# CMDA 3634: HW6

May 03, 2016

## 1 Introduction

This document outlines the answers to the questions with written answers in homework 6 for CMDA 3634. Code for the homework is listed with comments for each question.

## 2 Q2a-e

Here in question 2, we will be modifying the given OpenMP circuit solver to use OpenACC directives. Then we will compare the performance of newly created OpenMP code with the CUDA circuit solver. Note: when compiling OpenACC code on Hokiespeed

module purge module load pgi/15.7 pgcc -acc -ta=tesla -Minfo=accel -o accCircuit.c accCircuit.c -lm

## 3 Q2a-e

By using the -Minfo, I was able to see what is being done with the code I put in. Compiler can generate code eventhough from it's called from the kernel. This is the function that's going to be executed by a thread on the device.

```
#pragma acc routine seq
int idx(int N, int i, int j){

  return i + j*(N+2);

}
```

## 4 Q2a-e

Using Kernels, I was able literally see what's being done in the kernel. Information tells us what additional directive we need. Here, I had trouble with parser error and it helped me figure out the clues. data copy for both Ined and Iold indices are required because compiler can't automatically read the function.

```
double calculateEpsilon(int M,
                        int N,
                        double *Iold,
                        double *Inew){

  double epsilon = 0;

  int i,j;

  //#pragma acc kernels
#pragma acc data copy(Inew[0:(N+2)*(M+2)])
#pragma acc data copy(Iold[0:(N+2)*(M+2)])
#pragma acc parallel loop reduction(+:epsilon)
  //#pragma acc loop gang
  for(j=1;j<=M;++j){
    //#pragma acc loop vector(128)
#pragma acc loop
    for(i=1;i<=N;++i){
      epsilon += pow(Inew[idx(N,i,j)]-Iold[idx(N,i,j)],2);
    }
  }

  ...

}
```

## 5 Q2a-e

In OpenMP, private variable is used so that it won't overlap a data, but in OpenACC, any scalar accessed using parallel loop is set to private as default. Reduction is also a similar case, except now it's done at the end of the loop on all private copied variables.

```
void iterate(int M, int N, double *Iold, double *Inew){

  int i,j;

#pragma acc data copy(Inew[0:(N+2)*(M+2)])
#pragma acc data copy(Iold[0:(N+2)*(M+2)])
#pragma acc parallel loop
  for(j=1;j<=N;++j){
#pragma acc loop
    for(i=1;i<=N;++i){
      Inew[idx(N,i,j)] = ...



}
```

## 6 Q2d

Here are the loop currents tested for 100x100, 200x200, and 400x400 Compare to OpenMP code, there is significant difference in time. Obviously, time will increase as N increment. However in this case, OpenACC code shows noticeable lag even at low inputs.

**100**:

```
epsilon = 0.000576371
epsilon = 0.000554958
I_{11} = 0.302251
I_{10 10} = 0.000903038
elapsed time = 0.010000
```

**200**:

```
epsilon = 0.000576371
epsilon = 0.000554958
I_{11} = 0.302251
```

```
I_{10 10} = 0.000903038
elapsed time = 0.050000
```

**400**:

```
epsilon = 0.000576371
epsilon = 0.000554958
I_{11} = 0.302251
I_{10 10} = 0.000903038
elapsed time = 0.200000
```

## 7 Q2e

For this part, we will compare the time how long it takes for both to execute 40 iterations of the circuit problem. As you can see the performance of the time it took for both, OpenACC is significantly slower after size of 100. Huge data transfer bottlenecks due to large computations and data movements.

| N | OpenMP Timing | OpenACC Timing |
|---|---|---|
| 10 | 0.00541263s | 0.000000s |
| 100 | 0.00557934s | 0.010000s |
| 1000 | 0.312785s | 1.340000s |
| 2000 | 1.36741s | 5.440000s |
| 3000 | 3.4096s | 12.230000s |
| 4000 | 5.62738s | 21.650000s |

## 8 Q3a-g

Translate an existing CUDA circuit solver to OpenCL. Note: Compiling on HokieSpeed: module purge module load cuda/6.5.14 module load gcc/5.1.0

gcc -I/opt/apps/cuda/6.5.14/include/ -o oclCircuit oclCircuit.c -lOpenCL -lm

To run on HokieSpeed:

./oclCircuit.c 1000

## 9 Q3a-g cont.1

CUDA HOST code is based on reduce.c file given during class. Some of the basic platform and context are the same and modified for use interate and reduction kernel functions.

```
  ...

  // build kernel function
  const char *sourceFileName = "oclKernels.cl";
  const char *functionName = "oclIterateKernel";
  const char *functionName2 = "oclReductionKernel";


    ...

}
```

