Shazib Hussain, Cranfield University

GPGPU Assignment

Course: Software Engineering For Technical Computing

Shazib Hussain
4-1-2015

Contents

1.0 Introduction

Utilising graphics processors for general purpose computing (GPGPU) has become extremely popular in recent years due to the advancement of graphics hardware. The highly parallelisable GPUs available today offer a promising solution to computationally expensive problems. A GPU is composed [in essence] of a collection of multiprocessors, and are designed from the ground up to be calculation workhorses for graphics applications. This architecture can be leveraged for performing other computational work.

Despite the underlying promise of high performance, the complicated underlying hardware architecture of GPUs often make it difficult to take full advantage of the performance without advanced knowledge in the field. One approach to GPGPU programming is CUDA. CUDA is a parallel programming platform and model developed by NVIDIA which gives direct access to the instruction set and memory of NVIDIA GPUs. It is made available via libraries to software developers.

To take advantage of CUDA/GPGPU, it is important to understand the paradigms of designing programs for parallel execution, this report takes a practical approach to discussing how a parallel CUDA program is designed by presenting and discussing a parallel iterative solver to the Laplace equation. The implementation is tailored to parallel computation using the CUDA system and are tested on a NVIDIA GTX 460. A serial and analytical solution are also presented for comparative purposes.

Note: The GTX 460 is a compute capability 2.1 device with 1024MB of global memory, 7 streaming multiprocessors, a warp size of 32, and 49152 bytes of shared memory per block.

2.0 Laplace's Equation

Laplace's equation is a second-order partial differential equation (1.1), which can be used to model a variety of physical steady-state distributions, such as temperatures, flows and potential distributions. Unaffected by time the equation depends only on the two spatial variables x and y.

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \tag{1.1}$$

To illustrate the iterative solution of Laplace's equation a distribution of temperature is considered in a two dimensional plane, where the temperature at the edges is defined by boundary conditions. The solution is performed on the unit square $[0, 1]$ x $[0, 1]$ on a uniform n x n grid. The boundary conditions are as follows:

1. The temperature on the edge x = 0 is: (1.2)

$$\sin(\pi y)^2 \tag{1.2}$$

2. The temperature on the other edges of the region is 0.

Figure 1 describes the setup of the plane on a 10 x 10 point grid

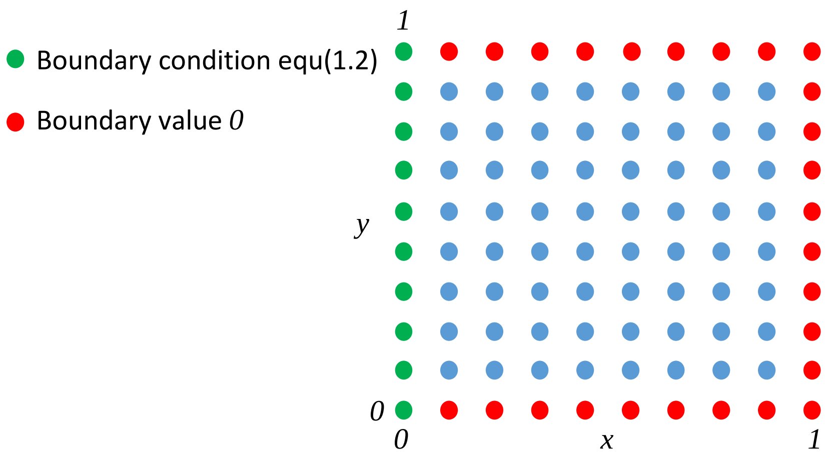

Figure 1 - Boundary Conditions on a 10 x 10 2D Plane

Whilst this is an arbitrary computation in terms of its application, the simplicity of the distribution allows us to focus on analysing the behaviour of parallel and serial architectures when performing the computations.

3.0 Analytical Solution

By implementing the separation of variables technique and, using the boundary conditions, we can begin to derive the analytical solution of Laplace's equation (3.1).

