2015 Computer Architecture Final Project Report

Student 1: 0010129 李冠瑜 電子系

Student 2: 0010780 廖健翔 電機系

A. Overview

We accelerated principal component analysis with CUDA. We compared the performance of 4 different kernels with different data sizes and different numbers of threads. Besides 4 basic kernels, we also use shared_memory to optimize the kernels. To gain further insight of the kernels, we used NVIDIA Visual Profiler to analysis the kernels.

B. Principal Component Analysis (PCA)

PCA is an useful machine learning algorithm used to reduce dimensions of data and extract dominant factor of data. PCA could find the axes with maximum variances where the data is most spread. Face recognition is one of the common application of PCA.

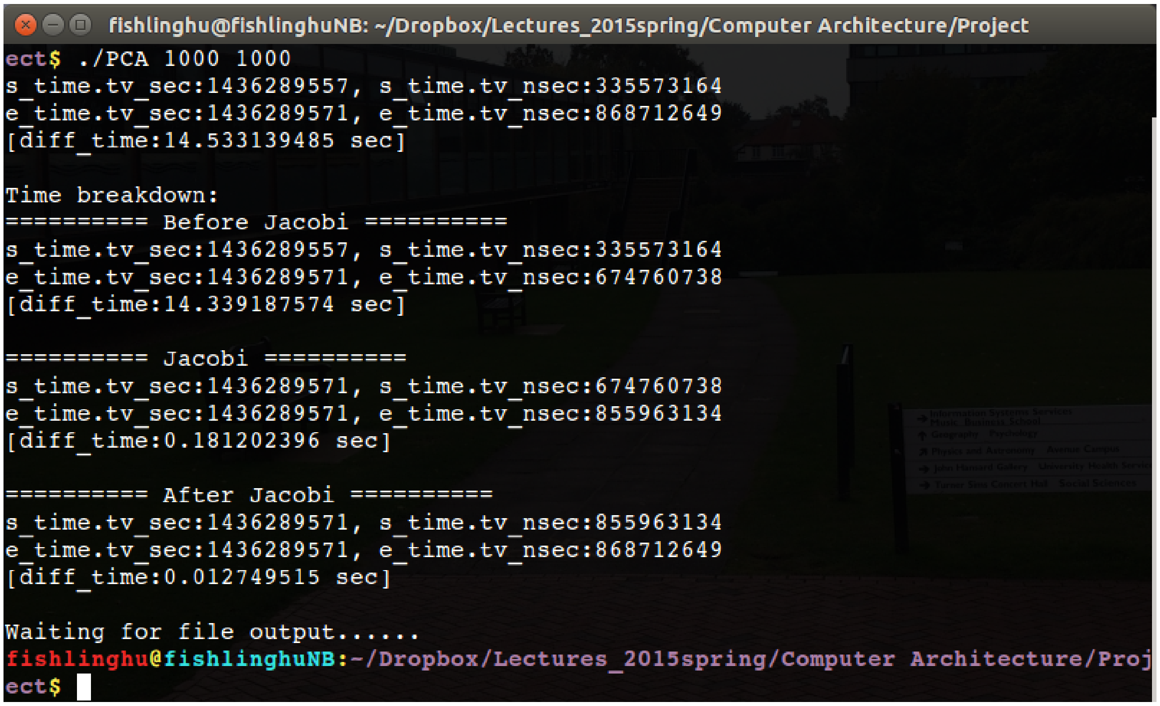
PCA step by step:

a) Randomly generate n sample data sets of dimension d: (n x d) matrix **X**.
b) Compute the mean of each dimension: (1 x d) matrix **M**.
c) Compute the scatter matrix: (d x d) matrix **S**.

   $\mathbf{S} = \sum(\mathbf{X}^k-\mathbf{M})(\mathbf{X}^k-\mathbf{M})^T$
d) Compute eigenvectors and corresponding eigenvalues: Jacobi Method
e) Sort the eigenvectors by decreasing eigenvalues.
f) Choose k eigenvectors with the largest eigenvalues. Then you could reduce the d dimensional data to k dimensional. (d x k) matrix **W**.
g) Transform the samples onto the new subspace.

