SOLVING QUADRATIC PROGRAMMING PROBLEMS ON GRAPHICS PROCESSING UNIT

Yunlong Huang[1], Keck Voon Ling[1], and Simon See[2]
[1]School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore, e-mail: {huan0170, ekvling}@ntu.edu.sg
[2] Technical computing, Oracle Corporation (S) Pte Ltd, e-mail: Simon.See@Oracle.COM

Received Date: December 28, 2010

Abstract

Quadratic Programming (QP) problems frequently appear as core component when solving constrained optimal control or estimation problems. The focus of this paper is on accelerating an existing Interior Point Method (IPM) for solving QP problems by exploiting the parallel computing characteristics of GPU. We compare the so-called data-parallel and the problem-parallel approaches to achieve speed up for solving QP problems. The data-parallel approach achieves speed up by parallelizing the vector and matrix computations such as the dot-product, while the problem-parallel approach solves multiple QP problems in parallel using one GPU. Our results show that solving several QP problems in parallel could lead to better utilization of the GPU resources. This problem-parallel approach is well-suited for implementing a new type of Model Predictive Control algorithm characterized by solving multiple copies of MPC in parallel to improve closed-loop performance.

Keywords: CUDA, GPU, Model predictive control, Parallel computing, Quadratic programming

Introduction

Driven by the insatiable market demand for real-time, high-definition graphics, GPU has evolved into a highly parallel, multi-threaded, multi-core processor with tremendous computational power and very high memory bandwidth. GPU is especially well-suited to address problems that can be expressed as data-parallel computation– the same computational step, e.g. multiplication, is executed in parallel on many data elements. Nowadays, GPU is widely applied to scientific computing in fields such as biomedical analysis, computational finance and physical modeling other than the traditional image rendering applications.

Quadratic Programming (QP) problems appear frequently when solving constrained optimal control or estimation problems. An example of constrained optimal control is Model Predictive Control (MPC) [17]. MPC has been well accepted in process control applications and it now beginning to be applied to other areas such as ships [1] and aerospace [2]. A comprehensive review of MPC and its industrial applications can be found in [3] and [4].

In MPC, a QP problem is solved at every sampling interval and a sequence of optimal control updates for the process is obtained. Only the first control update is applied and a new QP problem is formed and solved at the next sampling interval, leading to the so-call receding-horizon control strategy. Solving QP problems involve a series of vector and matrix computations which are ideal candidates for parallel processing on a GPU.

Existing work on accelerating scientific computation using GPU focused predominantly on data parallel approach to achieve the reported speed up [7]. Problem parallel approach, on the other hand, is relatively unexplored. The data-parallel approach

achieves speed up by parallelizing the computation of the vector and matrix computations such as the dot-product, while the problem-parallel approach solves multiple QP problems in parallel using one GPU. One motivation for accelerating the solution of QP problems is to enable MPC to run with faster sampling intervals which could potentially improve closed-loop performance [5, 6, 18]. Recently, a new Parallel MPC algorithm has been proposed which uses a deeply pipelined FPGA to solve multiple MPC problems in parallel [8]. A data parallel approach on a modest FPGA has also been explored [9]. In this paper, we compare the data-parallel and the problem-parallel approaches to achieve speed up for solving QP problems using GPU. The GPU platform in use is the Nvidia Telsa S1070, with core processor speed of 1.33GHz for parallel computation. A Quad-Core Intel(R) Xeon(R) E5520 at 2.27GHz is used as the host machine.

This paper is organized as follows: In Section I.A, the characteristics of CUDA (Compute Unified Device Architecture) are reviewed. In Section I.B, the suitability of GPU for problem-parallel computation is discussed. The Interior Point Method (IPM) for solving QP problems is reviewed Section II. The test results of solving one QP problem on a GPU are given in Section III.A while the results for solving several QP problems concurrently are shown in Section III.B. Finally, conclusion and potential applications and extensions of this work are presented in Section IV.

