COMP 605: Introduction to Parallel Computing
Homework 6: GPU/CUDA Programming:
Calculating PI.
Comparing MPI, OpenMP, and CDUA Results

Mary Thomas

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

Due: 04/27/16
Posted: 04/22/16
Updated: 04/24/16
## Table of Contents

1. HW #6, P1: Using Numerical Integration to Estimate $\pi$
2. HW #6, P2: Comparison of different parallel models for estimating PI
3. What to Report/Turn in for both problems:
HW #6, P1: Using Numerical Integration to Estimate $\pi$

![Figure 1](http://www.mathsisfun.com/numbers/pi.html)

$\pi = \frac{\text{Circumference of a Circle}}{\text{Diameter of a Circle}}$

Image Source: http://www.mathsisfun.com/numbers/pi.html
HW #6, P1: Using Numerical Integration to Estimate $\pi$

- Integral representation for $\pi$
  \[ \int_{0}^{1} dx \frac{4}{1+x^2} = \pi \]

- Discretize the problem:
  \[ \Delta = 1/N : \text{step} = 1/N_{\text{areas}} \]
  \[ x_i = (i + 0.5)\Delta (i = 0, \ldots, N_{\text{areas}} - 1) \]
  \[ \sum_{i=0}^{N-1} \frac{4}{1+x_i^2} \Delta \approx \pi \]

Figure 3

$\pi$ Formulae: http://en.wikipedia.org/wiki/Approximations_of_pi
Image: http://cacs.usc.edu/education/cs596/mpi-pi.pdf
#include <stdio.h>
#define NAREA 10000000

void main() {
    int i; double step,x,sum=0.0,pi;
    step = 1.0/NAREA;
    for (i=0; i<NAREA; i++) {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    pi = sum*step;
    printf(PI = %f\n,pi);
}
HW #6, P1: Instructions

- Write a CUDA program that uses numerical integration to estimate $\pi$.
- Find a reference value for $\pi$ to the limits of a double precision number.
- Estimate $\pi$ to the limits of a double precision number.
- Calculate the value for $\pi$ as a function of the number or areas used and number of threads.
- Calculate the error of your estimate: $Err = \pi_{\text{ref}} - \pi_{\text{measured}}$
- Use double precision for all calculations and outputs.
HW #6, P1: Instructions (cont.)

- Parse all key variables from the command line.
- Run the jobs using the batch queue.
- ProbSize Scaling:
  - Choose $N_{\text{areas}}$ to allow scaling from appx $10^3$ to $> 10^7$ or greater.
- Thread scaling:
  - Vary the total number of threads on the GPU Device.
  - Vary the total number of threads by changing the number of threads-per-block and blocks-per-grid (e.g.):

```c
int threadsperblock=atoi(argv[1]); /* read num thds from command line */
blocksPerGrid = imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );
add<<<blocksPerGrid,threadsPerBlock>>>( dev_a, dev_b, dev_c);
```

- What is the max number of total threads you can use and why?
- Time the job runs to check that you getting the proper scaling.
In this part of the HW, you will develop a method for comparing the scaling of the different methods used to estimate the value of PI.

Base this study on work you have done in past HW’s

Use your results
- Distributed computing: MPI
- Shared Memory: OpenMP
- GPU Programming: CUDA
Suggestions for comparing MPI and CUDA

- You cannot directly compare the MPI PEs/cores against the OpenMP or CUDA number of threads.
- You can compare common run-time characteristics and variables.
  - All runs can have same problem sizes
  - All runs can be timed
- $T_{optimal}$ can be defined as:
  - 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).
  - The figures below show how to identify $T_{optimal}$ for MPI and CUDA programming models.
- Figures 4-6 below show examples of how to determine $T_{optimal}$ for the different programming models and one way to compare the models (there are other ways). Note: the data is not for timings of code used to calculate $\pi$, so your data may differ.
Figure 4: 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.
Figure 5: CUDA $T_{wall}$ 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 6: $T_{optimal}$ 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
What to Report/Turn in for both problems:

- Create the homework directory `USER/hw(hw5)` 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.