

**CE 4011 Lab 3: CUDA Programming and Optimizations**

Submitted By: Chia Ming En, Eric Beckmann

Matriculation Numbers: U1122547B, N1403720B

School of Computer Engineering

13/11/2014

# 1. Introduction

The goal of the third lab is to gain experience writing programs using the Massively Parallel Programming techniques. In particular, we are to use CUDA to parallelize an algorithm by allowing it to run on multiple cores in the GPU. The algorithm we were given to parallelize is Floyd’s algorithm, which computes the all-pairs shortest-path matrix given an input mapping matrix. Floyd’s algorithm runs in O(N3) time. Our challenge was to find a method to use CUDA API to implement a basic kernel for the all pairs shortest path problem. Lastly, we will optimize our code using coalesced memory accesses and shared memory.

# 2. Benchmark the basic kernel

See Appendix A for the source code of the basic kernel. The premise of the basic kernel is to divide the graph into squares, each with a dimension of BLOCK\_SIZE by BLOCK\_SIZE. This results in a grid of size N/BLOCK\_SIZE by N/BLOCK\_SIZE. Each individual block contains BLOCK\_SIZE threads, and each thread is assigned to process a row of the square. Since threads in the same warp will read data points with scattered indexes, this is a non-coalesced algorithm.

Below is the comparison between GPU kernel and sequential CPU version of Floyd’s algorithm.

**Analysis for Different Problem Size N**

|  |  |  |
| --- | --- | --- |
| Problem Size N | Sequential algorithm(CPU) | GPU algorithm |
| 10 | 0.01ms | 23ms |
| 100 | 11ms | 23ms |
| 500 | 891ms | 40ms |
| 1000 | 6656ms | 145ms |
| 2000 | 53057ms | 527ms |

From the graph and the table, we can see that for small problem size, the sequential algorithm that run on CPU is faster than the GPU version of the algorithm. The reason is that for the GPU version of the algorithm, the overheads incur to initialize the kernel and transfer of matrix between CPU memory and GPU global memory actually takes more time than the time the sequential algorithm takes to compute the result.

However, for large problem size, the GPU version of the algorithm takes much shorter time to execute as GPU are able to utilize its large number of cores to solve the problem simultaneously.

**Analysis for Different Thread Block Size**

|  |  |  |
| --- | --- | --- |
| Thread Block Size | Achieved Occupancy | Execution Time |
| 4x4 (14) | 23% | 685ms |
| 8x8 (64) | 45% | 185ms |
| 12x8 (128) | 88% | 112ms |
| 16x16 (256) | 87% | 115ms |
| 32x32 (1024) | 81% | 147ms |

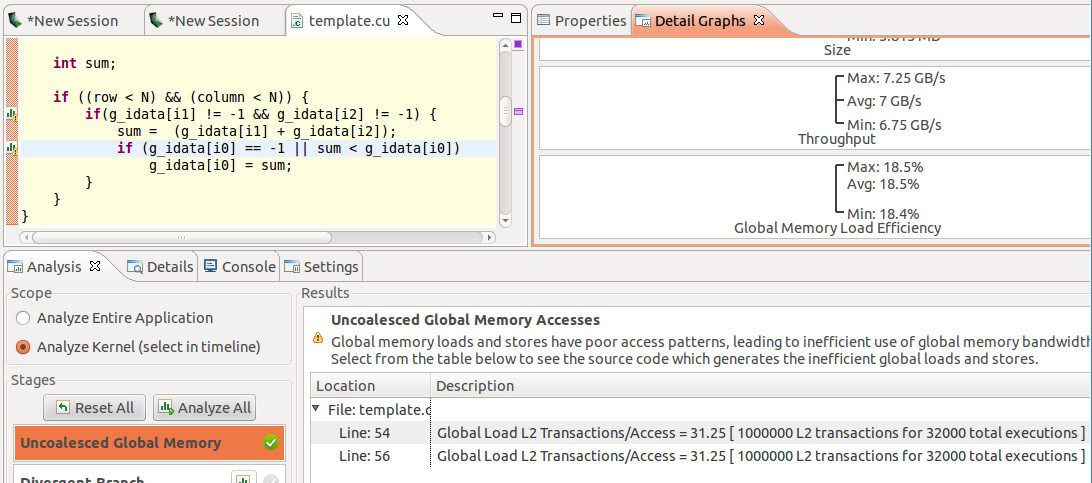
From the table, we can see that by achieving higher occupancy, we are able to reduce the execution time as there will be more active wraps/threads per multiprocessor to keep the GPU busy.