Introduction to CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing architecture developed by NVIDIA for software programmers to access the parallel computing engines in GPU through standard programming languages such as the 'C' programming language. With a NVCC preprocessor, programs in CUDA can be translated into code that can be processed by a C compiler. In other words, using CUDA, the latest NIVIDA GPUs become accessible for computation like CPUs, but emphasizes executing many concurrent threads rather than executing a single thread. This approach of solving general purpose problems on GPUs is known as GPGPU (General Purpose GPU).

A CUDA program is composed of several blocks of threads. With SIMD (same instruction multiple data) architecture, each thread in a CUDA program will execute the same piece of code or kernel. The thread can access the data on its local registers and shared data in the shared memory with other threads in the same block. Inside each block, the maximum number of threads that can be allocated is 512. The blocks are allocated in grids. The thread is in three dimensions; and blocks and grids are in two dimensions. Warp is the scheduling unit for one block in a Streaming Processor (SM), where each thread block in one SM is divided into 32-thread Warps. Warps, whose next instruction has its operands ready for consumption, are eligible for execution. Eligible warps are selected for execution on a prioritized scheduling policy. When one warp is selected, one instruction will be fetched and the SM broadcasts the same instructions to 32 threads of a Warp. Physically, all the threads in one warp are executed concurrently. When threads of a warp diverge due to a data-dependent conditional branch, the warp serially executes each branch path, disabling threads that are not on that path. When all paths complete, the threads converge back to the same execution path.

Data-parallelism and Problem-parallelism

With a set of SIMD processors in a GPU, most applications are executed in a data-parallel manner. As GPU is designed for three-dimensional graphics, each thread has three IDs. For scientific computing consisting of vector-matrix computations, a thread with two dimensional ID can represent an element of a matrix and participate in the computation in a data parallel manner, resulting in computational acceleration.

To fully exploit the threedimensional threads provided by the GPU, one can organize the computations in a "problem parallel" manner if the following tow conditions are satisfied:

a. The problems are of the same size.
b. The problems are solved using the same algorithm.

Two IDs (x, y) are used to indicate the position of the element in one matrix and the third ID (z) is used to indicate which problem the matrix belongs to. To illustrate, consider the problem of matrix addition. Here, the two matrix additions, A+B and C+D can be executed in parallel if the matrices are all of the same sizes.

$$A=\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix},\ B=\begin{bmatrix}9 & 8 & 7\\ 6 & 5 & 4\\ 3 & 2 & 1\end{bmatrix},$$

$$C=\begin{bmatrix}1 & 3 & 5\\ 7 & 9 & 2\\ 4 & 6 & 8\end{bmatrix},\ D=\begin{bmatrix}2 & 4 & 6\\ 8 & 1 & 3\\ 5 & 7 & 9\end{bmatrix}.$$

Traditionally, using GPU to conduct data-parallel computation of matrix addition, we would like to compute A+B, and then C+D, with two empty array Result one and Result two to store the results. The following program illustrates the implementation of this idea.

```
__global__ void Matrix_add(float* M, float* N, float* R)
{
 // the steps of loading data into shared memory are omitted
 int tx = threadIdx.x; // x-axis coordination in one matrix
 int ty = threadIdx.y; // y-axis coordination in one matrix
 int width = 3;
 int height = 3;
 int size = width * height;
 R[ tx *width + ty] = M[ tx *width + ty]  +  N[ tx*width + ty]; //the element  in the result matrix corresponding to one thread is calculated
}
  ...
int main()
{  float *d_A, *d_B, *d_C, *d_D, *d_Result_One,   *d_Result_Two;  // Pointer device arrays
      size_t size = 3*3 * sizeof(float); //the size of device memory for each matrix
      cudaMalloc((void **) &d_A, size); //allocate memory for matrix A on device
      cudaMalloc((void **) &d_B, size);
      cudaMalloc((void **) &d_C, size);
      cudaMalloc((void **) &d_D, size);
      cudaMalloc((void **) &d_Result_One, size);
      cudaMalloc((void **) &d_Result_Two, size);
      cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice); //Copy matrix A from host to device
      cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
      cudaMemcpy(d_C, C, size, cudaMemcpyHostToDevice);
      cudaMemcpy(d_D, D, size, cudaMemcpyHostToDevice);
      dim3 block( 3, 3 ); // To allocate 3*3 thread in each block for the parallel computation
      Matrix_add<<<1, block>>>(d_A, d_B, d_R_One); // " 1" stands for allocating one block in the grid,  '"block" is the number of threads define before in each block,
      Matrix_add<<<1, block>>>(d_C, d_D, d_R_Two);
  }
```

