COMP 605: Introduction to Parallel Computing Quiz 4: Module 4 Quiz: Comparing CUDA and MPI Matrix-Matrix Multiplication

Mary Thomas

Department of Computer Science
Computational Science Research Center (CSRC)
San Diego State University (SDSU)

Due: 05/12/17
Updated: May 13, 2017

Table of Contents

Comparing an MPI and CUDA Mat-Mat-Mult

- You may work with another member of the class on this project.
- Objective:
  - Develop and test a CUDA Matrix-Matrix Multiplication code
  - Compare your CUDA results to MPI reference data
  - All input and source code can be found in $/COMP605/quiz4/$
- Serial mat-mat-mult.c available in $/COMP605/quiz4/$
- MPI mat-mat-mult:
  - you do not need to run any MPI code
  - you can use the reference data provided in the *quiz* directory
  - MPI version in Pacheco text, Parallel Programming with MPI, Ch7.
- CUDA mat-mat-mult source code:
  - Your may write your own code, or modify existing code.
  - Working CUDA source code provided in $/COMP605/quiz4/$:
    - CUDA Toolkit (and other sources): `http://docs.nvidia.com/cuda/cuda-samples/index.html`
    - Nitin Gupta (Nvidia developer): `http://cuda-programming.blogspot.com/2013/01/cuda-c-program-for-matrix-addition-and.html`

Programming Instructions

- Generate input matrices A & B from within code
- All key variables and filenames read from command line
- Matrix size N and allocations should be dynamic
- Vary #threads/block for a given N (see Figure 2 below).
- Use cuda properties to check that your matrix fits on the device and to set the device
- All jobs should be run using batch scripts
- For *small* test cases (< 10), include logic to print out examples of A, B, and C.

- Vary the size of the matrices using square $[N_i \times N_i]$ matrices
- Vary #CUDA threads: use square grid/block/thread distribution
- Recall: GPU hardware limits the number of blocks per grid and the number of threads per block
- Larger problems require use of both grid and blocks
- Need to control the number of threads, since they are smaller
- Fix number of threads and distributed chunks along the blocks:

```
add<<<128,128>>>( dev_a, dev_b, dev_c);
add<<<h_N,h_N>>>( dev_a, dev_b, dev_c);
add<<<ceil(h_N/128),128>>>( dev_a, dev_b, dev_c);
add<<<(h_N+127)/128,128>>>( dev_a, dev_b, dev_c);
```

- if maxTh == maximum number of threads per block:

```
add<<<(h_N+(maxTh-1))/maxTh, maxTh>>>( dev_a, dev_b, dev_c);
```

- Compute thread index as:
  $tid = threadIdx.x + blockIdx.x * blockDim.x;$

```
$tid = threadIdx.x + blockIdx.x * blockDim.x;$
```

Performance

- keep track of what node/core you used (set the device)
- Timing:
  - Time critical blocks (T_{wall}) or (T_{kernel})
  - Compare MPI CPU to GPU timings.
- What is the largest #threads you were able to test? What happened, why do think this happened

Suggestions on what to Report/Turn in for both problems:

- Create the homework directory *USER/quiz/q4* with correct access permissions.
- Short lab report with comments, figures and table labels.
- Explain your results for Thread and ProbSize scaling.
- Include relevant tables of your test data
- Evidence you ran your jobs using the batch queue (short/small job); examples of batch scripts
- Plots of key results.
- A copy of your code (single spaced, two sided, two column format is OK).
- Reference key sources of information *in your report and code* where applicable.

Comparing MPI and CUDA

- You cannot directly compare scaling for MPI #cores against CUDA number of threads.
- You can compare common run-time characteristics and variables:
  - All runs can have same (or close) problem sizes
  - All runs can be timed
  - Identify $T_{optimal}$ for each programming model:

    $T_{optimal}$ is defined as the point where increasing the number of processors or the number of threads/block no longer significantly reduces the run-time ('turnover' point).
- Figures 1 & 2: determining $T_{optimal}$ for MPI and CUDA programming models.
- Figure 3: comparison of $T_{optimal}$ for the two programming models .
- Figures 4-6: MPI reference data provided for this assignment.
- Note: Figures are *not* for mat-mat-mul, so your data values may differ, but the trends should not change.

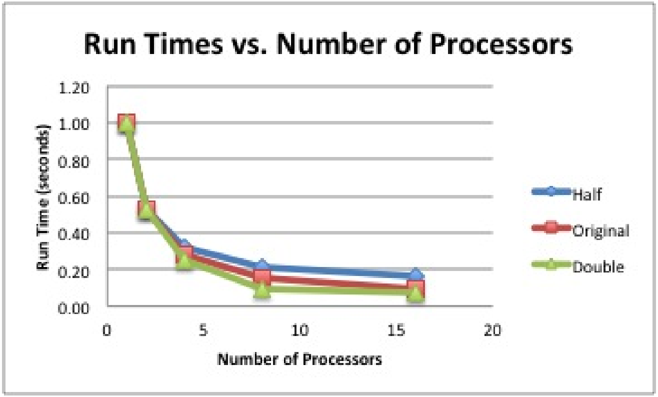

Figure 1: The figure above shows the run-time as a function of the number of processors, for different problem sizes,using MPI. The run time decreases as the number of cores increases, up to a limit where there is not much improvement. In this case, $T_{optimal}$ 16 cores

Example Comaring MPI and CUDA

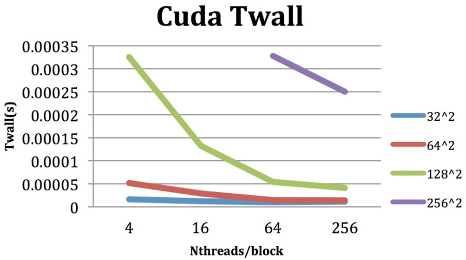

Figure 2: T_{wall} for different $N_{threads}/block$ vs Dim. The figure above shows the run-time as a function of the number of threads per block, for different problem sizes, using CUDA. The run time decreases as the number of threads per block increases, up to a limit where there is not much improvement. In this case, $T_{optimal} = 64$ threads/block.

