CMDA 3634 Spring 2016 Homework 05

T. Warburton

April 1, 2016

You must complete the following task by 5pm on Monday 04/11/16.

Each student must upload their homework to canvas.vt.edu. The write up should be presented in a \LaTeX formatted PDF document.

Upload checklist:

1. \texttt{firstnameLastnameHW05.tex} \LaTeX file.
2. Any figure files to be included by \texttt{firstnameLastnameHW05.tex} file.
3. \texttt{firstnameLastnameHW05.pdf} PDF file.
4. CUDA based circuit solver source code file named \texttt{cudaCircuitShell.cu}

This is an individual assignment. You must write all the code yourself. You may not copy code from another student. You may discuss the assignment with other students. You are encouraged to use textbooks and internet resources. You must cite all resources used via footnotes or a bibliography.

Notes:

- Do not use any third party code or software libraries besides CUDA for the implementation as it is not necessary.
- The CUDA performance studies should be performed on a single HokieSpeed compute node using an interactive job or via the queue system.
- For each individual coding task you must use the \texttt{lstinputlisting} macro from the \LaTeX listings package to include relevant source code to that question, i.e. do not just include a listing of the whole code.
- You must describe what you have done and observed for each question.

150 points will be awarded for successfully completing the assignment. Extra credit will be awarded as appropriate.
Q1 [150 points] Modify an existing serial resistor network solver using CUDA to run on an NVIDIA GPU

Problem description: Consider a direct current circuit composed of $N \times N$ indivisible closed loops as illustrated in the case of $N = 9$ in Figure 1. The black lines correspond to lossless wires. Assume all wires in the network include a resistor of 1 Ohm resistance. The circuit is powered by two 1 Volt DC battery represented by the blue rectangles.

Figure 1: Diagramatic representation of DC resistor network. Thin blue lines: lossless wires. Red rectangles: 1 Ohm resistors. Blue rectangle: 1 Volt DC battery. Open cells are the “ghost cells”.

In this question we yet again revert back to the serial circuit solver. You should edit the cudaCircuitShell.cu text source code file from canvas.vt.edu, or directly from here.

Q1a-f: (5 points for each task) In HW01 we used a doubly nested for loop to compute the sum of the squares of the change in circuit loop currents between iterations. In this assignment we will flatten the doubly nested for loop into a single for loops that is executed by massively parallel threading. We will use a one-dimensional reduction pattern as demoed in class. Specifically, each thread-block will use $T_{reduction}$ threads and each thread will compute the square of the change in circuit loop current for one cell. Then you will use a binary tree reduction pattern to sum these $T_{reduction}$ values up using shared memory. Each thread-block will write just one value to an intermediate DEVICE array.

You will of course need to specify that the kernel runs with enough thread-blocks such that the whole circuit of $(N+2)^2$ values is processed. Once the number of values is reduced by a factor of $T_{reduction}$ those values should be copied back to the HOST to be summed up to complete the calculation of epsilon.

The CUDA kernel `partialReductionKernel` DEVICE code that you need to complete is:

```c
void partialReductionKernel(int M, double *c_old, double *c_new, double *c_partEpsilon){
// Q1a: declare shared memory array for thread-block tree reduction */
// Q1b: use thread index and block index to identify array index */
// Q1c: each thread loads a value into shared memory array */
// Q1d: synchronize all threads in thread block */
// Q1e: tree based reduction of shared memory array to one entry per thread-block */
// Q1f: one thread from thread-block writes out reduced single entry to partEpsilon */
}
```

Q1g: (10 points) Complete the `cudaCalculateEpsilon` HOST code by invoking the `partialReductionKernel`.

2
I have provided grid dimensions (\texttt{gDim} for the number of thread-blocks required) and the thread-block dimension (\texttt{bDim} for the number of threads per thread-block).