In this method, the function of matrix addition has to be called twice for the computation of two matrix addition. Taking the problem parallel approach, the following code fragment shows one possible way by storing the elements of the matrices A and C,

and B and D in two one-dimension arrays:

$$M = [1,2,3,4,5,6,7,8,9,1,3,5,7,9,2,4,6,8]$$
$$N = [9,8,7,6,5,4,3,2,1,2,4,6,8,1,3,5,7,9]$$

The results are stored in an empty array R.

```
__global__ void Matrix_add(float* M, float* N, float* R)
{
 // the steps of loading data into shared memory are omitted
 int tx = threadIdx.x; // x-axis coordination in one matrix
 int ty = threadIdx.y; // y-axis coordination in one matrix
 int tz = threadIdx.z; // indicate which problem the matrix    belongs to
 int width = 3;
 int height = 3;
 int size = width * height;
 R[tz*size + tx *width + ty] = M[tz * size + tx *width + ty]  +   N[tz*size + tx*width + ty];
}
int main(){   ...
     dim3 block( 3, 3, 2);
  ... //similar steps in memory allocation and memory copy from host to device like previous program
     Matrix_add<<<1, block>>>(d_M, d_N, d_R);
}
```

Such problem parallel technique improves the efficiency of GPU.

Solving QP problems on GPU

It is well-known that Model Predictive Control (MPC) can be formulated as a QP problem [10, 11, 12] which takes the form:

$$\min_{z\in\mathbb{R}^{n_v}}\{\frac{1}{2}z'Qz + c'z\} \text{ subject to } Jz \le g \qquad (1)$$

where the decision variable z is a nx1 vector; Q is a nxn positive definite matrix, c is an nx1 constant known vector; J and g have sizes $m_c \times n$ and $m_c \times 1$ respectively, where m_c is the total number of inequalities.

Two common methods for solving QP problems are the Interior Point Method (IPM) and the Active Set Method (ASM). The worst-case complexity of ASM increases exponentially with the problem size and the size of the linear system that need to solve changes depending on which constraints are active each loop of the ASM loop. This makes ASM unsuitable for GPU implementation for reasons mentioned in Section I-B. The IPM, on the other hand, has polynomial complexity and maintains a constant predictable structure of the linear system that need to be solve each loop of the IPM loop. Hence the IPM is chosen for implementation on GPU. In addition, the regular structure of IPM will be exploited for problem parallelism, i.e. solve several QP problems in parallel.

The idea of IPM is to approach the solution of the Karush-Kuhn-Tucker (KKT) equations by successive descent steps. Each descent step is a Newton-like step and is obtained by solving a system of linear equations. Each loop of the iteration can be expensive to compute, but can make significant progress toward the solution. The algorithm of an infeasible interior point method is as follow [12, 13] :

1. Initial condition (z_0, λ_0, t_0) with positive (λ_0, t_0) and select a positive termination threshold δ

2: **for** k = 0, 1, 2 ... **do**

3: $\Lambda \leftarrow diag(\lambda_k)$ and $T \leftarrow diag(t_k)$

4: Solve for Δz with $(Q + J'\Lambda TJ)\Delta z = -r_d - J'T^{-1}\Lambda(-r_b - t_k + \sigma\mu_k\Lambda^{-1}e)$

5: Calculate $\Delta\lambda = \Lambda T^{-1}(J\Delta z_k - r_b) - \Lambda + \sigma\mu_k\Lambda^{-1}e$ and $\Delta t = -t_k + \Lambda^{-1}(\sigma\mu_k e - T\Delta\lambda)$