## 10 Q3a-g cont.2

Both kernels functions are first compiled to check for proper computation. If one fails, compiler will throw an error below.

```
    ...

    /* create runnable kernel */
  cl_kernel kernel = clCreateKernel(program,
                                    functionName,
                                    &err);
  cl_kernel kernel2 = clCreateKernel(program,
                                    functionName2, &err);
  if (! kernel || err != CL_SUCCESS){
    printf("Error: Failed to create compute
        oclIterateKernel kernel!\n");
    exit(-1);
  }
  else if (! kernel2 || err != CL_SUCCESS){
    printf("Error: Failed to create compute
        oclReductionKernel kernel2!\n");
    exit(-2);
  }


  ...
}
```

## 11 Q3a-g cont.3

Here is where all the deleted code from the CUDA DEVICE kernels are now crunched in this main function. First quarter of them are basically from CUDA circuit solver. `c_idx` function is unwrapped and inserted into appropriate indicies. "`c_Iold, c_Inew, c_partEpsilon`" create device buffer and copy from host buffer. Just like CUDA we will allocate storage on the DEVICE. Then set global thread array size and fix the local work group size. Reference to the CUDA code, we have iterate kernel queued twice swapping the `c_Inew` and `c_Iold` indices. Kernel2 is then queued to calculate for the epsilon values which will later copied back from device to host to finish reduction on HOST.

```
_/* read N from the command line arguments */
  int N = atoi(argv[1]);
  int M = N; // by default use a square circuit

  /* use for computed epsilon */
  int Ncells = (N+2)*(M+2); // number of cells

  /* create host array */
  size_t sz = Ncells*sizeof(double);
  size_t szEpsilon = ((Ncells+256-1)/256)*sizeof(double)

  double *h_Iold = (double*) malloc(sz);
  double *h_Inew = (double*) malloc(sz);
  double *h_partEpsilon = (double*) malloc(szEpsilon)

  // fill up host array.
  int n;
  for(n=0;n<Ncells;++n){
    h_Iold[n] = 1;
    h_Inew[n] = 0;
  }

  // set current
  // int c_idx(int N, int i, int j)
  // idx(i + j*(N+2))
  h_Iold[0 + 1*(N+2)] = 1;
  h_Inew[0 + 1*(N+2)] = 1;

  h_Iold[i + j*(N+3)] = 1;
  h_Inew[i + j*(N+3)] = 1;

  cl_mem c_Iold = clCreateBuffer(context,
                                 CL_MEM_READ_WRITE| CL_MEM_COPY_HOST_PTR,
                                 sz,
                                 h_Iold,
                                 &err);
  cl_mem c_Inew = clCreateBuffer(context,
                                  CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
                                 sz,
                                 h_Inew,
                                 &err);
// cl_mem c_partEpsilon = clCreateBuffer(context,
                                 CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
                                 szEpsilon,
                                 h_partEpsilon
                                 &err);

    int dim = 1;
    int Nt = 256;
    int Ng = Nt*((N+Nt-1)/Nt);
    size_t local[3] = {Nt,1,1};
    size_t global[3] = {Ng,1,1};


  /* iterate using the Jacobi method here */
  int it, Nit=40;
  for(it=0;it<Nit;++it){

    /* iterate from Iold to Inew */
    //  cudaIterate(M, N, c_Iold, c_Inew);
    /* now set kernel arguments */
    clSetKernelArg(kernel, 0, sizeof(int), &M);
    clSetKernelArg(kernel, 1, sizeof(int), &N);
    clSetKernelArg(kernel, 2, sizeof(cl_mem), &c_Iold);
    clSetKernelArg(kernel, 3, sizeof(cl_mem), &c_Inew);

    /* queue up kernel */
    clEnqueueNDRangeKernel(queue, kernel, dim, 0,
                           global, local, 0,
                           (cl_event*)NULL, NULL);


    /* iterate from Inew to Iold */
    //  cudaIterate(M, N, c_Inew, c_Iold);
    clSetKernelArg(kernel, 0, sizeof(int), &M);
    clSetKernelArg(kernel, 1, sizeof(int), &N);
    clSetKernelArg(kernel, 2, sizeof(cl_mem), &c_Inew);
    clSetKernelArg(kernel, 3, sizeof(cl_mem), &c_Iold);

    /* queue up kernel */
    clEnqueueNDRangeKernel(queue, kernel, dim, 0,
                           global, local, 0,
                           (cl_event*)NULL, NULL);

    /* Cells in circuit */
    int  L = (N+2)*(M+2);

    clSetKernelArg(kernel2, 0, sizeof(int), &L);
    clSetKernelArg(kernel2, 1, sizeof(int), &C_Iold);
    clSetKernelArg(kernel2, 2, sizeof(cl_mem), &c_Inew);
    clSetKernelArg(kernel2, 3, sizeof(cl_mem),
                                   &h_partEpsilon);

    /* Assume a one-dimensional thread array */
    int NgEpsilon = Nt*((L+Nt-2)/Nt);

    /* call kernel to partialReductionKernel */
    size_t global2[3] = {NgEpsilon,1,1};
    clEnqueueNDRangeKernel(queue, kernel2, dim, 0, global2,
                           local, 0, (cl_event*)NULL, NULL);

    /* Copy partially reduced array from DEVICE to HOST */
    cl_mem c_partEpsilon = clCreateBuffer(context,
                               CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
                               szEpsilon, h_partEpsilon &err);

    /* Finish reduction on HOST */
    double epsilon = 0;
    int n;
    for(n=0;n<NgEpsilon;++n){
      epsilon += h_partEpsilon[n];
    }

    epsilon = sqrt(epsilon);

    /* print current residual */
    printf("epsilon␣=␣%g\n", epsilon);

  }

  ...

}
```