   $\mathbf{Result} = \mathbf{W}^T \times \mathbf{X}$

C. Profiling of the Original Program



```
fishlinghu@fishlinghuNB: ~/Dropbox/Lectures_2015spring/Computer Architecture/Project
ect$ ./PCA 1000 1000
s_time.tv_sec:1436289557, s_time.tv_nsec:335573164
e_time.tv_sec:1436289571, e_time.tv_nsec:868712649
[diff_time:14.533139485 sec]

Time breakdown:
========== Before Jacobi ==========
s_time.tv_sec:1436289557, s_time.tv_nsec:335573164
e_time.tv_sec:1436289571, e_time.tv_nsec:674760738
[diff_time:14.339187574 sec]

========== Jacobi ==========
s_time.tv_sec:1436289571, s_time.tv_nsec:674760738
e_time.tv_sec:1436289571, e_time.tv_nsec:855963134
[diff_time:0.181202396 sec]

========== After Jacobi ==========
s_time.tv_sec:1436289571, s_time.tv_nsec:855963134
e_time.tv_sec:1436289571, e_time.tv_nsec:868712649
[diff_time:0.012749515 sec]

Waiting for file output......
fishlinghu@fishlinghuNB:~/Dropbox/Lectures_2015spring/Computer Architecture/Proj
ect$
```

As you can see, the jobs before Jacobi Method require over 95% of execution time. After looking carefully into the codes, we found a 3_layer for loop, which is used to compute the scatter matrix and is the most time-consuming. Therefore, we decided to accelerate that part with CUDA.

D. 4 Ways of Computing Scatter Matrix with CUDA

a) Type A: Each block computes a row. N threads compute element of the row sequentially.

Block 0:	Thread 0	Thread 1	Thread 2
Block 1:	Thread 0	Thread 1	Thread 2
Block 2:	Thread 0	Thread 1	Thread 2

…

…

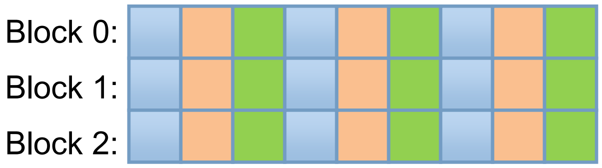
…

…

…

…

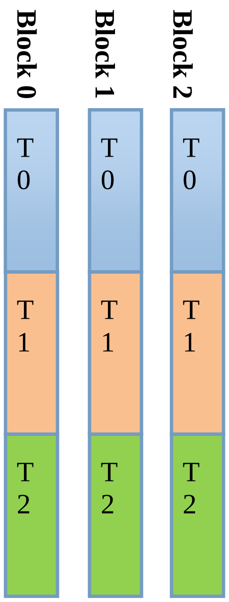
b) Type B: Each block computes a row. N threads compute element of the row interleavingly.



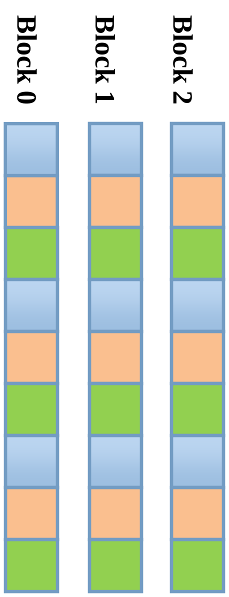
…

…

c) Type C: Each block computes a column. N threads compute element of the column sequentially.



d) Type D: Each block computes a column. N threads compute element of the column interleavingly.



E. Implementation

a) Execution command: ./PCA_CUDA n dim

The program will generate “n” sets of data of dimension “dim”.

b) Compute the scatter matrix

The following codes are used to compute the scatter matrix sequentially:

```
for( I = 0 ~ n-1 )
    {
    for( j = 0 ~ dim-1 )
        {
        for( k = 0 ~ dim-1 )
            scatter[j*dim+k] = scatter[j*dim+k] + (samples[i][j] - mean[j]) * (samples[i][k]
- mean[k]);
        }
    }
```

As you can see, to compute an element of scatter matrix scatter[j][k], we need access mean[j], mean[k], and the column j, k of samples matrix.

c) Check the result with the command: cmp result.txt Golden_Ans.txt

d) Debug CUDA code

I detect CUDA error with the following codes:

```
cudaError_t err = CUDA_API;
if (err != cudaSuccess)
    printf( cudaGetErrorString(err) );
```