Finally, using the Fourier series:

$$\emptyset(x,y) = \frac{4}{\pi}\sum_{n=1}^{\infty}\frac{(\cos(\pi\, n) - 1\sinh\big(n\,\pi(x-1)\big)\sin(n\,\pi\,y)}{\sinh(-n\,\pi)(n^3 - 4n)} \tag{3.1}$$

The numerical solution of Laplace's equation generally results in a large sparse system and for large problems can become very computationally expensive (as evidenced in equation (3)). Here, iterative schemes can construct approximate solutions within a specific tolerance of accuracy where exact precision is not required.

For the purposes of analysing errors and comparing the results of our serial and parallel iterative solutions, the numerical solution is implemented on a *100* x *100* grid and plotted in Figure 2.

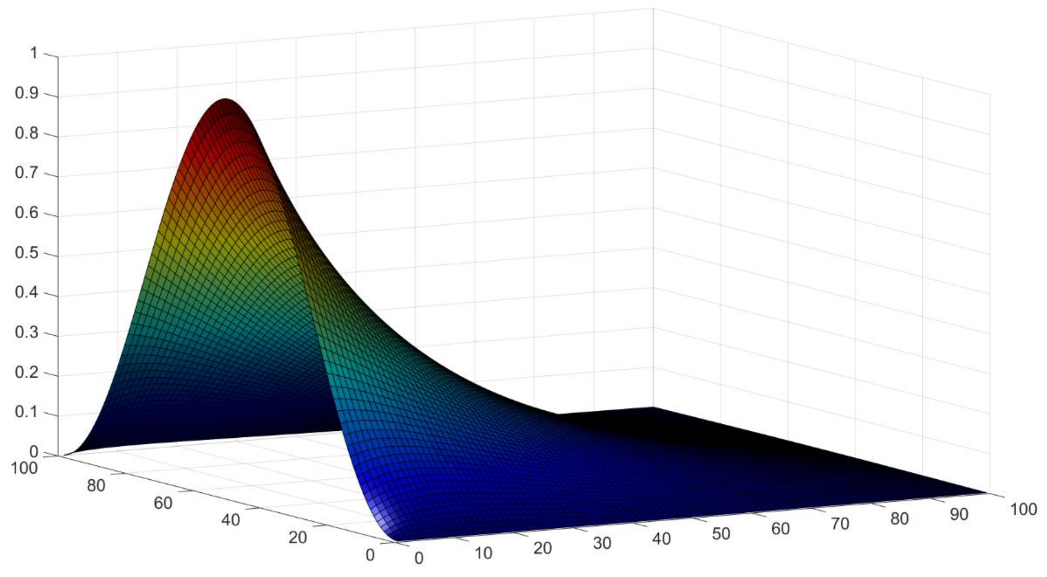

Figure 2 - Analytical Solution to Laplace with Boundary Conditions

The analytical solution was implemented in C++ and no timings were performed as they are not pertinent to the remainder of the report.

4.0 CUDA Implementation

CUDA extends the C programming language by introducing functions called *kernels* which when called are executed simultaneously in CUDA threads on the multiple processors available on the GPU. GPU's are composed of several types of memory, and processor blocks, called streaming multiprocessors (SM).

All the threads you want to run are grouped into *blocks.* Multiple blocks of threads will make up a *grid.* These blocks of threads will execute concurrently on a SM. Within the kernel, various variables are available to calculate the index [location] of the thread which is running. Kernels in different blocks are totally independent and cannot communicate with each other or exchange any type of data. Threads within a block can synchronise execution and share data via *shared memory*.

4.1 Kernel 1 – Global Memory

This is the most basic version of the kernel and uses global memory to compute a solution. The first task is to define the problem. A 1000 X 1000 matrix is allocated on host memory and initialised with the boundary conditions.

```
double *A = new double[ROWS * COLS];
```

Equal sized blocks of memory are allocated for two matrices on the GPU's global memory. Global memory is the slowest type of GPU memory but is read writable and accessible to all threads at all times.

```
checkCudaErrors(cudaMalloc((void **)&A_d, arrSize));
checkCudaErrors(cudaMalloc((void **)&B_d, arrSize));
```

A CUDA call is used to copy the initialised matrix to the GPU

```
checkCudaErrors(cudaMemcpy(A_d, A, arrSize, cudaMemcpyHostToDevice));
```

Next the kernel is called using a set of grid and block sizes. The maximum number of threads available to a block in compute capability 2.1 is 1024. As such we can safely set the block dimensions to (32,32) in for 1024 threads in 2D.

