Parallel Programming Fall 2012

Assignment 5 CUDA Programming Assignment

B. Wilkinson and C Ferner, October 31, 2012

For this assignment, you will gain experience in writing and executing CUDA programs. For this assignment, **coit-grid07.uncc.edu** will be used, which has a single 448-core NVIDIA Tesla C2050 GPU installed.[1] If you have PC or laptop that has a NVIDIA GPU installed, you may wish to also try the programs on your PC/laptop. You will need to install the NVIDIA CUDA Toolkit. (Version 5.0 is the most recent.)

The assignment has two parts: In Part 1, you will practice compiling and executing simple CUDA programs - vector addition and matrix multiplication. The code is either given or is easy to write. In Part 2, you are asked to write a program to compute the static heat distribution in a room, which can the extended to other situations such as the heat distributed around a printed circuit board.

Preliminaries (2%)

Log onto **coit-grid07.uncc.edu**. Your username and password is the same as for the other coit-grid systems. You will be able to see your home directory on the cluster.

GPU limitations. Display the details of the GPU(s) installed by issuing the command **deviceQuery**. Keep the output as you may need it. In particular, note the maximum number of threads in a block and maximum sizes of blocks and grid. Also invoke the bandwidth test by issuing the command **bandwidthTest**. Note the maximum host to device, device to host, and device to device bandwidths.

Part 1 Compiling and executing CUDA program - Vector and matrix operations (38%)

Task 1 Compiling and executing vector addition CUDA program

In this task, you will compile and execute a CUDA program to perform vector addition. This program is given.

Create a directory called **VectorAdd** in your home directory and cd into it. Create a file called **VectorAdd.cu** containing the program:

```
#include <stdio.h>
#include <cuda.h>
#include <stdlib.h>
#define N 10      // size of array
#define T 10      // threads per block
#define B 1       // blocks per grid
```

[1] coit-grid07.uncc.edu can actually hold four C2050 GPUs. As a backup, we also have coit-grid06.uncc.edu server with the same GPU installed although that server can only be accessed off-campus through another server such as coit-grid01.uncc.edu.

```
__global__ void add(int *a,int *b, int *c) {

      int tid = blockIdx.x *  blockDim.x + threadIdx.x;
      if(tid < N) {
            c[tid] = a[tid]+b[tid];
      }
}

int main(void){
      int a[N],b[N],c[N];
      int *dev_a, *dev_b, *dev_c;

      cudaMalloc((void**)&dev_a,N * sizeof(int));
      cudaMalloc((void**)&dev_b,N * sizeof(int));
      cudaMalloc((void**)&dev_c,N * sizeof(int));

      for(int i=0;i<N;i++) {
            a[i] = i;
            b[i] = i*1;
      }

      cudaMemcpy(dev_a, a , N*sizeof(int),cudaMemcpyHostToDevice);
      cudaMemcpy(dev_b, b , N*sizeof(int),cudaMemcpyHostToDevice);
      cudaMemcpy(dev_c, c , N*sizeof(int),cudaMemcpyHostToDevice);

      add<<<B,T>>>(dev_a,dev_b,dev_c);

      cudaMemcpy(c,dev_c,N*sizeof(int),cudaMemcpyDeviceToHost);

      for(int i=0;i<N;i++) {
            printf("%d+%d=%d\n",a[i],b[i],c[i]);
      }

      cudaFree(dev_a);
      cudaFree(dev_b);
      cudaFree(dev_c);

      return 0;
}
```

Next, create a file called Makefile and copy the following into it:

```
NVCC = /usr/local/cuda/bin/nvcc
CUDAPATH = /usr/local/cuda

NVCCFLAGS = -I$(CUDAPATH)/include
LFLAGS = -L$(CUDAPATH)/lib64 -lcuda -lcudart -lm

VectorAdd:
   $(NVCC) $(NVCCFLAGS) $(LFLAGS) -o VectorAdd VectorAdd.cu
```

Be careful to have tabs where needed.

To compile the program, type **make VectorAdd** (or **make** as there is only one build command). Execute the program by typing the name of the executable (to include the current directory **./**), i.e. **./VectorAdd**. Confirm the results are correct.

Task 2 Experiments with Vector Addition code

Make the following changes to **VectorAdd.cu**, with good program structure. (Declare separate routines as appropriate.):

(a) Different sizes for the vectors – Replace the static declaration for a[N], b[N], and c[N] with dynamically allocated memory and add keyboard input statements to be to specify N.

