# CSE-375/CSE-475-Prin & Prac In Parallel Comp-SP16 Final Project

XinYang(xizc15)
Song Li(sol315)

May 17, 2016

## 1 Introduction

Parallel computing has already got it's population. In order to learn more about parallel computing, we did an experiment about comparing the performance of different parallel platforms. The purpose of this project is to compare the parallel performance of CUDA, pthread, Intel intrinsics AVX2 and Intel tbb. In order to do this job, we implemented two algorithms – matrix multiplication and k-means. For each of them, we implemented four programs for these four platforms.

## 2 Test Machine

We test our code mainly in two machines:

- Machine 1
  - CPU: Intel Core i5-2410M Processor (3M Cache, up to 2.90 GHz)
  - GPU: NVIDIA GeForce GT 550M
- Machine 2
  - CPU: Intel@ Xeon(R) CPU E5630 @ 2.54Ghz * 8
  - GPU: Geforce GTX 980 Ti/PCIe/SSE2

## 3 Environment Setting Up

We try to develop CUDA on two different popular platforms, namely Microsoft Windows and Linux (Ubuntu). In fact, we develop our CUDA in Microsoft Windows platform to take advantage of Microsoft Visual Studio's excellent development environment; and later we move test and run our experiments on a high performance Linux workstation.

### 3.1 Windows

To set up an environment for CUDA on Microsoft Windows is relatively easy. We follow Nvidia's *CUDA Getting Started Guide for Microsoft Windows*. The machine was installed with Microsoft Visual Studio Community 2013 before and a Nvidia Geforce 980M graphical card. So we just followed the guide, downloading and installing the CUDA toolkit on the machine. We also verify the installation by running the sample programs in the CUDA toolkit.

### 3.2 Ubuntu

It is a bit complicated to set up a CUDA environment on a Linux (Ubuntu) machine. But along the way, the Nvidia's *NVIDIA CUDA Getting Started Guide for Linux* gives us great help. We first check the device, and install the CUDA toolkit following the instruction, finally verify the success of installment.

## 4 First Example: Matrix Multiplication

### 4.1 pthread

In this part, we use gcc pthread to do multi-thread computing. In Machine 1,the size of matrix is 4096 * 4096. we use 4 threads to run and in Machine 2, the size of matrix is 8192 * 8192. we use 8 threads. (Based on the physical condition of these CPUs).

### 4.2 TBB

In this part, we use Intel TBB and gnu++ to do the experiment. The size of matrix is same as the size in pthread test. By many times of tests, finally we choose the number 64 as the single block size(Based on the performance of different block size). The number of threads is controlled by tbb so we don't need to decide it.

### 4.3 AVX2

In order to have a better performance, we added AVX2 to our experiment. The size of matrix is also same as the other two tests. The number of threads is same as the pthread test. By using AVX2, the performance is significantly enhanced.

### 4.4 CUDA

As our first CUDA experience, this program mainly study the codes from a tutorial [1], make some slight modifications and some optimizations. Besides, we add a few lines of coding for performance measuring, to compare its efficiency with CPU versions.

Now we describe the method in detail. We first allocate our source matrices $A, B$, and target matrix $C$. Then we copy the $A, B$ from host to device. So the most important part is to invoke our kernel. For the kernel, just like [1], we decompose the whole matrix in a set of $16 \times 16$ 2D blocks. Then we let each thread just compute the element that allocated to it according to the block index and thread index within the block. To accurately timing the GPU part of the program, we take CUDA event to record the start and the end of the calculation, so to calculate total running time in GPU. After synchronizing with ending event, we copy the matrices back to the host, and print its running time and some of the resulting matrix for checking.

## 5 Depth Study: CUDA Programming and Machine Learning

Cuda can be used in many machine learning or deep learning tasks. For example, it can help to train neural networks for image classification. In this project, we explore to take CUDA to implement a simple clustering algorithm K-means. We observe how the

### 5.1 CPU

The experiment of K-means is mostly same as matrix multiplication. Except that the size of k-means test is different. The size is 8192 * 1500 * 50. (The number of points is 8192, the test will have 1500 iterations and the dimension of a point is 50)

### 5.2 CUDA

Our CUDA K-means algorithm just like the CPU one's, but with the two stage of the K-means update are run parallel in GPU. After we copy the sample data points and initial centers of clusters into device, we start the K-means iterations. The first stage is to find the nearest center for each sample data points. In this part, we partition the workload in the way that each thread calculate the nearest centers for exactly one sample data points. Then we synchronize the threads, go to the stage of computing centers in terms of all the labels of sample data points. In this stage we let one each thread calculate one coordinate for a given label, and then combine all the coordinates together. We continue this process until there is no drops in SSE (sum of squared errors) between adjacent two iterations. As same in the matrix multiplication experiment, we record the time GPU consuming.

