COM2039: Parallel Computing

Lab Session 3

Matrix Multiplication and Shared Memory

Contents

1 INTRODUCTION

We covered the details of the Kernel for basic matrix multiplication last week. This lab class will modify it to use shared memory to improve the execution time by reducing access to global memory. I walked you through the core structure of the code for the use of shared memory in Wednesday's lecture. We will run the complete code in this week's lab class, together with generating some timings.

2 THE HEADER INFORMATION

I suggest that you create a new CUDA project so that you don't have to change the main() method in last week's work. As before, we will need to include some "includes" and then define the struct that will be used to represent our matrices. But note the slight change to the struct for Matrix, as mentioned in Wednesday's lecture. That is, we have included `stride` as a property of a Matrix.

For our parent matrices, A, B and C, `stride` us going to have the same value as `width`, so it is not adding value in those cases. However, we will need it for the sub-matrices As, Bs and Cs, as in each case it will need to be assigned the value of the width of the respective parent matrices. Check pages 36, 37 and 37 from Wednesday's notes in SurreyLearn to make sure you understand why!

```
#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;
 int stride;
} Matrix;

__global__ void MultKernShared(const Matrix A, const Matrix B, Matrix C);
```

As always, I strongly recommend that after you have made each change to the files that 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.

3 SOME UTILITY FUNCTIONS

Now, we can start to build up the source code file. You can delete the content of the file that Nsight created for you and replace it with the code that follows.

Referring back to the lecture, in order to manage the usage of the limitation of shared memory (**Question:** What is the limitation?), we need to tile the matrices with sub-matrices where each sub-matrix of the result is mapped onto a block of threads.

As a result of this, we have slightly more complex formulae to elements within sub-matrices, and a little bit of work to do to find the address of a sub-matrix. If you refer back to the slide on "stride" in Lecture 8, the calculations in the `GetElement()` and `SetElement()` functions should be clear. The code for `GetSubMatrix()` is also quite straightforward, but remember that "`row`" is in units of *Blocks*, so we need to first multiply it by `BLOCK_SIZE` to convert into units of matrix *rows* before multiplying it by `A.stride` in order to find the number of rows we need to count down before stepping across columns to find the address of the start of our sub-matrix[1].

Now add the following code to your .cu source file:

[1] It might have been sensible to change the variable name to "blockRows" to make this clearer, but I have stayed with the variable names of the CUDA Programming Guide for ease of reference.

```
#include “<<YourHeaderFileName>>.h”
// For simplicity you could just put all the content into this file instead of having a separate .h file
// Now, we are going to have to define some utilities to get and set
// elements in matrices and to get sub-matrices. Please refer to Lecture 13
// to make sure you understand why we need to use the "stride" property to
// walk down the column of a sub-matrix

// Get a matrix element
__device__ float GetElement(const Matrix A, int row, int col) {
 return A.elements[row * A.stride + col];
}

// Set a matrix element
__device__ void SetElement(Matrix A, int row, int col, float value) {
 A.elements[row * A.stride + col] = value;
}

// Get the BLOCK_SIZExBLOCK_SIZE sub-matrix Asub of A that is
// located col sub-matrices to the right and row sub-matrices down
// from the upper-left corner of A
// Note, we are not doing any copying here - just finding the address of the
// start of the sub-matrix we are interested in
__device__ Matrix GetSubMatrix(Matrix A, int row, int col) {
 Matrix Asub;
 Asub.width = BLOCK_SIZE;
 Asub.height = BLOCK_SIZE;
 Asub.stride = A.stride;
 Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row + BLOCK_SIZE * col];
 return Asub;
}
```

When you build this (do it!), you will 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) {


}
```

4 __device__ versus __global__

You will have noticed that the function declarations in the previous section are all declared as __device__. This means that they will (not surprisingly) be executed on the device, but also that they can only be called from functions executing on the device; that is, either a __global__ function or another __device__ function. This contrasts with a function that is declared as __global__ (also known as a kernel) which must be called from the host but will execute on the device.

Note that calling a __device__ function is just like calling a normal C function (except it can only be called from the device, of course). That is, the function invocation is not wrapped in the foo<<< dimGrid, dimBlock >>>({param list}) structure.

Note also that with current compute capabilities, although a __device__ function can invoke another __device__ function, it cannot invoke itself (i.e. recursive function calls are not currently supported). This may change, but we will cover a work around for this in a later lecture.

5 THE MATRIX MULTIPLICATION KERNEL

The Kernel code was discussed in detail in Lecture 8, so I provide it with little description here. But, as always, please make sure you understand how it works before proceeding. 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 function definitions of Section 4, but above the main() method:

```
// Now we have all the device functions we need to define the matrix multiplication
// Kernel. This is going to be called from the host by MatMul()
__global__ void MultSharedKernel(Matrix A, Matrix B, Matrix C) {
  // Identify the Block row and column
  int blockRow = blockIdx.y;
  int blockCol = blockIdx.x;

  // Each thread block computes one sub-matrix Csub of C
  Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

  // Each thread computes one element of Csub
  // by accumulating results into Cvalue
  float Cvalue = 0;