6: $\alpha \leftarrow \min\{\min_{i\in\Gamma(\Delta t)}\{-\frac{t_k(i)}{\Delta t(i)}\}, \min_{i\in\Gamma(\Delta\lambda)}\{-\frac{\lambda_k(i)}{\Delta\lambda(i)}\}\}$

7: $(z_{k+1}, \lambda_{k+1}, t_{k+1}) \leftarrow (z_k, \lambda_k, t_k) + \alpha(\Delta z, \Delta\lambda, \Delta t)$

8: $\mu_{k+1} \leftarrow t'_{k+1}\lambda_{k+1}m_c^{-1}$

9: **if** $\mu_{k+1} \le \delta$ **then**

10: Terminate with solution $z^* = z_{k+1}$

11: **end if**

12: **end for**

where,

$r_d = Qz_k + J'\lambda_k + c$,

$r_b = -Jz_k + g - t_k$,

$e = (1,1,...,1) \in \mathbb{R}^{m_c \times 1}$

$\Gamma(\Delta t) = \{i = 1,2,...,m_c : \Delta t(i) < 0\}$,

$\Gamma(\Delta\lambda) = \{i = 1,2,...,m_c : \Delta\lambda(i) < 0\}$,

In the algorithm, $\sigma = 0.25$ is pre-defined. The sub-optimality of z_k is measured by $\mu_k = t'_k\lambda_k m_c^{-1}$, where t_k and λ_k are iterated for the slack variables and Lagrange multipliers; m_c is the number of inequalities and n_v is the number of decision variables.

In line 4 of IPM algorithm, a system of linear equations in the form of $Ax = b$ is solved. For a linear system with n variables, time cost of solving such a system with Gaussian - Jordan elimination will be O(n³) - it needs n times iteration and the computation cost in each iteration loop is O(n²), where matrix multiplication and addition are conducted. These computation in each iteration of the Gaussian-Jordan elimination will be reduced to be O(n) using data parallelism on a GPU. Thus, the computational complexity of Gaussian-Jordan elimination can be reduced to O(n²) on a GPU. Other IPM steps are mainly vector or matrix operations whose computational complexity can be similarly reduced from O(n²) to O(n). All computations are executed on the GPU in single precision floating point arithmetic, with the exception of floating point divisions in line 4 (inversion of t_k (T^{-1})) and line 6 ($\frac{t_k(i)}{\Delta t(i)}, \frac{\lambda_k(i)}{\Delta\lambda(i)}$) in the IPM algorithm. This is because in our current implementation, we found that the algorithm is very sensitive to rounding errors and in our current implementation, the floating point division operations are executed in the host machine sequentially in double precision.

Results

A. Solving one QP problem on GPU

The performance of solving one QP problem is investigated in this section. As mentioned earlier, a QP can be parameterized by two variables: n, the number of decision variable and m_c, the number of inequality constraints. For simplicity, we set $m_c = 10n$, since in most circumstances, more decision variables usually lead to more inequality constraints. By

profiling the time taken for each step in one iteration loop of the IPM, which is implemented with only simply matrix parallel computation of GPU, it was found that about 80% of the computational time was spent in matrix multiplication- AJ, where A is of size of $n \times m_c$ and J is of size of $m_c \times n$, and $A = J'\Lambda T^{-1}$ is computed in previous steps.

```
__device__ void kernel (float* A, float* J, float* result)
{
 int x = blockIdx.x*blockDim.x + threadIdx.x;
 int y = blockIdx.y*blockDim.y + threadIdx.y;
 for (int i = 0; i < m_c ; i++)
 {
     result[ x*n + y] + = A[x*n + i]*J[ i*m + y];
     }
 }
```

The main computation instructions in each thread will be m_c multiplication and m_c addition. As our test data grows very fast, the matrices, A and J are becoming large matrices. On GPU, large matrix multiplications can be accelerated by dividing the matrices into smaller matrices [14]. For example, if there are three matrices, $A[hA \times wA]$, $B[hB \times wB]$ and $C[hA \times wB] = A \times B$, matrices A and B can be divided into several small matrices and for each element in C. It means that the loop of multiplication and addition is broken up into several phases. For each phase, the corresponding sub matrices of A and B are loaded into the shared memory and the result of the phase will be used to calculate the corresponding element in C. Figure 1 and the code fragment that follows illustrates this idea.

