COM2039: Parallel Computing

Lab Session 2

Matrix Multiplication

Contents

1 Introduction

You will develop a basic matrix multiplication algorithm in this lab class. Although it will be using the GPU, a little bit of analysis will show that it will not perform as well as one might expect from its use of a parallelised algorithm.

There are two key learning outcomes from this lab class:

1. Apply what you have learnt so far to flesh out the template for performing matrix multiplication;
2. Perform a quick analysis of your resulting code to understand where there might be any performance bottlenecks.

We will then see how to overcome the performance issue using shared memory next week.

You will need nvcc to compile the file you create. However, this will be managed for you within Nsight Eclipse edition. You should find building and executing a project within Nsight to be a familiar process but in case you need a little guidance, take a look at: http://docs.nvidia.com/cuda/nsight-eclipse-edition-getting-started-guide/#axzz4aa6o7rRj

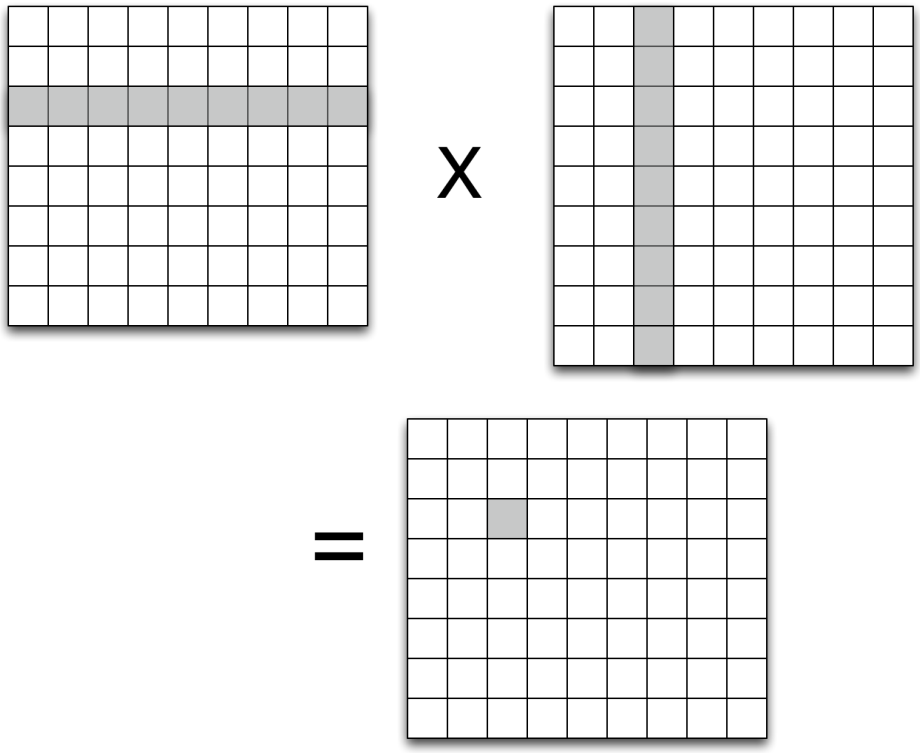
2 Matrix Multiplication

Matrix multiplication is a fundamental operation in linear algebra and required for a very broad range of applications. I will assume familiarity with basic linear algebra but will provide more details in appendices to lab class handouts and lectures in case it is needed. This time round, I will start the lab class with a refresher lecture on matrix multiplication, so the notes below are just for completeness.

Just to quickly summarise, if we have an *n* x *m* matrix A (*n* rows, *m* columns), and a *m* x *p* matrix B, then their product C = AB is an *n* x *p* matrix where each element of C is the inner product of a row from A and a column from B:

$$c_{ji} = \sum_{k=1}^{m} a_{jk} \times b_{ki}$$

for $j = 1, \ldots n$, and $i = 1, \ldots p$.



Firstly, we are going to use the Big Idea, and map threads to elements of the result matrix C. So, each thread is going to implement a Gather, to calculate its corresponding element c_{ij}. Now, to make our program scale for large matrices (and this is why we are doing this; we want to solve BIG problems), we are going to need to tile C with multiple blocks.

3 Open a new project in Nsight

Once you have Nsight running, open a new Cuda C/C++ project. Select new runtime project and add your name to the copyright information. Then in the next window it should detect the appropriate compute capability automatically. Leave that as selected and finish.

As you saw last week, Nsight will create an example file for you in that new project. It is simplest to just delete that file and start all over. Create a new source file. Name it something relevant and make sure it has the .cu extension.