Source: Fall 2012: G. Pham

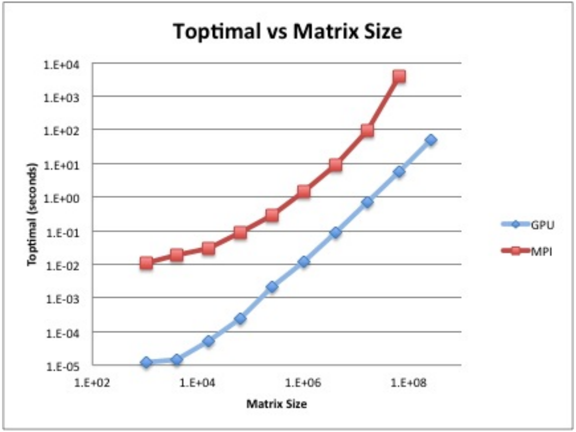

Figure 3: $T_{optimal}$ as a function of matrix size for MPI and CUDA/GPU tests. The figure above shows that for a given problem size, $T_{optimal}$ for the GPU programming model is better that MPI. This problem is for a matrix-matrix multiplication problem.

Source: Fall 2012: G. Pham

MPI Matrix-Matrix Multiplication ref data

Cores	420	1260	1680	2520	3360	4200	5040
1	7.78E-01	2.56E+01	6.91E+01	2.34E+02	4.98E+02	1.09E+03	1.88E+03
4	1.96E-01	5.61E+00	1.37E+01	5.85E+01	1.37E+02	2.71E+02	4.66E+02
9	1.10E-01	2.35E+00	6.84E+00	2.39E+01	5.91E+01	1.25E+02	2.10E+02
16	6.28E-02	1.64E+00	3.68E+00	1.38E+01	3.30E+01	8.05E+01	1.39E+02
25	4.88E-02	1.45E+00	3.52E+00	8.23E+00	3.00E+01	6.79E+01	8.67E+01
36	9.40E-02	9.70E-01	2.43E+00	8.23E+00	2.20E+01	4.08E+01	7.58E+01
49	4.01E-02	7.08E-01	1.98E+00	4.59E+00	1.25E+01	2.49E+01	3.99E+01

Figure 4: MPI Matrix-Matrix Multiplication ref data. The Table shows the runtime (in seconds) as a function of the number of processors for different matrix sizes.

MPI Matrix-Matrix Multiplication ref data

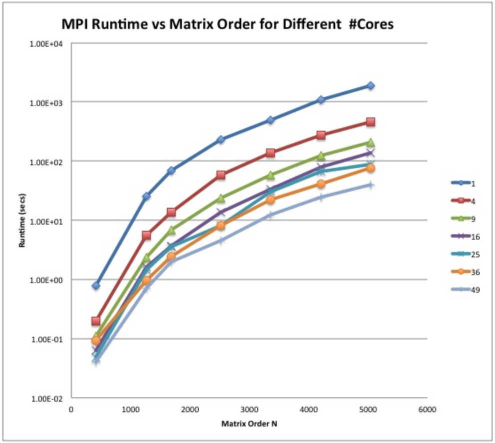

Figure 5: MPI Matrix-Matrix Multiplication ref data: Curves show the runtime (in seconds) as a function of the matrix size for different number processors.

MPI Matrix-Matrix Multiplication ref data

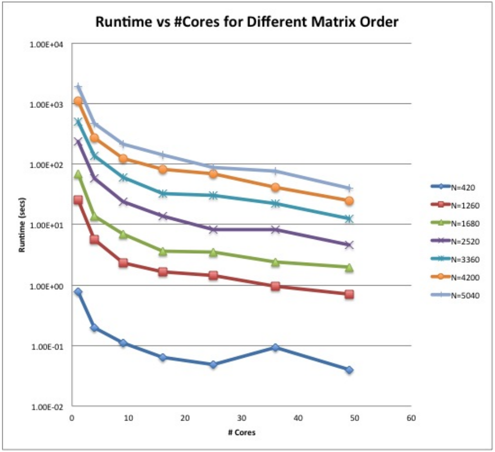

Figure 6: MPI Matrix-Matrix Multiplication ref data. The Table shows the runtime (in seconds) as a function of the number of processors (Cores) vs matrix size M, for matrices of dimension $[MxM]$.

The CUDA Compiler support for doubles: nvcc

- you can install CUDA toolkit, compile code without a GPU device.
- To compile use: *nvcc*
- **NOTE: CUDA does not support doubles on the *device* by default:** You need to add the switch "-arch sm_13" (or a higher compute capability) to your nvcc command:

```
[mthomas/dblTst]
[mthomas/dblTst]nvcc -o dblTst dblTst.cu
nvcc warning : The 'compute_10' and 'sm_10' architectures are
deprecated, and may be removed in a future release.
ptxas /tmp/tmpxft_00006578_00000000-5_dblTst.ptx, line 76;
warning : Double is not supported. Demoting to float
[mthomas/dblTst]
```

```
[mthomas/dblTst]
[mthomas/dblTst] nvcc -arch=sm_13 -o dblTst dblTst.cu
[mthomas/dblTst]
```