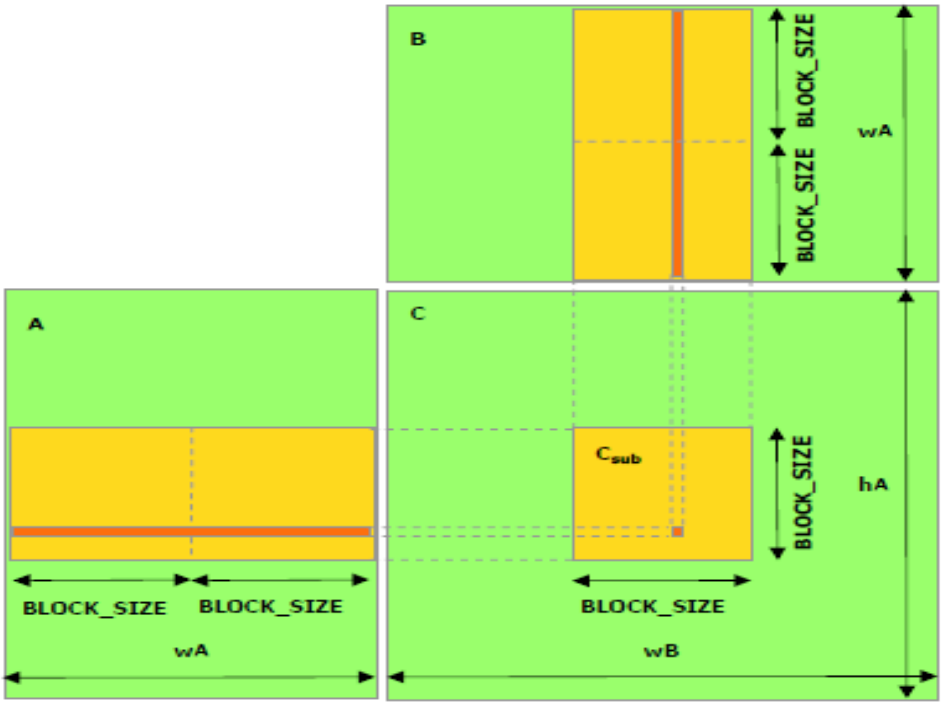

Figure 1 improved matrix multiplication

```
__global__ void MatrixMulKernel(float* A, float* B, float* C)
{
 __shared__ float As[TILE_WIDTH][TILE_WIDTH];
 __shared__ float Bs[TILE_WIDTH][TILE_WIDTH];
 int bx = blockIdx.x;
 int by = blockIdx.y;
 int tx = threadIdx.x;
 int ty = threadIdx.y; //identify the row and column of the C element to work on;
 int Row = by * TILE_WIDTH + ty;
 int Col = bx * TILE_WIDTH + tx;
 float Cvalue = 0;
 for ( int m=0; m < Width/ TILE_WIDTH; ++m)
 {
     As[ty][tx] = A[Row * Width + ( m* TILE_WIDTH + tx)]; //load A and B tiles into shared memory
      Bs[ty][tx] = B[ Col + (m*TILE_WIDTH + ty) * Width];
      __syncthreads();
      for( int k = 0; k < TILE_WIDTH; ++k)
      Pvalue + = As[ty][k]*Bs[k][tx];
      Syncthreads();
   }
 C[Row*Width + Col] = Pvalue;
}
```

The program for the improved matrix multiplication is developed earlier by others [14].

Dividing large matrices into smaller matrices will increase the number of threads. The shared memory in each SM and the size of sub matrices for each block has been discussed Dividing large matrics into smaller matrics will increase the number of threads. The shared memory in each SM and the size of sub-matrices for each block has been discussed in [14]. Figure 2 shows the test results of IPM implemented sequentially, and in data parallel manner on GPU with and without using the method of partitioning large matrices. In Figure 2, the sequential program refers to implementing the IPM sequentially, without parallel computation. This serves as a baseline to compare the speed up for data parallel implementation and its improved version. As can be seen in Figure 2, the method of partitioning large matrices resulted in a significant acceleration, about 5x over that without partitioning. It has been reported that GPU could reach about 25x acceleration over CPU [14].