## 6 Evaluation

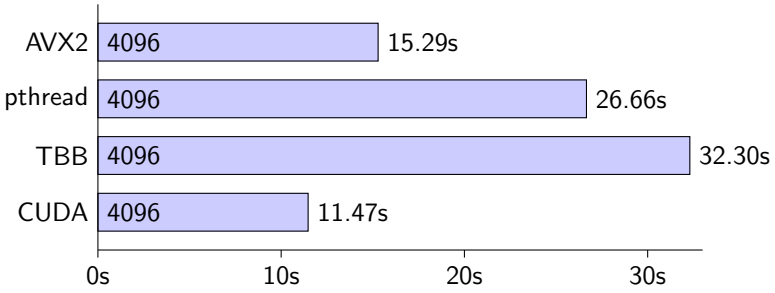


Figure 1: Machine 1 Matrix Multiplication

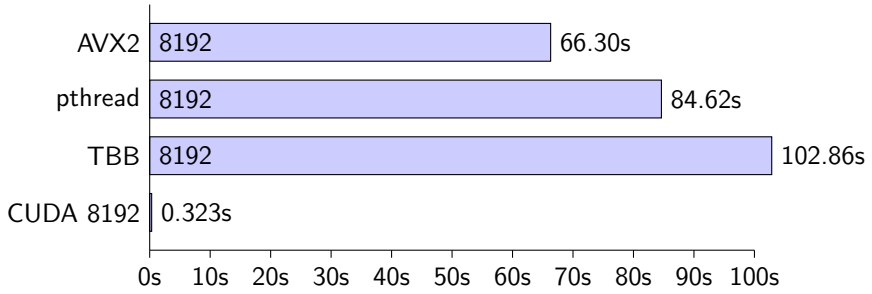


Figure 2: Machine 2 Matrix Multiplication

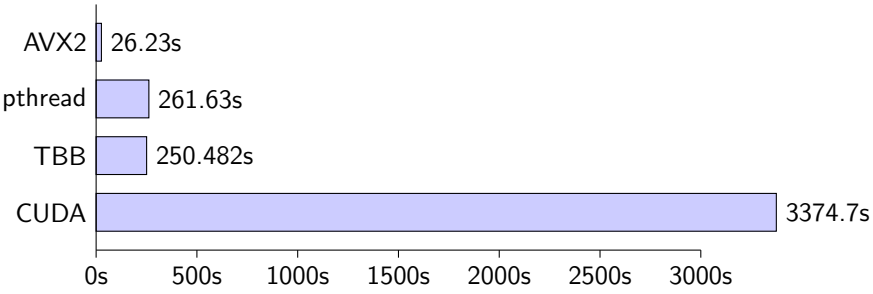


Figure 3: Machine 1 K-means

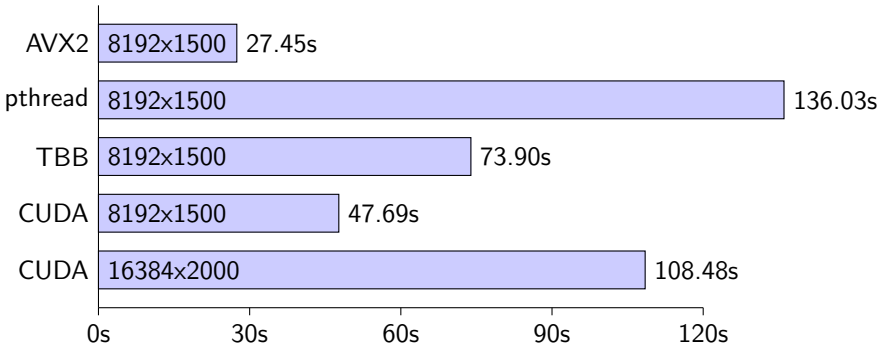


Figure 4: Machine 2 K-means

# References

[1] Hochberg, Robert. *"Matrix Multiplication with CUDA A Basic Introduction to the CUDA Programming Model." N.p.*, 11 Aug. 2012. Web. 15 May 2016.