F. Experiment Environment

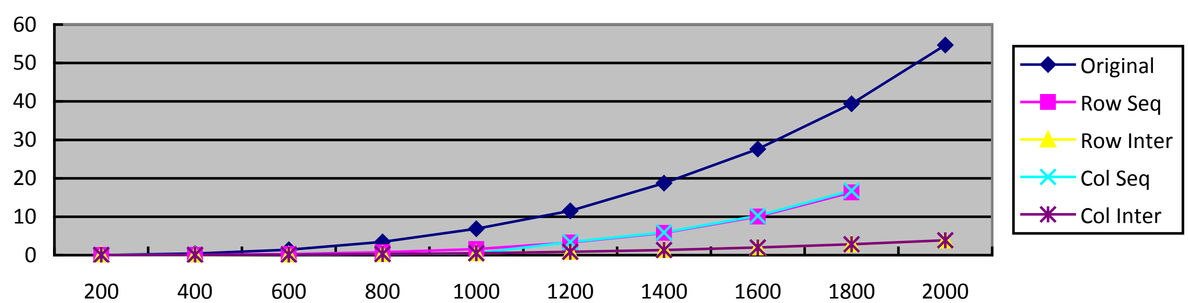
a) CPU: Intel i5-3230m
b) GPU: NVIDIA GT730m / 2G
c) Memory: 8G
d) OS: Ubuntu 14.04 LTS
e) CUDA 6.5

G. Kernel Performance with Different Number of Threads and Different Data Sizes

a) Different data sizes

The numbers in the table are time (in seconds) required to compute the scatter matrix. # of blocks = n = dim; # of threads = 128.

n, dim =	200	400	600	800	1000
Original	0.055	0.424	1.423	3.47	6.866
Row Seq	0.033	0.093	0.278	0.731	1.585
Row Inter	0.032	0.059	0.139	0.277	0.52
Col Seq	0.025	0.086	0.282	0.736	1.637
Col Inter	0.032	0.069	0.141	0.28	0.5
n, dim =	1200	1400	1600	1800	2000
Original	11.544	18.74	27.604	39.376	54.68
Row Seq	3.281	5.777	10.04	16.313	-
Row Inter	0.862	1.34	1.979	2.831	3.836
Col Seq	3.394	5.919	10.263	16.78	-
Col Inter	0.877	1.338	1.998	2.845	3.885



Discussion:

The kernels that have threads compute the scatter matrix interleavingly have the best performance. There is not much difference between the performance of row-majored kernel and column-majored kernel. This is probably because each kernel only accesses the element of the scatter matrix once when computing, so how the kernel accesses the element of the scatter matrix would not affect the performance much.

There are two blanks in the table in case n=dim=2000. This is because the kernel took too much time and was terminated by the OS.

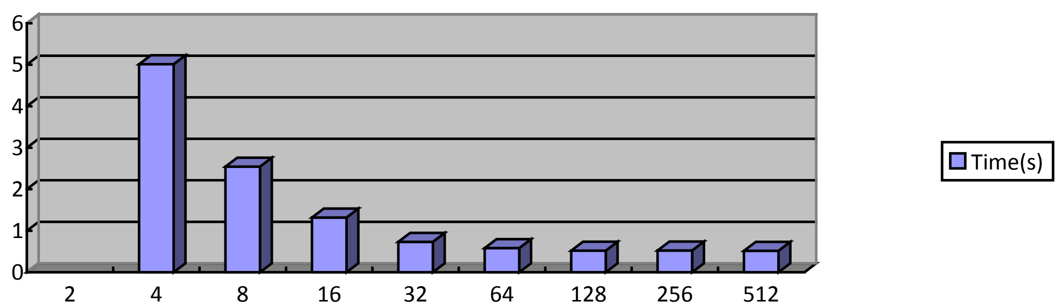
Since the GPU is used to display, the user's screen will be hanged when the GPU is doing other works, such as executing the CUDA kernel. Therefore, there is a time limit for the GPU to do other works to prevent the user's screen from being hanged for too long.

b) Different # of threads

The numbers in the table are time (in seconds) required to compute the scatter matrix. The kernel is column majored and interleaving (type d).

# of blocks = n = dim = 1000;

# of T	2	4	8	16	32	64	128	256	512
Time(s)	-	5.013	2.546	1.32	0.732	0.584	0.522	0.526	0.518



When the number of threads is small, we could improve the performance by increasing the number of threads. After the number of threads exceeds 128, there is not much improvement. Since each streaming processor could only execute a limited number of threads at the same time, assigning number of threads that exceeds that limit would not improve the performance.