  // Now find the row and column of the element within Csub
  // that this thread is going to calculate
  int row = threadIdx.y;
  int col = threadIdx.x;

  // Loop over all the sub-matrices of A and B that are
  // required to compute Csub
  for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {

    // Get sub-matrix Asub of A
    Matrix Asub = GetSubMatrix(A, blockRow, m);
    // Get sub-matrix Bsub of B
    Matrix Bsub = GetSubMatrix(B, m, blockCol);

    // Shared memory used to store Asub and Bsub respectively
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    // Load Asub and Bsub from global memory into shared memory
    // Each thread loads one element of each sub-matrix
    As[row][col] = GetElement(Asub, row, col);
    Bs[row][col] = GetElement(Bsub, row, col);
    // Continued on next page (you don't need to copy in this line ☺
    // Synchronise to make sure the sub-matrices are completely loaded
    // before starting the computation for each phase
    __syncthreads();

    // Now multiply Asub and Bsub together to complete phase m of the
    // calculation of this threads element of Csub
    for (int e = 0; e < BLOCK_SIZE; ++e)
     Cvalue += As[row][e] * Bs[e][col];

    // Synchronise again to make sure that the preceding calculation
    // has been completed by all threads in the block before loading two new
    // sub-matrices of A and B in the next iteration
    __syncthreads();
  }

  // Once all the phases are complete we can write Csub to device (global) memory
  // Each thread writes one element
  SetElement(Csub, row, col, Cvalue);
}
```

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

Answer: As before, although we have broken up the calculation of the inner product into partial results for a series of tiles of BLOCK-SIZE, we still have a total of A.width multiplications and A.width additions. So, the total number of compute operations stays the same.

However, for each element of matrix A that a thread reads in, it will have access to (BLOCK_SIZE-1) *additional* elements from shared memory.

If we let $m = \frac{A.width}{BLOCK\_SIZE}$ then each thread requires a total of:

2m reads from Global Memory

2m(*BLOCK_SIZE-1*) reads from Shared Memory

(Remember that the factor of 2 is because we need to read in from both A and B).

To a first order, we could take the read time from Shared Memory as negligible compared with the read time from Global Memory. So we have reduced the memory accesses by a factor of BLOCK_SIZE, thus increasing the Arithmetic Intensity by the same factor.

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

6 THE HOST CODE

Now we can write the host code for matrix multiplication. As with the first part, 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.

In contrast to the way I structured the code last week, this time we can actually paste the host code early in the file, which I think makes more logical sense (as it is the host that is calling the device).

Question: What has changed so that I can do this now, whereas last week this would have generated an error?

Answer: Refer back to section 2, and you will see that I included the declaration:

```
__global__ void MultKernShared(const Matrix A, const Matrix B, Matrix C);
```

This allows me to reference the kernel in the Host code before I have actually defined it.

There should be no surprises in the following code but there are some differences to what the code from last week. Copy in the code and I will then explain the error handling:

```
// 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 to device memory
  Matrix d_A;
  d_A.width = d_A.stride = h_A.width;
  d_A.height = h_A.height;
  size_t size = h_A.width * h_A.height * sizeof(float);
  cudaError_t err = cudaMalloc(&d_A.elements, size);
  printf("CUDA malloc h_A: %s\n",cudaGetErrorString(err));
  cudaMemcpy(d_A.elements, h_A.elements, size, cudaMemcpyHostToDevice);

  Matrix d_B;
  d_B.width = d_B.stride = h_B.width;
  d_B.height = h_B.height;
  size = h_B.width * h_B.height * sizeof(float);
  err = cudaMalloc(&d_B.elements, size);
  printf("CUDA malloc h_B: %s\n",cudaGetErrorString(err));
  cudaMemcpy(d_B.elements, h_B.elements, size, cudaMemcpyHostToDevice);

  // Allocate C in device memory
  Matrix d_C;
  d_C.width = d_C.stride = h_C.width;
```

```
 d_C.height = h_C.height;
 size = h_C.width * h_C.height * sizeof(float);
 err = cudaMalloc(&d_C.elements, size);
 printf("CUDA malloc h_C: %s\n",cudaGetErrorString(err));

 // Invoke kernel
 dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
 dim3 dimGrid(h_B.width / dimBlock.x, h_A.height / dimBlock.y);
 MultSharedKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
 err = cudaThreadSynchronize();
 printf("Run kernel: %s\n", cudaGetErrorString(err));

 // Read C from device memory
 err = cudaMemcpy(h_C.elements, d_C.elements, size, cudaMemcpyDeviceToHost);
 printf("Copy h_C off device: %s\n",cudaGetErrorString(err));
 // Free device memory
 cudaFree(d_A.elements);
 cudaFree(d_B.elements);
 cudaFree(d_C.elements);
}
```

You will see that on three occasions I have performed the memory allocations as a side effect on an assignment to a variable err. We are going to experiment with some large matrices and we need to check to make sure that the memory bounds of the GPU are not exceeded. Unless we do some checking, multiplication of matrices that are too large for the available memory will run to completion but return the wrong results. Cuda will, however, generate an internal error. So, to warn us of this we catch and print out any errors.
After calling the Kernel function, we have called cudaThreadSynchronise( ). This will return an error if any of the preceding tasks has failed and so again gives us a test to make sure we can trust the result.

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. I have not included any code for this but I assure you that you can define it yourself if you completed last week's lab class!

Your task: Once you have completed the code, run the project with two 16 x 16 matrices to check it is executing correctly.

7.1 Solution:

This is essentially the same code as for last week, but we need to add in assignments for the stride property. The additions are in bold.