(b) Add host code to compute the vector addition on the host only.

(c) Add code to verify that both CPU and GPU versions of vector addition produce the same correct results

(d) Different CUDA grid/block structures – Add keyboard statements to input different values for:

- Numbers of threads in a block (T)
- Number of blocks in a grid (B)

Include checks for invalid input. Ensure that GPUs limitations are met from the data given in deviceQuery (Preliminaries).

(e) Timing -- Add statements to time the execution of the code using CUDA events, both for the host-only (CPU) computation and with the device (GPU) computation, and display results. Compute and display the speed-up factor.

Arrange that the code returns to keyboard input after each computation with entered keyboard input rather than re-starting the code and having kernel code re-launch. Include print statements to show all input values.

During code development, it is recommended that the code is recompiled and tested after each of (a), (b), (c), (d) and (e).

Execute your code and experiment with different input values (at least eight different combinations of T, B, and N) and collect timing results including speed-up factor. What is the effect of the first kernel launch? Discuss results.

Task 3 Matrix multiplication

Modify the vector-addition code to perform matrix multiplication using two dimensional thread and block structures. Provide input keyboard input for different square 2-D grid/block structures. Modify the make file according to compile the code. Execute your code and experiment with different input values (at least eight different combinations of T, B, and N) and collect timing results including speed-up factor. Discuss results.

Part 2 Static Heat distribution problem

(Undergraduates: Tasks 1, 2, and 3 60%, Graduates: Tasks 1, 2, and 3 45%, Task 4 15%)

Preliminaries. we will write CUDA programs to determine the heat distribution in a space using synchronous iteration on a GPU. We will be solving Laplace's equation, which has wide application in science and engineering [1][2]. We will start with 2-dimensional space (square) and simple boundary conditions (walls at fixed temperatures). This program can then be modified to satisfy additional requirements.

Determining Heat Distribution by a Finite Difference Method. Consider an area that has known temperatures along each of its edges. The objective is to find the temperature distribution within. The temperature of the interior will depend upon the temperatures around it. We can find the temperature distribution by dividing the area into a fine mesh of points, $h_{i,j}$. The temperature at an inside point can be taken to be the average of the temperatures of the four neighboring points, as illustrated in Figure 1. For this calculation, it is convenient to describe the edges by points adjacent to the interior points. The interior points of $h_{i,j}$ are where $0 < i < n$, $0 < j < n$ [$(n - 1) \times (n - 1)$ interior points]. The edge points are when $i = 0$, $i = n$, $j = 0$, or $j = n$, and have fixed values corresponding to the fixed temperatures of the edges. Hence, the full range of $h_{i,j}$ is $0 \le i \le n$, $0 \le j \le n$, and there are $(n + 1) \times (n + 1)$ points. We can compute the temperature of each point by iterating the equation:

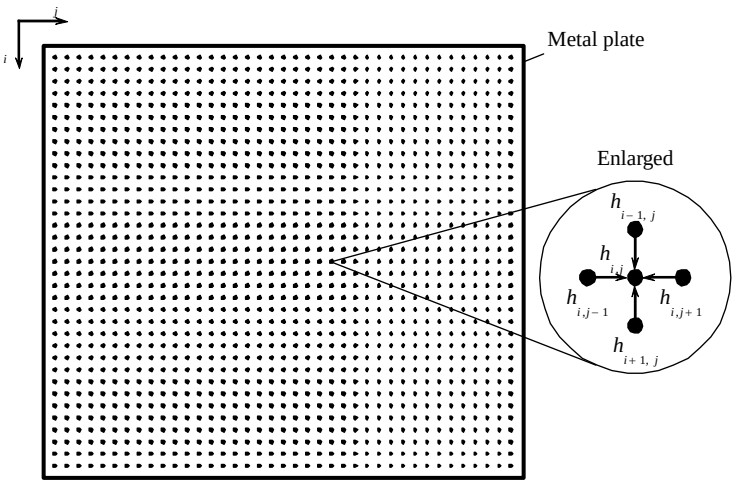

Figure 1 Heat distribution problem.

$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4}$$

$(0 < i < n, 0 < j < n)$ for a fixed number of iterations or until the difference between iterations of a point is less than some very small prescribed amount. This iteration equation occurs in several other similar problems; for example, with pressure and voltage. More complex versions appear for solving important problems in science and engineering. In fact, we are solving a system of linear equations. The method is known as the *finite difference* method. It can be extended into three dimensions by taking the average of six neighboring points, two in each dimension. We are also solving Laplace's equation.