## 12 Q3b-e

In device kernels, all the `___` functions are now renamed to `c__kernel` and saved them separately as oclKernels.cl(CUDA DEVICE). Rest of the functions are moved to oclCircuit.c(CUDA HOST) for a further modification. All the local and global indices are renamed to appropriate OpenCL indices. Thread array dimensions are renamed as well. Pragma OPENCL EXTENSION near the header is required when the double floating-point directive is declared in the kernel code. Jacobi kernel formula is unwrapped from $\mathtt{c}_i dx() function.$

```
/* here we define block-size for reduction */
```

```
#define Treduction 256
#define Titerate 16
#pragma OPENCL EXTENSION cl_khr_fp64: enable

/* Use a 1D thread array to compute a partially reduced epsilon */
__kernel void oclReductionKernel(int L,
                                 __global double *c_Iold,
                                 __global double *c_Inew,
                                 __global double *c_partEpsilon){

  __local double s_Idiff[Treduction];

  int t = get_local_id(0); //threadIdx.x;
  int b = get_group_id(0); //blockIdx.x;
  int id = t + Treduction*b;

  s_Idiff[t] = 0;
  if(id<L)
    s_Idiff[t] = pow(c_Inew[id]-c_Iold[id], 2);

  /* tree based reduction of shared memory array
  to one entry per thread-block */
  int alive = Treduction/2;
  while(t<alive && alive>=1){

    barrier(CLK_LOCAL_MEM_FENCE);

    if(t+alive<Treduction)
      s_Idiff[t] += s_Idiff[t+alive];
    alive /=2;
  }

  ...

}

/* implement CUDA iterate kernel using a two dimensional array of threads */
__kernel void oclIterateKernel(int M, int N, __global double *c_Iold, __global double *c_Inew){

  /* compute i and j using CUDA thread indices, block indices, and block dimensions */
  /* Remember: use 1-indexing to match the for loop in the original iterate function */
  int i,j;
  i = 1 + get_global_id(0);
  j = 1 + get_global_id(1);

  /* each thread only updates one single entry of Inew using update formula */
  if(i<=N && j<=M){

    //int c_idx(int N, int i, int j)
    //int id = i + j*(N+2);
    c_Inew[i + j*(N+2)]
      = 0.25*(c_Iold[(i+1) + j*(N+2)] +
              c_Iold[(i-1) + j*(N+2)] +
                     c_Iold[i + (j+1)*(N+2)] +
                            c_Iold[i + (j-1)*(N+2)]);
  }
}

}
```

## 13 EC

Explain what happens when you try to solve a circuit of size N = 4096 or larger using your CUDA code? Circuit size of 4096 x 4096(n=4096) gives no circuit current outputs due to the exceeding the number of memory size. There are total number of registers available per block for each individual GPU, but using 4096 as the size exceeds the maximum dimension of a grid size.

## 14 References

2016. URL:http://tex.stackexchange.com/questions/42144/how-to-define-macro-that-only-makes-argument-su

2016. http://on-demand.gputechconf.com/gtc/2013/presentations/S3076-Getting-Started-with-OpenACC.pd

2016. http://on-demand.gputechconf.com/gtc/2013/presentations/S3084-OpenACC-OpenMP-Directives-CCE.p

2016. https://www.olcf.ornl.gov/wp-content/uploads/2013/02/Intro$_{t}o_{O}penACC-JL.pdf$

2016. https://devblogs.nvidia.com/parallelforall/7-powerful-new-features-openacc-2-0/