4 The header information

You may want to write the following in a separate .h file, but just to keep things simple I have just left it in the top of the .cu file. We need to include some "includes" and then define the struct that will be used to represent our matrices. Enter the following into your new file:

```
#include <stdio.h>
#include <stdlib.h>

#include <cuda.h>
#define BLOCK_SIZE 16

// Matrices are stored in row-major order
typedef struct {
  int width;
  int height;
  float* elements;
} Matrix;
```

I strongly recommend that after you have made each change to the file you rebuild the project. You can do this by clicking the hammer icon towards the left side of the menu bar in NSight. If you don't do this, you can pile up a load of small errors that can be hard to disentangle.

You should see a warning that no main() method is defined. So fix this with a placeholder main() for the moment. Leave some space between it and the above (we will leave the main method at the bottom of the file), and then insert the following:

```
int main(void) {

}
```

5 THE MATRIX MULTIPLICATION KERNEL

You should understand the code in the following from the lecture. If not, please make sure you do before you move on to the next section!!!

Now, paste the following Kernel definition into the file, below the header information but above the main() method:

```
__global__ void MatrixMultKern(const Matrix A, const Matrix B, const Matrix C) {
 // Calculate the column index of C and B
 int col = blockIdx.x * blockDim.x + threadIdx.x;
 // Calculate the row index of C and of A
 int row = blockIdx.y * blockDim.y + threadIdx.y;

 if ((row < A.height) && (col < B.width)) {
  float Cvalue = 0;
  // each thread computes one element of the block sub-matrix
  for (int k = 0; k < A.width; ++k) {
   Cvalue += A.elements[row * A.width + k] * B.elements[k*B.width + col];
  }
  C.elements[row * C.width + col] = Cvalue;
 }
}
```

Question: What is the Arithmetic Intensity of the above Kernel?

We will worry a bit more about arithmetic intensity in the next lecture but have a little bit of a think about this now. You won't usually perform precise calculations. Instead you look to see what you need to do in general to optimise *arithmetic intensity*, which is defined as:

$$\frac{\#compute\ opps\ per\ thread}{\#memory\ accesses\ per\ thread}$$

You can informally memorise this as:

$$\frac{math}{memory}$$

In order to optimise arithmetic intensity, we will need to:

1. Maximise the number of compute operations performed per thread
2. Minimise the number of memory accesses (from global memory) per thread

You will not be able to do anything about point 2. this week. However, next week we will see how *shared memory* can be used to reduce the number of global memory accesses needed per thread.

Rebuild the project to make sure you have not introduced any errors.

6 THE HOST CODE

Remember that the host needs to launch the Kernel on the device. In addition, the matrices to be multiplied need to be copied from host to device. Finally, the result needs to be copied back to the host and all memory on the device freed up.

Remember also that the memory copies go from host into the global memory of the device. This is the slowest of the three forms of memory on the device (what are the other two?).

The code that follows is a function that can be called on the host to multiply two matrices and instantiate the resulting matrix. Please make sure you understand what is going on here. The matrix C will need to be initialised in some way before we call this function (from the main method, of course).

Please do not copy the following code blindly. If you do not understand what is going on in this, you will have learned nothing from the lab class. So please read it line by line before pasting it into your project.

This function should go into your .cu file below the kernel definition, and above the main() method.

The code is on the next page to keep it together on one page.

N.B. You can now use Cuda's unified memory for this (as you did last week) of course, but I prefer you to manage the memory in the GPU yourself just so that you can see what is actually going on.

```
// Matrix multiplication - Host Code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatrixMult(const Matrix h_A, const Matrix h_B, Matrix h_C)
{
  // Load A and B into device memory
 Matrix d_A;
  d_A.width = h_A.width; d_A.height = h_A.height;
  size_t size = h_A.width * h_A.height * sizeof(float);
  cudaMalloc(&d_A.elements, size);
  cudaMemcpy(d_A.elements, h_A.elements, size, cudaMemcpyHostToDevice);

  Matrix d_B;
  d_B.width = h_B.width; d_B.height = h_B.height;
  size = h_B.width * h_B.height * sizeof(float);
  cudaMalloc(&d_B.elements, size);
  cudaMemcpy(d_B.elements, h_B.elements, size, cudaMemcpyHostToDevice);

  // Allocate C in Device memory
  Matrix d_C;
  d_C.width = h_C.width; d_C.height = h_C.height;
  size = h_C.width * h_C.height * sizeof(float);
  cudaMalloc(&d_C.elements, size);

  // Invoke Kernel
  dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
  dim3 dimGrid(d_B.width / dimBlock.x, d_A.height / dimBlock.y);
  MatrixMultKern<<< dimGrid, dimBlock >>>(d_A, d_B, d_C);

  // Read C from Device to Host
  cudaMemcpy(h_C.elements, d_C.elements, size, cudaMemcpyDeviceToHost);

  // Free Device Memory
  cudaFree(d_A.elements);
  cudaFree(d_B.elements);
  cudaFree(d_C.elements);
}
```

Question: A comment in the above states that this function definition assumes the matrix dimensions are multiples of BLOCK_SIZE. Why is this?