Sequential Code. Suppose the temperature of each point is held in an array h[i][j] and the boundary points h[0][x], h[x][0], h[n][x], and h[x][n] ($0 \le x \le n$) have been initialized to the edge temperatures. The calculation as sequential code could be

```
for (iteration = 0; iteration < limit; iteration++) {

   for (i = 1; i < n; i++)
      for (j = 1; j < n; j++)
         g[i][j] = 0.25*(h[i-1][j]+h[i+1][j]+h[i][j-1]+h[i][j+1];

   for (i = 1; i < n; i++)// update points, Jacobi iteration
      for (j = 1; j < n; j++)
         h[i][j] = g[i][j];

}
```

using a fixed number of iterations. Notice that a second array g[][] is used to hold the newly computed values of the points from the old values. The array h[][] is updated with the new values held in g[][]. This is known as Jacobi iteration. Multiplying by 0.25 is done for computing the new value of the point rather than dividing by 4 because multiplication is usually more efficient than division. Normal methods to improve efficiency in sequential code carry over to GPU code and should be done where possible in all instances. (Of course, a good optimizing compiler would make such changes.)

Note: It is possible to use the same array for the updated points, thereby using some newly computed values for subsequent points (a Gauss-Seidel iteration) - this will converge significantly faster but may be difficult to implement on the GPU as it implies a sequential calculation. However a sequential version should really use Gauss-Seidel iteration for comparison purposes when computing speedup factors

Task 1 - Sequential Program

Write a C program to compute the temperature distribution inside the room shown in Figure 2 using Jacobi iteration. The room has four walls and a fireplace. The temperature of the wall is 20°C, and the temperature of the fireplace is 100°C. Divide the room into $N \times N$ points (including the boundaries), where N is input and can vary. The values of the points are stored in an array.

Make the program produce X11 graphics displaying the temperature contours at 10°C intervals in color. (Details of X11 code are given separately.). Instrument the code so that the elapsed time is displayed.

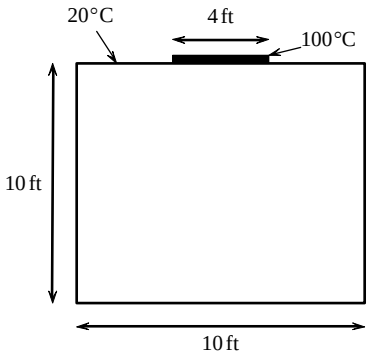

Figure 2 Room

Execute your program on coit-grid07.uncc.edu.

Task 2 - Basic CUDA Program

Modify the sequential program in Task 1 to be a CUDA program using host synchronization of threads. As with Part 1, incorporate:

(a) Use dynamically allocated memory for the data arrays (h[N][N], g[N][N]) and add keyboard input statements to be to specify N.

(b) Add host code to compute the heat distribution on the host only.

(c) Add code to ensure both CPU and GPU versions of heat distribution calculation produce the same correct results

(d) Different CUDA grid/block structures -- Add keyboard statements to input different values for the CUDA grid/block structure:

- Numbers of threads in a block (T)
- Number of blocks in a grid (B)

(2-D grid and 2-D blocks). Include checks for invalid input. Ensure that GPUs limitations are met from the data given in deviceQuery (Preliminaries).

(e) Timing -- Add statements to time the execution of the code using CUDA events, both for the host-only (CPU) computation and with the device (GPU) computation, and display results. Compute the speed-up factor and display.

Include print statements to show all input values. It is recommended that the code is recompiled and tested after each change. Experiment with different input values (at least eight different combinations of T, B, and N) and collect results on the Linux GPU server. Discuss.

Task 3 Warm body in room

Modify the CUDA code to compute the temperature distribution inside the room when there is a warm body in the room at a fixed temperature of 37°C (98.6°F) (any fixed location with a suitable size). Display results with one grid/block structure. It is suggested you get the sequential code working first with this modification.

Task 4 Termination Detection -- Graduate student only (30%), extra credit for undergraduate students

In the sample sequential code in Task 2, termination is set by a specific number of iterations. However the computed values may not have converged sufficiently towards the solution by that time. Re-write the CUDA code to termination the computation when all values computed in iteration t+1 differ by those in iteration t by less than a value that is input, say e. Repeat the study in Task 2 with this CUDA program and comment on the results. Use synchronization within blocks to terminate each block separately.