The grid sizes can be basically calculated with basic arithmetic utilising the total number of threads [1000 X 1000] and the number available per block [1024].

```
const dim3 BLOCK_DIM(32, 32); // 1024 threads
const dim3 GRID_DIM((COLS-1)/BLOCK_DIM.x + 1, (ROWS-1)/BLOCK_DIM.y + 1);
```

The kernel then runs once for each thread in each block, a block on an SM at a time.

The kernel uses the thread index and block index vectors to calculate its position on the matrix. This is then translated into the 1-dimensional value for the matrix in the GPU's global memory via pointer arithmetic.

If the thread is within the boundary values the calculation is performed.

```
__global__ void deviceJacobi(const double *M_src, double *M_dest, int rows,
int cols)
{
// Get the theoretical 2D row and col value from the block and grid
dimensions
int row = threadIdx.y + blockIdx.y * blockDim.y;
int col = threadIdx.x + blockIdx.x * blockDim.x;
// Translate this into the 1D index for the data in memory
int idx = ((row-1) * rows + col);

// Check if the value is within the boundaries
if (row < rows && col < cols && row > 1 && col > 1) {
      // Perform the jacobi iteration
      M_dest[idx] =  (0.25) * (M_src[idx+rows] + M_src[idx-rows] +
M_src[idx-1] + M_src[idx+1]);
}
}
```

The kernel is run for a fixed number of iterations, in this case 2000, alternatively writing to the two matrices on the GPUs memory. The time it takes for these 2000 iterations to perform is recorded using CUDA events.

A basic for loop runs this entire procedure 100 times and an average time is taken.

The time taken for this version of the kernel was **1576.65ms**

4.2 – Kernel 2 – Block Sizes

The underlying mechanics of threads and blocks are extremely important in efficient parallelisation. Threads inside a block, which as mentioned can exist up to 1024, are executed in consecutive *warps*. The SM can only execute a warp at a time. The threads within a warp all execute the same operations concurrently. Warps consist of 32 threads, hence blocks should be made up of a multiple of 32 otherwise a SM could potentially be running only a single thread as it executes the last warp of a block.

The second key consideration relates to the bottlenecks in the GPU architecture. SMs regularly page out warps when latency issues occur. For example, if a warp requests some global data, which will take many cycles to retrieve, this warp can be paged out and another paged in to execute its instructions until the data becomes available. The order of execution of warps cannot be guaranteed. To overcome this, each streaming processor must have sufficient warps available in a block to achieve maximum *occupancy*. Occupancy does not always guarantee performance, but gives an indication of the kernels ability to mask latencies.

Whilst the occupancy calculations were traditionally performed in a spreadsheet, in CUDA 6.5 NVIDIA introduced an occupancy API to make calculations to determine optimal block and grid sizes based on the kernel and size of the problem. This API has been used in the kernel to improve the performance of the code.

A call is made to the API to get the suggested grid and block sizes

```
int blockSize, minGridSize, gridSize;
cudaOccupancyMaxPotentialBlockSize(&minGridSize, &blockSize, deviceJacobi
        0, (ROWS * COLS));
```

These suggested values are then rounded and translated to 3D dim3 values for use in the kernel

```
// Round up according to array
gridSize = ((ROWS*COLS) + blockSize - 1) / blockSize;

// Turn these values into dim3 values
double gridVal = sqrt(gridSize);
double blockVal = sqrt(blockSize);
gridVal = ceil(gridVal);
blockVal = ceil(blockVal);
```

```
int grid = (int)gridVal;
int block = (int)blockVal;

const dim3 BLOCK(block, block);
const dim3 GRID(grid, grid);
```

Performing the computation again, results in an average computation time of **772.596ms.** The occupancy of the device has successfully been optimised and the grid size is a more accurate value ensuring that extra empty threads are minimised. The values suggested by the occupancy API could be further investigated and replaced with potentially, more accurate values, to increase performance further.