H. Optimized Kernel with Shared-Memory

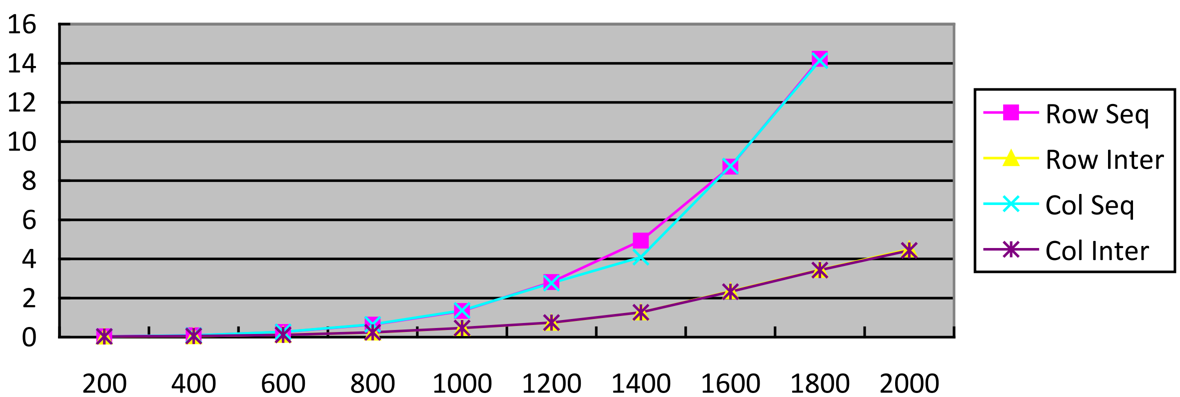
We found that a column of samples matrix is used by every thread in the same block. Therefore, loading the column into the shared-memory of CUDA core is a good way to reduce cache miss and improve the performance of the kernel. The following table shows the performance of different kernels that used shared-memory.

The numbers in the table are time (in seconds) required to compute the scatter matrix.

# of blocks = n = dim; # of threads = 128.

n, dim =	200	400	600	800	1000
Row Seq	0.04	0.086	0.263	0.64	1.334
Row Inter	0.036	0.05	0.122	0.237	0.467
Col Seq	0.031	0.088	0.268	0.652	1.364
Col Inter	0.033	0.059	0.118	0.242	0.464
n, dim =	1200	1400	1600	1800	2000
Row Seq	2.811	4.935	8.71	14.221	-
Row Inter	0.74	1.265	2.347	3.446	4.491
Col Seq	2.766	4.088	8.741	14.137	-
Col Inter	0.745	1.27	2.323	3.426	4.434

Compared to the original kernels, the memory-shared kernels are a little faster.



I. Profiling Kernels

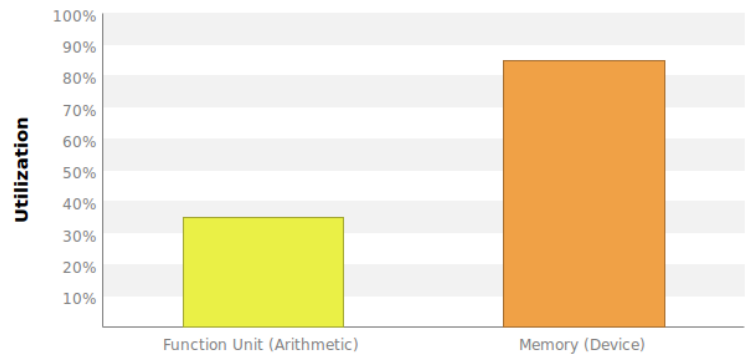
We use NVIDIA Visual Profiler to compare three kernels that have different performance.

a) Row-majored, sequential, without shared-memory

Kernel execution time: 426.98ms

i Kernel Performance Is Bound By Memory Bandwidth

For device "GeForce GT 730M" the kernel's compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by the memory system. For this kernel the limiting factor in the memory system is the bandwidth of the Device memory.



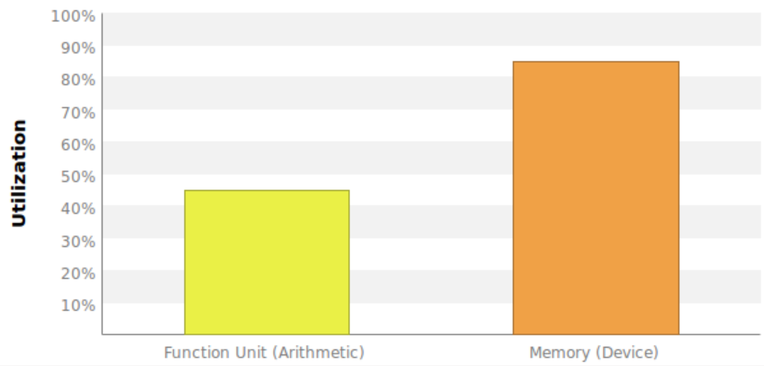
	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Global Loads	147446600	26.811 GB/s	
Global Stores	193550	36.979 MB/s	
Atomic	0	0 B/s	
L1/Shared Total	147640150	26.848 GB/s	Idle Low Medium High Max
L2 Cache			
L1 Reads	355266800	26.811 GB/s	
L1 Writes	490000	36.979 MB/s	
Texture Reads	0	0 B/s	
Atomic	0	0 B/s	
Total	355756800	26.848 GB/s	Idle Low Medium High Max
Texture Cache			
Reads	0	0 B/s	Idle Low Medium High Max
Device Memory			
Reads	325769264	24.585 GB/s	
Writes	489965	36.976 MB/s	
Total	326259229	24.622 GB/s	Idle Low Medium High Max

