Speech Recognition," in *2014 43rd International Conference on Parallel Processing Workshops*, pp. 395–402, IEEE, 2014.

[30] ArrayFire, "Zero Copy on Tegra K1." `http://arrayfire.com/zero-copy-on-tegra-k1/`.

[31] M. Morari and J. H. Lee, "Model predictive control: past, present and future," *Computers & Chemical Engineering*, vol. 23, no. 4, pp. 667–682, 1999.

[32] M. Annergren, A. Hansson, and B. Wahlberg, "An admm algorithm for solving âĎŞ 1 regularized mpc," in *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*, pp. 4486–4491, IEEE, 2012.

[33] M. Brand, V. Shilpiekandula, C. Yao, S. A. Bortoff, T. Nishiyama, S. Yoshikawa, and T. Iwasaki, "A parallel quadratic programming algorithm for model predictive control," *IFAC Proceedings Volumes*, vol. 44, no. 1, pp. 1031–1039, 2011.

[34] G. Frison and J. B. Jørgensen, "Mpc related computational capabilities of armv7a processors," in *Control Conference (ECC), 2015 European*, pp. 3414–3421, IEEE, 2015.

[35] J. L. Jerez, P. J. Goulart, S. Richter, G. A. Constantinides, E. C. Kerrigan, and M. Morari, "Embedded online optimization for model predictive control at megahertz rates," *IEEE Transactions on Automatic Control*, vol. 59, no. 12, pp. 3238–3251, 2014.

[36] N. F. Gade-Nielsen, J. B. Jørgensen, and B. Dammann, "MPC toolbox with GPU accelerated optimization algorithms," in *10th European Workshop on Advanced Control and Diagnosis*, 2012.

[37] A. Abdelfattah, D. Keyes, and H. Ltaief, "Kblas: an optimized library for dense matrix-vector multiplication on gpu accelerators," *ACM Transactions on Mathematical Software (TOMS)*, vol. 42, no. 3, p. 18, 2016.

[38] D. Tarjan, K. Skadron, and P. Micikevicius, "The art of performance tuning for CUDA and manycore architectures," *Birds-of-a-feather session at SC*, vol. 9, 2009.