Another metric taken at this time is the time taken to copy the host data to the device global memory. The average taken was **2.10249ms.**

4.3 – Kernel 3 – Data Types

The third iteration of the kernel focuses of the arithmetic precision of the kernel by trading out *double* values for *float* values. Although doubles allow for higher precision, in certain calculations they will also produce larger errors. Doubles have a larger computational expense so we expect to see a faster execution time. A double is typically 8 bytes where a float is 4, so there should be some memory transfer gains also. Although the differences should not be too significant as compliers tend to perform significant amounts of optimisation.

The kernel now performs at an average of **761.751ms**, and the data copy time was recorded at **2.08508ms**. As expected small increases in both times.

4.4 – Kernel 4 – Pinned Memory

Host memory allocations made using the new keyword or using malloc are by default, pageable. Therefore, when a cudaMemcpy call initiates a data transfer from pageable host memory to device memory, the CUDA driver must create a non-paged (pinned/locked) array, copy the data to this array, and then copy the data to the device. CUDA implements a function call to implement pinned memory on the host directly. This memory is guaranteed to exist in system RAM and not be transferred to virtual memory. Because of this, the CUDA driver can request transfers without the involvement of the CPU. This results in faster Host to Device and Device to Host transfers.

The host array in now created with a CUDA API call:

```
// Allocate the matrix on the host
float *A;
checkCudaErrors(cudaMallocHost((void**)&A, arrSize));
```

This change should have no impact on the performance of the laplace computation.

As expected there is a noticeable increase in the speed of the host to device transfer, which averages **1.318ms**. Also as expected is the time of the computation, steady at **761.566ms**

4.5 – Kernel 5 – Memory Coalescing and 2D Arrays

Arrays allocated in global memory on the device are aligned to 256-byte segments by the CUDA driver. When warps access this memory, the device *coalesces* memory loads/stores to minimise the number of transactions which are taken, to reduce the bandwidth. The device will access memory in 32, 64 or 128 byte transactions. When data is requested, the device will fetch data into the L1 cache in *lines* of these fixed sizes and try to keep the transactions to a minimum. To satisfy the requirements of coalescing the data should exist in segments equal to these byte segments and the threads should access the data in sequence.

When allocating arrays linearly, e.g. via malloc/cudaMalloc, the data is instantiated as a single continuous chunk. In the case of a 1D array, a vector, data can be accessed continuously from ‘left to right’. Accessing the element directly adjacent to the prior one is computationally cheap, and the CUDA driver will fetch these surrounding values as it coalesces transactions.

When accessing a 2D array, which is represented as a 1D array, it is necessary to jump between values, and the memory access operation is dependent on the number of columns and rows in the array. A single row of the array could exist across two aligned segments. To minimise the number of memory accesses when reading any single row; to ensure that each row starts on a new segment, it is required to ‘pad’ the array rows with unused memory until the start of the next row.

This technique also mitigates bank conflicts which can occur in shared memory. Shared memory is memory which is ‘on-chip’ and much faster to access than global memory which is contained on a separate chip on the GPU. When stored in shared memory, the array will be split into several banks on the device. Each row can be padded to the beginning of a new bank to ensure problem free simultaneous access to the threads. (Multiple threads can access the shared memory).

CUDA provides a way of allocating memory to automatically pad an array from optimal coalescing with *cudaMallocPitch*:

```
checkCudaErrors(cudaMallocPitch(&A_d, &devPitchB, COLS*sizeof(float),
ROWS));
```

This function allocates the required memory on the device for the required number of rows. The width of each row is passed, in bytes, and the function determines the best value for padding, which it returns as the pitch value. The pitch value gives the new width of the rows in bytes.