### Q1: compute \(\varepsilon\) with CUDA

```c
double cudaCalculateEpsilon(int N, double *c_old, double *c_new, double *c_partEpsilon, double *h_partEpsilon) {
    // Cells in circuit
    int M = (N+2)*(N+2);

    // Assume a one-dimensional thread array
    dim3 gDim((M+T_reduction-1)/T_reduction); // number of blocks of size T_reduction
to cover M
dim3 bDim(T_reduction); // number of threads per block

    // Q1g: call kernel to partialReductionKernel
    // Q1g: end
}
```

### Q1h: (10 points) Add CUDA DEVICE kernel code to \texttt{cudaIterateKernel} so that each thread uses its two-dimensional thread indices and thread-block indices to associate itself uniquely with one \((i,j)\) cell in the circuit. [ assume that the kernel is launched with thread-blocks of size \((T_reduction \times T_reduction)\)].

```c
// implement CUDA iterate kernel using a two dimensional array of threads */
.global. void cudaIterateKernel(int N, double *c_old, double *c_new) {
    // Q1h: compute i and j using CUDA thread indices, block indices, and block dimensions */
    // Remember: use 1-indexing to match the above for loop in the original iterate function */
    int i, j;

    // Q1h ends here */
}
```

### Q1i: (10 points) Add CUDA DEVICE kernel code to \texttt{cudaIterateKernel} so that each thread computes the new loop current at cell \((i,j)\). Remember to check that the \(i\) and \(j\) indices are valid (i.e. check that \(1 \leq i, j \leq N\) before writing the result out).

```c
// Q1i: each thread updates one single entry of Inew using update formula */
// Q1i ends here */
}
```

### Q1j: (10 points) Write CUDA HOST code that launches the \texttt{cudaIterateKernel} with the specified thread grid dimension \texttt{gDim} and thread-block dimension \texttt{bDim}.

```c
void cudaIterate(int N, double *c_old, double *c_new) {

    // CUDA thread dimensions for two-dimensional array of thread-blocks */
dim3 gDim((N+Titerate-1)/Titerate, (N+Titerate-1)/Titerate);
dim3 bDim(Titerate, Titerate);

    // Q1j: invoke CUDA cudaIterateKernel */
}
```

### Q1k: (10 points) Write CUDA HOST code that uses \texttt{cudaMalloc} to allocate DEVICE storage for the new and old loop currents and the intermediate partially reduced \(\varepsilon\) values.

```c
void cudaSolve(int N, double tol) {
    /* use for computed \(\varepsilon\) */
}
```
\begin{verbatim}
int N2 = (N+2)*(N+2);  // number of cells
double epsilon;

int NpartEpsilon = (N2+Treduction−1)/Treduction;

double *h_Inew = (double*)malloc(N2, sizeof(double));
double *h_Iold = (double*)malloc(N2, sizeof(double));
double *h_partEpsilon = (double*)malloc(NpartEpsilon, sizeof(double));

double *c_Inew, *c_Iold, *c_partEpsilon;

/* your Q1k code to build and zero out DEVICE vectors for c_Inew, c_Iold, c_partEpsilon starts here */

/* your Q1k code ends here */
\end{verbatim}

Q1: (20 points) Check that your CUDA code reproduces the results of the original serial code from HW01 to a reasonable tolerance. Use a convergence tolerance of $10^{-9}$ to guarantee that the circuit currents have adequately converged for this test. Use the \texttt{verbatim} environment to report the loop currents output by your code.

Q1m: (50 points) In this question you are going to compare the performance of your HW04 OpenMP circuit code and your new HW05 CUDA code. Both OpenMP and CUDA involve parallel computations that are subject to Amdahl’s law, albeit in slightly different ways. To determine the impact of problem size on efficiency you will time how long it takes your CPU (using OpenMP) and GPU (using CUDA) codes to solve progressively larger circuits.

Use your HW04 OpenMP circuit code to time how long it takes to solve the circuit problem for $N = 10, 100, 1000, 2000, 3000,$ and $4000$ to a tolerance of $10^{-5}$ using 12 threads on a HokieSpeed compute node. Repeat this experiment using your new CUDA code on a single GPU on a HokieSpeed compute node. Plot the compute times for both OpenMP and CUDA as a function of $N$ on the same chart. Comment on the relative performance of the CPU using OpenMP and GPU using CUDA.

Extra Credit: (10 points) explain what happens when you try to solve a circuit of size $N = 4096$ or larger using your CUDA code.

Extra Credit: (25 points) include a selfie of you attending the ARC advanced CUDA class in your report.