```
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]);
  A.stride = A.width;
  B.height = A.width;
  B.width = atoi(argv[3]);
  B.stride = B.width;

  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.stride = C.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;
}
```

If you look in detail at the Host Code in Section 6, you will also see that the stride is instantiated in the copy of each matrix that is built in global memory on the Device. So, we don't actually use the values of stride in the copies of the matrices that are stored on the host. Which means that if you simply copied the code unchanged from last week, it would still work. However, that would not be good practice as if you later made an addition to the host code that referenced the stride in each matrix for some reason, you would throw an error. So, if you don't add in the two lines in bold above you would be injecting a "latent fault" – something that could turn into the root cause of an error at a later date.

8 TIMING THE EXECUTION OF THE KERNEL

We want to compare the execution speeds of the two versions. We could use the profiler in Nsight, but another reasonably straightforward way to do this is to use CUDA events. We need to make the following modifications to MatrixMult() (the function that executes on the host, and calls the Kernel).

First of all, we need to initialise start and stop events. This is done by adding three lines at the top of the function definition as follows:

```
void MatrixMult(const Matrix h_A, const Matrix h_B, Matrix h_C) {
  cudaEvent_t start, stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
```

These CUDA events are going to be processed on the device. The function cudaEventStream() is used to place these events in an event stream on the device. When the device reaches an event in an event stream, it will record a time stamp for the event. So, we use this to place time stamps on start and stop, respectively before and after execution of the kernel:

```
  cudaEventRecord(start);
  MultSharedKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
  err = cudaThreadSynchronize();
  cudaEventRecord(stop);
```

Now we can use cudaEventElapsedTime() to calculate the difference in times (accurate to about half a microsecond) of the two events. But there is a little finesse required. We need to be sure that the host has recorded the stop event before trying to calculate the time so a synchronisation is required. This will halt execution of the CPU, if required, until the stop event has been recorded. Place the following code after the cudaMemcpy() as here:

```
  err = cudaMemcpy(h_C.elements, d_C.elements, size, cudaMemcpyDeviceToHost);
  printf("Copy h_C off device: %s\n",cudaGetErrorString(err));

  cudaEventSynchronize(stop);
  float milliseconds = 0;
  cudaEventElapsedTime(&milliseconds, start, stop);
  printf("Elapsed time was: %f\n milliseconds", milliseconds);
```

9 YOUR TASKS

You can now check that the timer is working.

1. Re-run the project to check the timing information is being correctly printed out to the console. Make a note of the execution time (you will need to scroll the console back up above the print out of the three matrices).
2. Switch to the project from last week's lab class and add the timer events into the `MatrixMult()` function of that project.
3. Use Run/Configuration to change to last week's project and run that on two 16x16 matrices. Note the timing.
4. You will see that for small matrix sizes the two implementations produce similar timings, perhaps with the shared memory version running a bit slower.
5. Now, we want to scale up the size of the matrices. However, we don't want massive print outs at the console so I suggest you first change the print statements so that only the first ten rows and columns of each matrix is printed out. You should know how to do this, but ask one of us if you are unsure.
6. What you are going to do now is to progressively scale up the matrix size for each of your two implementations and record the execution times for each implementation. A simple way to record this is to open Excel and then create a worksheet capturing the respective times as matrix size is increased. Don't forget to keep the row sizes to be multiples of 16 (and use square matrices). It should give you enough to analyse if you follow the sequence of {16, 32, 64, 128, 256, …, 8192}. You can use the charting feature of Excel to plot execution time vs matrix size.
7. One final thing. Once you have completed the profiling you should continue to increase the matrix size to see how large a matrix is needed before a memory error is recorded.

10 CONCLUSION

I have trawled back over some earlier benchmarks and there is a clear trend. If you look into the Nvidia documentation you will see that different GPU architectures have different "compute capabilities", with .n increments between new architecture types. The Tesla architecture has compute capability 1; the Fermi architecture has compute capability 2; and the Kepler architecture has compute capability 3.

From a quick search around, I have found these are typical benchmarks:

Compute Capability	**Speed up through use of shared memory**
1.1	Speed up of 7x with a matrix size of 1000x1000 and continuing to increase significantly with increasing matrix size.
1.2	Speed up of 2x for a matrix size of 1000x1000
1.3	Only a 10% difference in running time for 1000x1000 matrices but increasing to a speed up of around 6.5 times for matrices of size 8000x8000.
2.0	Little significant difference in execution times for matrix sizes up to 1000x1000. A speed up of around 2.5 times for matrices of size 8000x8000.

So, let's see what you find! DO make a note of the compute capability of the machine you are using.