We have also observed that to achieve higher occupancy, the number of threads per block should be larger than 64. Anything lesser than 64 threads per block results in too little active wrap in each multiprocessor and therefore low occupancy and longer execution time.

# 3. Optimization

**Coalesced Accesses**

Using Nvidia Visual Profile, we profile our basic kernel and found out that accesses to global memory are uncoalesced.



From the screenshot above, we can see that the global memory load efficiency is only 18.4%. After some analysis, found out that the block size that we have chosen result in misaligned access pattern.

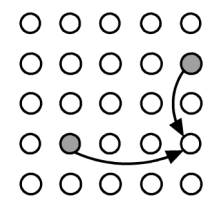
The source code for the coalesced accesses kernel can be found in Appendix B. This kernel is the same as the basic kernel with one difference: instead of being oriented vertically, the threads in a block are oriented horizontally in order to read the data squares row by row rather than column by column. Using this method, threads in the same warp read adjacent data in memory, so this method is coalesced. Below are the results of comparing the coalesced kernel against the basic kernel.

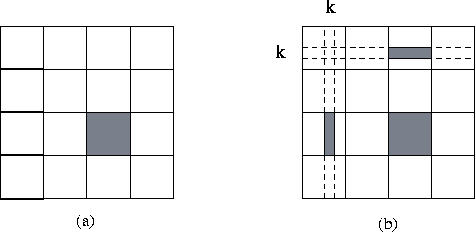
|  |  |  |
| --- | --- | --- |
| Problem Size N | Basic Kernel | Coalesced Kernel |
| 500 | 25ms | 87ms |
| 1000 | 544ms | 148ms |
| 1500 | 1730ms | 412ms |
| 2000 | 4042ms | 540ms |
| 2500 | 7831ms | 1124ms |

From the graph, we can see that the coalesced kernel is more scalable and takes shorter time to execute compare to the basic kernel.

**Shared Memory**

We can improve the performance by reducing the number of global memory accesses by caching commonly used data into shared memory.

The figure on the left shows the data depedency for the given node for k = 2. Therefore, as the lower figure shows, in our basic kernel where the vertical block of threads processes nodes column by column, all the threads share a dependency on a section of the kth row of data. This data can be collaboratively loaded into shared memory by the blocks before the actual processing begins in order to reduce global memory accesses. The source code for our kernel which does just that can be found in Appendix C.



Below are results comparing the shared memory kernel to the basic kernel.

|  |  |  |
| --- | --- | --- |
| Problem Size N | Non-Shared Memory Kernel | Shared Memory Kernel |
| 256 | 6143 | 2079 |
| 512 | 24575 | 8319 |
| 1024 | 98302 | 33279 |
| 1536 | 221180 | 74878 |
| 2048 | 393212 | 133118 |

From the graph and table above, we can see that the number of global memory requests reduce significantly after adding the shared memory.

**Comparison between Basic, Coalesced, Shared Memory and Full Optimization kernel**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Problem Size N | Basic | Coalesced | Shared Memory | Full optimization |
| 256 | 99 | 49 | 65 | 37 |
| 512 | 182 | 87 | 111 | 58 |
| 1024 | 1467 | 145 | 808 | 98 |
| 1536 | 9267 | 390 | 2970 | 243 |
| 2048 | 11703 | 538 | 6688 | 351 |

From the graph and table above, we can see that by performing optimization to our kernel, we are able speed up the execution time up to 30 times compared to the basic kernel (N=2048).

# 4. Conclusion

In this lab, we were successfully able to parallelize Floyd’s algorithm using CUDA. We have also optimized the kernel by ensuring memory accesses are coalesced and loading commonly used data into the shared memory to reduce the number of memory accesses. Our Full Optimized version of the Floyd’s algorithm are able to reduce the execution time by factor of 30 when compared to basic kernel and 160 when compared to sequential CPU version (N=2048).