Rebuild the project to make sure you have not introduced any errors.

7 DEFINING THE MAIN() METHOD

Now we need to make this work. We do this through the main() method.

We are going to need to pass in the dimensions of the input arrays, and then generate example matrices of these sizes. Note the modification of the declaration of the main() method to enable it to accept an array of arguments.

Note, this means that when you run the project, you will need use /Run Configuration and set the values of the arguments.

We will input three arguments: the height and width of A, but only the width of B.

Question: Why don't we need to input the height of B?

We will then initialise the .elements properties of all three matrices.

After that, there is a simple generator to put some random(ish) numbers into the elements of A and B.
We then call our host function to multiply the two matrices and finally print out all three matrices for visual checking.
Again, I have put the code (almost!) all together on the next page:

Note:
The main method on the next page requires three argument. You need to enter this from within Nsight. What you do is click on the run arrow and select:
Run Configuration
In the dialogue box that opens, first check that the correct project name is entered. Change this to the correct project if needed.
Then select the Arguments tab.
In the text entry space, enter the arguments with a space separating each:
16 16 16
in this case.
These will now be available within the main method by reading from the argv[] array.
You can also use argc to access the number of arguments. So if you wish, you could put a guard at the beginning of the main() method on the next page to check for the correct number if arguments and print out a warning before terminating if they haven't been provided. Do also note that you will need to check that argc = 4, not 3 as you might expect. This is because argc includes the name of the executable being invoked in its count.

```
int main(int argc, char* argv[]) {
 Matrix A, B, C;
 // Read Dimensions of A and B
 A.height = atoi(argv[1]);
 A.width = atoi(argv[2]);
 B.height = A.width;
 B.width = atoi(argv[3]);

 A.elements = (float*)malloc(A.width * A.height * sizeof(float));
 B.elements = (float*)malloc(B.width * B.height * sizeof(float));
 C.height = A.height;
 C.width = B.width;
 C.elements = (float*)malloc(C.width * C.height * sizeof(float));

 for(int i = 0; i < A.height; i++)
  for(int j = 0; j < A.width; j++)
    A.elements[i*A.width + j] = (float)(rand() % 3);
 for(int i = 0; i < B.height; i++)
  for(int j = 0; j < B.width; j++)
    B.elements[i*B.width + j] = (float)(rand() % 2);
 MatrixMult(A, B, C);

 for(int i = 0; i < A.height; i++){
  for(int j = 0; j < A.width; j++)
    printf("%f ", A.elements[i*A.width + j]);
  printf("\n");
 }
 printf("\n");
 for(int i = 0; i < B.height; i++){
  for(int j = 0; j < B.width; j++)
    printf("%f ", B.elements[i*B.width + j]);
  printf("\n");
 }
 printf("\n");
 for(int i = 0; i < C.height; i++){
  for(int j = 0; j < C.width; j++)
    printf("%f ", C.elements[i*C.width + j]);
```

```
  printf("\n");
 }
 printf("\n");

 return 0;
}
```

Once you have added this and rebuilt the project without errors, you can run the project.
Set the arguments so that you generate two 16 x 16 matrices and then multiply them together.

8 NEXT STEPS

Please make sure you do the following before moving on.

1. Modify the code of Section 7 to remove the restriction that the dimensions of the input matrices must be multiples of the block size.
2. Define some different generators than those in Section 7, so that you have more control over the content of the input matrices. See if you can replicate the two example structures that I gave you in the lecture, but add more if you can. You might like to move the generator to a new function (in fact, you probably should!).
3. Once you have done the above, run some more tests on larger matrices. Before you try this out on Very Big matrices, you might like to modify the printf() statements so that they only print out a small number of elements in the top left corner of each matrix. Do make sure you record timings for these as you will need this information for your coursework, so be systematic about recording time versus matrix size.
4. Be bold with the matrix size and also experiment with the Block Size. Remember that a block can contain up to 1024 threads. So, our 16x16 matrix is pathetically small – go big, really BIG.

N.B. 1024 is the maximum number of threads per block, whatever dimensionality you choose (remember, we can have 1-, 2-, or 3-dimensional blocks of threads. My arithmetic was a bit rubbish in the lecture yesterday. To see what this means for a 2-dimensional block, rewrite 1024 as a power of 2; 2^{10}.

If we want to have the same number of threads in both the x- and y-dimensions, then we need to set a maximum of $2^5 = 32$ threads in each dimension ($2^5 \times 2^5 = 2^{(5+5)} = 2^{10}$).