Whilst this does result in some ‘wasted’ memory which won’t be used, the performance gains outweigh the memory losses.

```
checkCudaErrors(cudaMemcpy2D(A_d, devPitchA, A, hostPitchA,
      sizeof(float)*COLS, ROWS, cudaMemcpyHostToDevice));
```

The cudaMemcpy2D function uses the pitch values to copy the normal host array to the device as a 2D array with the corrct padding. Ordinary cudaMemcpy will no longer work as the host array is a different size. The 2D storage will support correct indexing of the elements in the memory allocated by cudaMallocPitch

It is possible to have a host array which is already a suitable size, in which case pitch would return the same value as the width of the array.

The new array in device global memory also requires the kernel function to be changed as the array elements can no longer be accessed traditionally.

```
// GPU Implementation of Jacobi using one cuda thread per element in the
matrix
__global__ void deviceJacobi(const float *M_src, float *M_dest, int rows,
int cols, size_t pitch)
{

// Get the theoretical 2D row and col value from the block and grid
dimensions
int row = threadIdx.y + blockIdx.y * blockDim.y;
int col = threadIdx.x + blockIdx.x * blockDim.x;


if (row < 1000 && row > 0 && col < 1000 && col > 0) {

      float* rowDest = (float*)(((char*)M_dest) + ((row)* pitch));
      float* rowUp = (float*)(((char*)M_src) + ((row-1)* pitch));
      float* rowDown = (float*)(((char*)M_src) + ((row+1)* pitch));
      float* rowSrc = (float*)(((char*)M_src) + ((row)* pitch));

      __syncthreads();

rowDest[col] = (0.25) *  (rowSrc[col - 1] + rowSrc[col + 1] + rowDown[col]
               + rowUp[col]);

      }
}
```

It was expected that this kernel would not offer significant speedup from the previous, but the results showed that the kernel performed significantly slower, at **991.968ms**. In this kernel, pointers must be created to the individual rows of the array, which are then indexed individually. Analysing the code, it does appear that there is significantly more pointer arithmetic occurring and accessing the elements has become more complicated. The theory is that this additional complexity has outweighed the current advantages of the coalesced global memory.

4.6 – Kernel 6 – Shared Memory

Shared memory is significantly faster than global memory, and the previous kernel has arranged the global memory in a way which is optimal for transfer to global memory. Shared memory is shared between all the threads within a block so all threads but the first and last should have fluid access to the memory.

The principle is fairly simple. A block will load three rows of the matrix into the shared memory, (or half of three rows, etc), and use the faster shared memory to perform the calculations before assigning the values back to global memory. In the case of the current kernel, rows 1-3 would be loaded into shared memory, then the first block would calculate the new values for row 2, (1 is a boundary). The block would have extremely fast and guaranteed coalesced reading for all the threads in the block. The results would be stored into a shared memory back and then row 2 would be updated in the result matrix. Then the next block would replace row 1, with row 4, and calculate the values for row 3. The changes implemented in Kernel 5 make using the shared memory ideal.

NOTE: This kernel was not successfully implemented and did not execute as intended. No results were generated to verify the shared memory speed increase.

5.0 Results

MATLAB was used to perform a matrix subtraction of the Kernel 5 code against the analytical solution. The errors were plotted and the serial solution is provided for reference.