# 5. Appendix A

**Basic Kernel Code**

\_\_global\_\_ void

compute\_k\_iter(int \*g\_idata, int k, int N, int block\_size)

{

int column = blockIdx.x \* block\_size + threadIdx.x;//starting column

int row = blockIdx.y \* blockDim.y + threadIdx.y;

int i0 = row \* N + column;

int i1 = row \* N + k;

int i2 = k \* N + column;

int i;

int sum;

int run\_range = (N <=block\_size) ? 1 : N-(N%block\_size) + 1;

int loop\_count = min(block\_size, N-column);

//int loop\_count = (N < block\_size) ? N : block\_size;

if ((row < N) && (column < run\_range)) {

for(i = 0; i < loop\_count ; i++){

if(g\_idata[i1] != -1 && g\_idata[i2] != -1) {

sum = (g\_idata[i1] + g\_idata[i2]);

if (g\_idata[i0] == -1 || sum < g\_idata[i0])

g\_idata[i0] = sum;

}

i0++;

i2++;

}

}

}

////////////////////////////////////////////////////////////////////////////////

// Program main

////////////////////////////////////////////////////////////////////////////////

int

main(int argc, char \*\*argv)

{

//GPU initialization as well as sequential result computed up here

////……………………..

/////……………………………….

/////////………………….

////////////

////////////////

// Compute GPU parallel result

// allocate host memory

int mem\_size = sizeof(int) \* N \* N;

int \*h\_idata = (int \*) malloc(mem\_size);

// initalize the memory

memcpy(h\_idata, mat, sizeof(int)\*N\*N);

// allocate device memory

int \*d\_idata;

checkCudaErrors(cudaMalloc((void \*\*) &d\_idata, mem\_size));

// copy host memory to device

checkCudaErrors(cudaMemcpy(d\_idata, h\_idata, mem\_size,

cudaMemcpyHostToDevice));

int block\_size = 128;

// Start a 2D thread block

dim3 dimGrid(ceil(N/(float)block\_size), ceil(N/(float)block\_size), 1);

dim3 dimBlock(1, block\_size, 1);

for(int k = 0; k < N; k++) {

// execute the kernel

compute\_k\_iter<<< dimGrid, dimBlock >>>(d\_idata, k, N, block\_size);

}

// check if kernel execution generated and error

getLastCudaError("Kernel execution failed");

// copy result from device to host

checkCudaErrors(cudaMemcpy(h\_idata, d\_idata, mem\_size,

cudaMemcpyDeviceToHost));

// cleanup memory

checkCudaErrors(cudaFree(d\_idata));

//Down here:

//compare your result with reference result

//also do clean up for GPU

///////////////////////

//////////////////////////

//////////////////////////

/////////////////////////

}

# 6. Appendix B

**Coalesced Kernel Code**

\_\_global\_\_ void

compute\_k\_iter(int \*g\_idata, int k, int N, int block\_size)

{

int column = blockIdx.x \* blockDim.x + threadIdx.x;

int row = blockIdx.y \* block\_size + threadIdx.y;

int i0 = row \* N + column;

int i1 = row \* N + k;

int i2 = k \* N + column;

int i;

int sum;

int run\_range = (N <= block\_size) ? 1 : N - (N%block\_size) + 1;

int loop\_count = min(block\_size, N-row);

if ((row < run\_range) && (column < N)) {

for(i = 0; i < loop\_count; i++){

if(g\_idata[i1] != -1 && g\_idata[i2] != -1) {

sum = (g\_idata[i1] + g\_idata[i2]);

if (g\_idata[i0] == -1 || sum < g\_idata[i0])

g\_idata[i0] = sum;

}

i0 += N;

i1 +=N;

}

}

}

////////////////////////////////////////////////////////////////////////////////

// Program main

////////////////////////////////////////////////////////////////////////////////

int

main(int argc, char \*\*argv)

{

//GPU initialization as well as sequential result computed up here

////……………………..

/////……………………………….

/////////………………….