b) Row-majored, sequential, with shared-memory

Kernel execution time: 373.028ms

i Kernel Performance Is Bound By Memory Bandwidth

For device "GeForce GT 730M" the kernel's compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by the memory system. For this kernel the limiting factor in the memory system is the bandwidth of the Device memory.



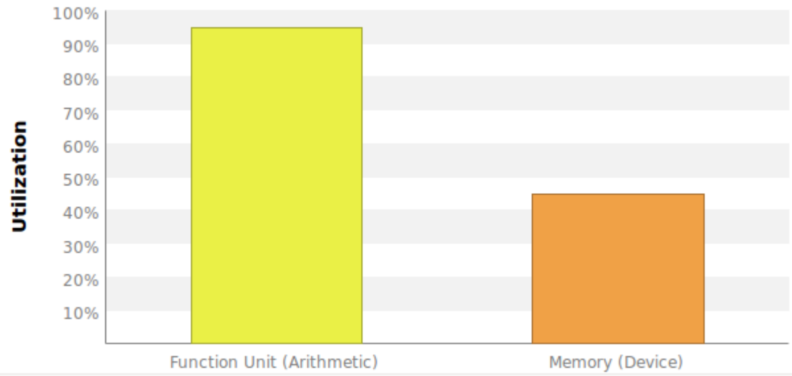
	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	11760000	8.114 GB/s	
Shared Stores	16100	11.109 MB/s	
Global Loads	136176600	29.669 GB/s	
Global Stores	193550	42.261 MB/s	
Atomic	0	0 B/s	
L1/Shared Total	148146250	37.836 GB/s	Idle Low Medium High Max
L2 Cache			
L1 Reads	343996800	29.669 GB/s	
L1 Writes	490000	42.261 MB/s	
Texture Reads	0	0 B/s	
Atomic	0	0 B/s	
Total	344486800	29.711 GB/s	Idle Low Medium High Max
Texture Cache			
Reads	0	0 B/s	Idle Low Medium High Max
Device Memory			
Reads	287934434	24.834 GB/s	
Writes	490894	42.338 MB/s	
Total	288425328	24.876 GB/s	Idle Low Medium High Max

c) Row-majored, interleaving, without shared-memory

Kernel execution time: 164.976ms

i Kernel Performance Is Bound By Compute

For device "GeForce GT 730M" the kernel's memory utilization is significantly lower than its compute utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by computation on the SMs.



	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Global Loads	40348700	18.705 GB/s	
Global Stores	42175	23.703 MB/s	
Atomic	0	0 B/s	
L1/Shared Total	40390875	18.728 GB/s	Idle Low Medium High Max
L2 Cache			
L1 Reads	96667900	18.705 GB/s	
L1 Writes	122500	23.703 MB/s	
Texture Reads	0	0 B/s	
Atomic	0	0 B/s	
Total	96790400	18.728 GB/s	Idle Low Medium High Max
Texture Cache			
Reads	0	0 B/s	Idle Low Medium High Max
Device Memory			
Reads	73900721	14.299 GB/s	
Writes	122528	23.708 MB/s	
Total	74023249	14.323 GB/s	Idle Low Medium High Max

d) Discussion

The third kernel has the best performance. Its performance is bound by compute. This means that the kernel spends most of the time computing. The overhead of data transferring or memory accessing is relatively small in this kernel. Therefore, we do not have much to do to optimize the kernel, since the kernel already uses the hardware efficiently. To further improve the performance, we could only modify the algorithm or reduce the computation workloads with some techniques, such as loop unrolling.

As for the first and second kernels, their performance is bound by memory bandwidth. You could see this clearly in the analysis of memory bandwidth. The third kernel has significantly less times of memory access than that of the first and second kernel. The second kernel has slightly less times of memory access to L2 cache and device memory than that of the first kernel. This is because the second kernel used the shared-memory technique to reduce cache miss in L1 cache.

J. Reference

a) http://blog.csdn.net/zhouxuguang236/article/details/40212143
b) http://sebastianraschka.com/Articles/2014_pca_step_by_step.html#drop_labels
c) CUDA Programming: A Developer's Guide to Parallel Computing with GPUs, by Shane Cook