Figure 3 shows a plot for the CUDA Solution

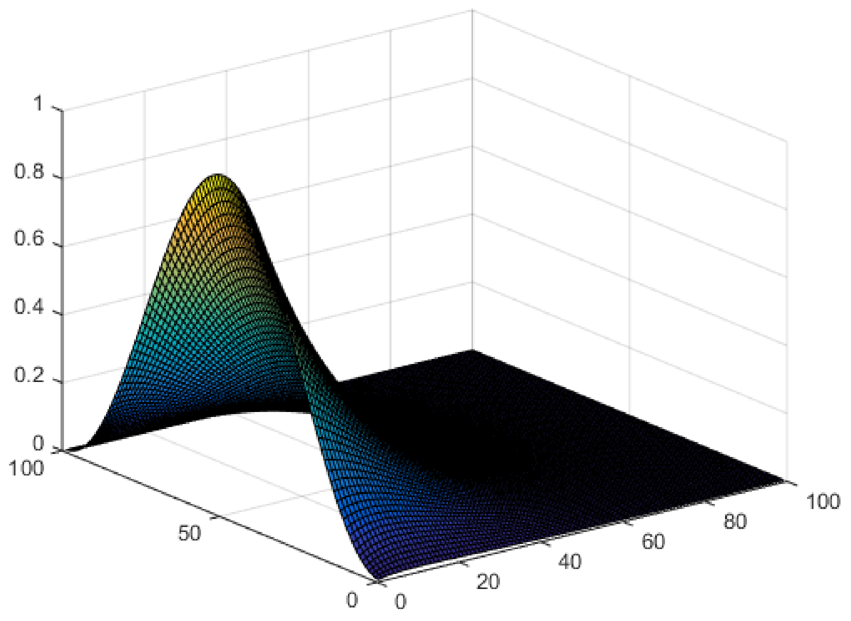

Figure 3 – CUDA Solution to Laplace with Jacobi

Figure 4 shows the errors from the CUDA solution. These results are as we would expect with the errors propagating into the centre from the edges where accurate boundary values are. The errors echo that of the serial solution.

The CUDA implementations developed have managed to achieve a best case speed of 761ms. This report echoes the common thoughts that *good* parallelised code is very difficult to write, with the kernel iterations at least halving the time taken to perform the iteration. The report has also considered the wider architecture concerning GPGPU programming, notably host data transfer bottlenecks.

The NVIDIA GTX 460 GPU used could perform much faster than the maximum that was achieved, through the use of shared memory, and further refinement on the grid and block sizes. An alternative strategy would involve using 1D blocks on 1D data, rather than the 2D approach taken in kernel 5. The texture memory could also be used to store a read-only version of the matrix. Texture memory is highly suited to data with 2D spatiality.

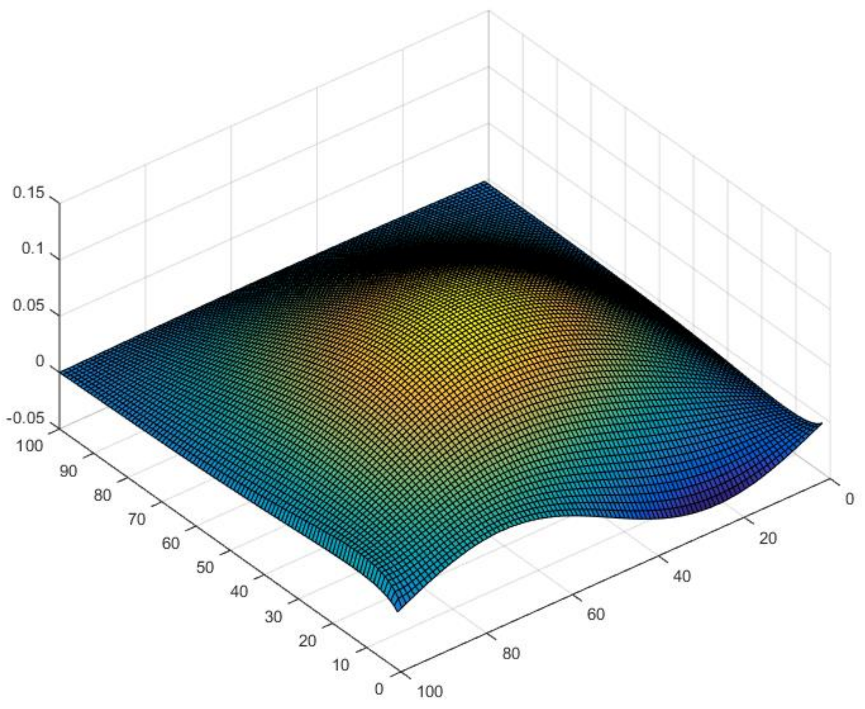

Figure 4 – Errors of CUDA Solution to Laplace with Jacobi on 100X100 grid with 2000 iterations