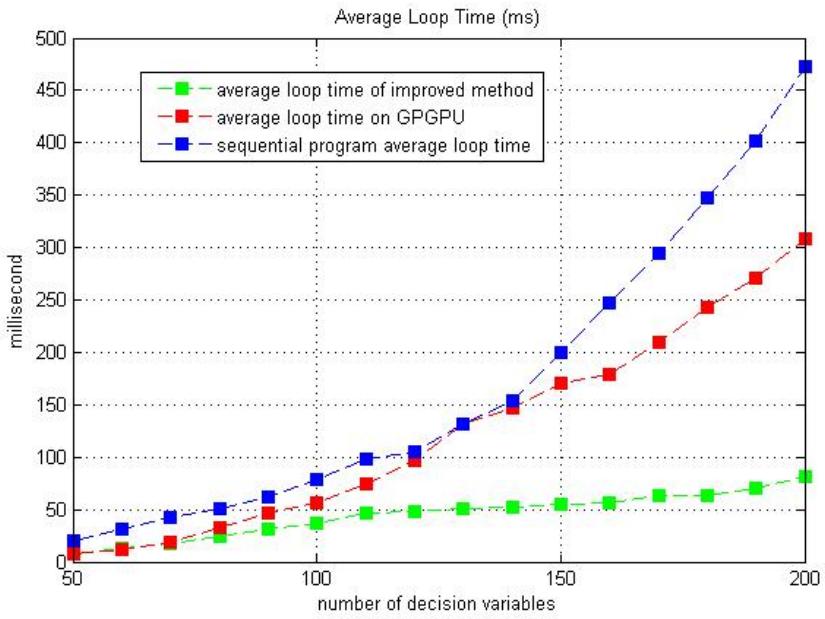

Figure 2. Implementation IPM on GPGPU

From Figure 2, there are two observations: (1) comparing sequential IPM and data parallel IPM, the computational time for sequential implementation increases at a faster rate than data parallel implementations (with and without matrix partitioning), and (2) data parallelism does not significantly accelerate the solution of QP on GPU when the size of QP is small (less than 100 decision variables).

The first observation supports the analysis in Section II which concluded that GPU, with data parallelism, could reduced computational complexity from $O(n^3)$ to $O(n^2)$. It can also be seen that the improved IPM (with matrix partitioning) on GPU can accelerate about 6x over the sequential program. The second observation highlights that the benefit of GPU only shows when solving very large QP problems (more than 100 decision variables). When the number of decision variables is small, the number of blocks will be small. Only a few SMs will be allocated with blocks, other SMs will be idle. In each SM, the blocks are divided into warps, only the threads in one warp will be executed concurrently. So, even though several SMs are working, only a few warps of threads will be executed in parallel. Thus, the computational resources of GPU is not fully utilized and the additional overhead incurred outstrips the speed up for small QP problems.

B. Solving several QP problems in parallel

In this section, we consider solving several QP problems of the same size in parallel on a GPU. Even though the number of iteration weakly depends on the number of decision variables, each QP will reach to their optimal solution with different number of iteration. Then one flag for each QP problem status will be used to indicate whether the QP has been solved or not. In the end of each loop of each problem, if some problems reach their optimal solutions, their flags will be changed. In later loops, the processors will check the flags of all problems and skip the problems who have reached their optimal solutions.

```
void one_step_in_iteration(d_flag, d_M, d_N, .....)
{
  if( d_flag[tz]==0)
  {
     // conduct the computation;
     }
```

This kind of situation will result in the thread divergence, which will slow down the parallel processing. However, as the iteration number of IPM weakly depends on number of decision variables. The thread divergence will only happen in the last few loops. The effect is not very serious. There is an alternative. At the end of each iteration loop, check the problems status, which reach their optimal solutions and which do not. Formulate those unsolved problems into the next iteration loop. However, latter method will increase the latency from loading memory between device and host more frequently than the former one. In the latter test, method with flag is implemented.