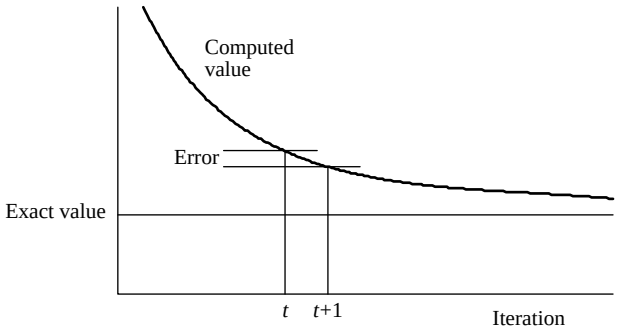

Figure 3 Convergence rate.

Note that the above does not guarantee the computed values are accurate to ±e, see Figure 3. A more complex termination calculation can be done, see [3] page 176.

Grading

Every task and subtask specified will be allocated a score so make sure you clearly identify each part you did. The computational efficiency and elegance of your solutions is will be a factor in grading.

Assignment Submission

Produce a document that show that you successfully followed the instructions and performs all tasks by taking screen shots and include these screen shots in the document. Give sufficient screen shots to demonstrate each task and sub-task has been fully completed. Provide insightful conclusions. Submit by the due date as described on the course home page. ***Include all code, not as screen shots but complete properly documented code listing. All work must be your own.***

Derivation of Jacobi Iteration Equation

(from [3] page 357)

The steady-state heat distribution is governed by Laplace's equation:

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0$$

(in two dimensions). The two-dimensional solution space is "discretized" into a large number of solution points, as shown in Figure 4.

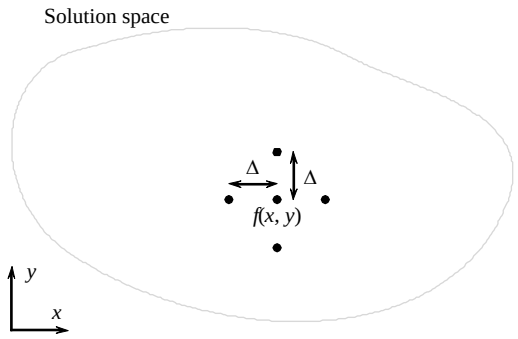

Figure 4 Finite difference method.

If the distance between the points in the x and y directions, Δ, is made small enough, the central difference approximation of the second derivative can be used:

$$\frac{\partial^2 f}{\partial x^2} \approx \frac{1}{\Delta^2}[f(x+\Delta, y) - 2f(x, y) + f(x-\Delta, y)]$$

$$\frac{\partial^2 f}{\partial y^2} \approx \frac{1}{\Delta^2}[f(x, y+\Delta) - 2f(x, y) + f(x, y-\Delta)]$$

[See Bertsekas and Tsitsiklis (1989) for proof.] Substituting into Laplace's equation, we get

$$\frac{1}{\Delta^2}[f(x+\Delta, y) + f(x-\Delta, y) + f(x, y+\Delta) + f(x, y-\Delta) - 4f(x, y)] = 0$$

Rearranging, we get

$$f(x, y) = \frac{[f(x-\Delta, y) + f(x, y-\Delta) + f(x+\Delta, y) + f(x, y+\Delta)]}{4}$$

The formula can be rewritten as an iterative formula:

$$f^k(x, y) = \frac{[f^{k-1}(x-\Delta, y) + f^{k-1}(x, y-\Delta) + f^{k-1}(x+\Delta, y) + f^{k-1}(x, y+\Delta)]}{4}$$

where $f^k(x, y)$ is the value obtained from kth iteration, and $f^{k-1}(x, y)$ is the value obtained from the $(k-1)$th iteration. By repeated application of the formula, we can converge on the solution.

Further Information

[1] Wikipedia “Laplace’s equation” http://en.wikipedia.org/wiki/Laplace%27s_equation
[2] Wikipedia “Heat equation” http://en.wikipedia.org/wiki/Heat_equation
[3] Barry Wilkinson and Michael Allen, *Parallel Programming: Techniques and Application Using Networked Workstations and Parallel Computers 2nd edition*, Prentice Hall Inc., 2005.