////////////

////////////////

// Compute GPU parallel result

// allocate host memory

int mem\_size = sizeof(int) \* N \* N;

int \*h\_idata = (int \*) malloc(mem\_size);

// initalize the memory

memcpy(h\_idata, mat, sizeof(int)\*N\*N);

// allocate device memory

int \*d\_idata;

checkCudaErrors(cudaMalloc((void \*\*) &d\_idata, mem\_size));

// copy host memory to device

checkCudaErrors(cudaMemcpy(d\_idata, h\_idata, mem\_size,

cudaMemcpyHostToDevice));

int block\_size = 128;

// Start a 2D thread block

dim3 dimGrid(ceil(N/(float)block\_size) , ceil(N/(float)block\_size), 1);

dim3 dimBlock(block\_size,1, 1);

for(int k = 0; k < N; k++) {

// execute the kernel

compute\_k\_iter<<< dimGrid, dimBlock >>>(d\_idata, k, N, block\_size);

}

// check if kernel execution generated and error

getLastCudaError("Kernel execution failed");

// copy result from device to host

checkCudaErrors(cudaMemcpy(h\_idata, d\_idata, mem\_size,

cudaMemcpyDeviceToHost));

// cleanup memory

checkCudaErrors(cudaFree(d\_idata));

//Down here:

//compare your result with reference result

//also do clean up for GPU

///////////////////////

//////////////////////////

//////////////////////////

/////////////////////////

}

# 7. Appendix C

**Shared Memory Kernel Code**

\_\_global\_\_ void

compute\_k\_iter(int \*g\_idata, int k, int N)

{

\_\_shared\_\_ int from\_k[BLOCK\_SIZE];

int to\_k;

int column = blockIdx.x \* BLOCK\_SIZE + threadIdx.x;//starting column

int row = blockIdx.y \* blockDim.y + threadIdx.y;

int i0 = row \* N + column;

int i1 = row \* N + k;

int i2 = k \* N + column;

int i;

int sum;

int run\_range = (N <=BLOCK\_SIZE) ? 1 : N-(N%BLOCK\_SIZE) + 1;

int loop\_count = min(BLOCK\_SIZE, N-column);

//int loop\_count = (N < block\_size) ? N : block\_size;

if((row < N) && (column < run\_range)){

from\_k[threadIdx.y] = g\_idata[i2 + threadIdx.y];

\_\_syncthreads();

to\_k = g\_idata[i1];

for(i = 0; i < loop\_count ; i++){

if(to\_k != -1 && from\_k[i] != -1) {

sum = (to\_k + from\_k[i]);

if (g\_idata[i0] == -1 || sum < g\_idata[i0])

g\_idata[i0] = sum;

}

i0++;

}

}

}

**//**The main function is equivalent to the basic kernel main function, so it is not shown again here.

# 8. Appendix D

**Full Optimized Kernel Code**

\_\_global\_\_ void

compute\_k\_iter**(**int **\***g\_idata**,** int k**,** int N**)**

**{**

// Allocate space for k column nodes in shared memory

\_\_shared\_\_ int to\_k**[**BLOCK\_SIZE**];**

int from\_k**;**

int column **=** blockIdx**.**x **\*** blockDim**.**x **+** threadIdx**.**x**;**

int row **=** blockIdx**.**y **\*** BLOCK\_SIZE **+** threadIdx**.**y**;**

int i0 **=** row **\*** N **+** column**;**

int i1 **=** row **\*** N **+** k**;**

int i2 **=** k **\*** N **+** column**;**

int i**;**

int sum**;**

int factor **=** N**%**BLOCK\_SIZE**;**

**if(**factor **==** 0**)** factor **=** BLOCK\_SIZE**;**

int run\_range **=** **(**N **<=** BLOCK\_SIZE**)** **?** 1 **:** N **-** factor **+** 1**;**

int loop\_count **=** min**(**BLOCK\_SIZE**,** N**-**row**);**

// Collaboratively load k column nodes into shared memory

to\_k**[**threadIdx**.**x**]** **=** g\_idata**[**i1 **+** N**\***threadIdx**.**x**];**

// Wait till all threads finished loading nodes

\_\_syncthreads**();**

**if** **((**row **<** run\_range**)** **&&** **(**column **<** N**))** **{**

from\_k **=** g\_idata**[**i2**];**

**for(**i **=** 0**;** i **<** loop\_count**;** i**++){**

**if(**to\_k**[**i**]** **!=** **-**1 **&&** from\_k **!=** **-**1**)** **{**

sum **=** **(**to\_k**[**i**]** **+** from\_k**);**

**if** **(**g\_idata**[**i0**]** **==** **-**1 **||** sum **<** g\_idata**[**i0**])**

g\_idata**[**i0**]** **=** sum**;**

**}**

i0 **+=** N**;**

**}**

**}**

**//The main function is equivalent to the coalesced access main function, so it is not shown here**