Figure 3 shows the results of solving 1 to 20 QP problems in parallel on a GPU. Each QP is of 50 decision variables and 500 inequality constraints.

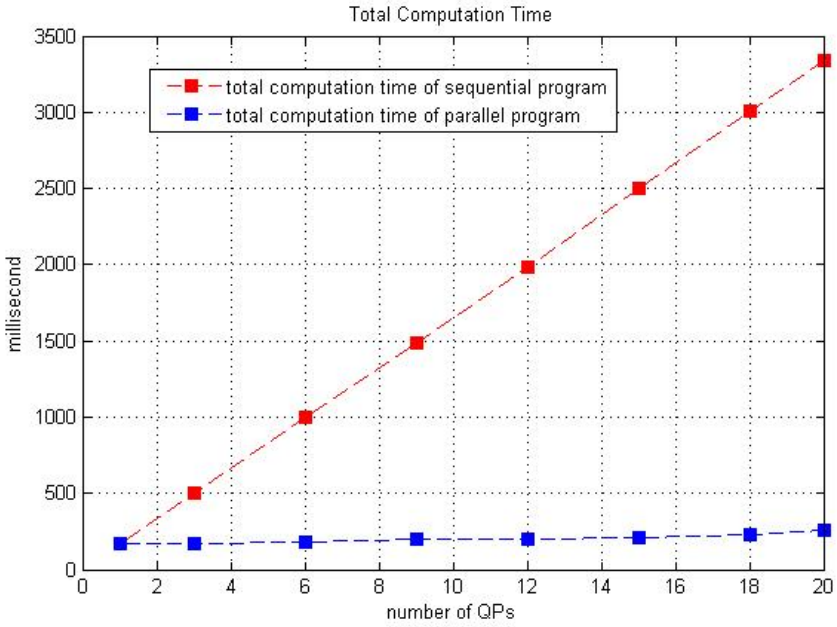

Figure 3. Solving multiple QP problems concurrently

From Figure 3, it is clear that problem parallelism, i.e. solving several QP problems in parallel, would fully utilise the available computational resources of GPU. The GPU can accelerate around 11x when solving 20 QPs with 50 decision variables and 500 inequalities. The increment of time cost mainly results from the floating point error sensitive steps which were executed on the host machine sequentially, rather than on the GPU in parallel. In conclusion, by employing both data and problem parallelism, GPU can achieve good speed up in solving several QP problems in parallel. As explained in previous section, with application of problem-parallel idea, this achievement is gained from the enhancement of utilization of computation resource of the GPU. For small problem, sequentially solving one by one only make two to three SM works. However, with solving several QP problems in parallel, more SM will be active in the computation. Meanwhile, from the point of view on parallel computation, employment of problem-parallel adds a new level parallelism in the computation. With data-parallel and problem-parallel ideas work together contributing to computational time reduction.

Conclusion and future work

In this paper, the IPM algorithm was implemented on a GPU. The data parallel and the problem parallel approaches were compared. It has been demonstrated that, in general,

GPU can accelerate the solution of QP problems if the size of the QP problem is very large (with more than 100 decision variables). For smaller QP problems, the overhead incurred outstrips the speed up. In other words, the available computational resources available in a GPU could be put to good use by solving several QP problems in parallel on one GPU. The results obtained suggest that the problem-parallel approach achieves a better utilization of GPU resources. It should be noted that the QP problems used in this paper are randomly generated according to [15], rather than from any MPC problem. Hence, there could be room for further improvement by exploiting the structure of the QP problems arising from MPC. For example, [10] advocates a sparse banded matrix formulation of QP and its effect on a GPU implementation should be investigated. In addition, the problem parallel approach could also be well-suited to implement a new type of parallel MPC algorithms on a single GPU. In a recent development parallel MPC [8] and Channel-Hopping MPC [16] have been proposed. In parallel and channel-hopping MPC algorithm, several MPC problems are solved in parallel, and the MPC which gives the smallest cost is selected for implementation. The parallel and Channel-Hopping MPC appear to be suitable candidates for implementation on the GPU using techniques described in this paper, and it is a subject of current research.

References

[1] T. Perez, G.C. Goodwin, and C.W. Tzeng, "Model predictive rudder rolls stabilization control for ships," In: *Proceedings of 5th IFAC Conference on Manoeuvring and Control of Marine Craft*, 2002.

[2] A. Richards, and J.P. How, "Model predictive control of vehicle maneuvers with guaranteed completion time and robust feasibility," In: P*roceedings of the 2003 American Control Conference*, Vol. 5, pp. 4034-4040, 2003.

[3] C.E. Garcia, D.M. Prett, and M. Morari, "Model predictive control: Theory and practice," *Automatica*, Vol. 25, pp. 335–348, 1989.

[4] D.Q. Mayne, and H. Michalska, "Receding horizon control of nonlinear systems," *IEEE Transactions on Automatic Control 35*, pp. 814–824, 1990.

[5] K.V. Ling, J.M. Maciejowski, and B.F. Wu, "Multiplexed Model Predictive Control," Paper presented at the *16th IFAC World Congress*, Prague, 2005.

[6] K.V. Ling, J.M. Maciejowski, A. Richards, and B.F. Wu, *Multiplexed Model Predictive Control*, 2010. [Online]. Available: http://arxiv.org/abs/1101.2785 [Accessed: January 2011]

[7] M. Ławryńczuk, "Computationally efficient nonlinear predictive control based on state-space neural models," In: *PPAM'09 Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part I*, 2009.

[8] J.L. Jerez, G.A. Constantinides, E.C. Kerrigan, and K.V. Ling "Parallel MPC for Real-Time FPGA-based Implementation," Paper presented at *IFAC World Congress*, 2011

[9] K.V. Ling, B.F. Wu, and J.M. Maciejowski, "Embedded Model Predictive Control (MPC) using a FPGA," In: *Proceedings of the 17th IFAC World Congress*, Seoul, Korea, pp. 15250-15255, 2008.

[10] S.J. Wright, "Applying new optimization algorithms to model predictive control," In: *Proceeding of Chemical Process Control V*, pp. 147-155, 1997.

[11] R.A. Barllet, L.T. Bieger, J. Backstrom, and V. Gopal, "Quadratic programming algorithms for large scale model predictive control," *Journal of Process Control*, Vol. 12, No. 7, pp. 775-795, 2002.

[12] K.V. Ling, S.P. Yue, and J.M. Maciejowski, "A FPGA Implementation of Model Predictive Control," In: *Proceedings of the 2006 American Control Conference*, Minneapolis, Minnesota, United States of America, pp.1930-1935, 2006.

[13] C.V. Rao, S.J. Wright, and J. B. Rawlings, "Application if interior-point methods to model predictive control," *Journal of Optimization Theory and Applications*, Vol. 99, No. 3, pp. 723-757, 1999.
[14] R. Abdelkhalek, H. Calandra, O. Coulaud, G. Latu, and J. Roman, "Fast seismic modeling and reverse time migration on a GPU cluster," Paper presented at *2009 High Performance Computing & Simulation-HPCS'09*, Leipzig, Germany, 2009.
[15] M. L. Lenard, and M. Minkoff, "Randomly generated test problems for positive definite quadratic programming," *ACM Transactions on Mathematical Software*, Vol. 10, No.1, pp. 86-96, 1984.
[16] K.V. Ling, J.M. Maciejowski, J. Guo, and E. Siva, "Channel-Hopping Model Predictive Control," paper presented at *IFAC World Congress*, 2011.
[17] J.M. Maciejowski, *Predictive Control with Constraints*, Prentice-Hall, 2002.
[18] K.V. Ling, W.K. Ho, B.F. Wu, A. Lo, and H. Yan, "Multiplexed MPC for multi-zone thermal processing in semiconductor manufacturing," *IEEE Transactions on Control Systems Technology*, Vol.18, No. 6, pp.1371-1380, November 2010.