Week 13 Discussion Questions

This discussion questions are should aid you in Homework 10 which is to implement the 2D SOR algorithm using CUDA on a GPU. Recall in 2D SOR that on each iteration we replace all interior values by the average of their four nearest neighbors (top, bottom, left, and right on the below diagram).

You can make a simplifying assumption that the array is square with nxn interior elements. However, your code should handle n not matching the dimensions of the grid of threads.

Answering the following questions should help your design.

a) Maximum threads per block is 1024 ($2^{10}$). If we want to make it 2-dimensional (and square), what would the dimensions of the thread block (DIM by DIM)?

b) If we want to "tile" blocks across the n x n interior of the array, what is the dimension of the grid of blocks?

```c
#define DIM __

dim3 dimBlock( _____________ , _____________ );
dim3 dimGrid( _____________ , _____________ );
```

![Diagram showing the grid and block structure](image)

Complete the assignment statements.
"global thread" x, y calculated by:

```
  x =
  y =
```

offset is the 1-d index from the beginning of the "2-d matrix":

```
  offset =
  left =
  right =
  top =
  bottom =
```
Week 13 Discussion Questions

**Hint on (c) and (d) below:** You might consider having the host performing the main 2D SOR loop and the kernel doing a single iteration of the calculation which gets invoked repeatedly by the host. (Include data copies between host and device global memory, synchronization of threads, etc.)

c) Design the host’s algorithm:

d) Design the device’s kernel:
Comp. Arch. Lab 13 Due: Saturday, Nov 10 at 11 PM

Learning Objectives:
- Write CUDA commands to allocate and copy between host memory and device memory
- Apply CUDA commands to launch a kernel with a specified grid of thread blocks
- Time CUDA events to evaluate performance.

To start the lab:
- watch the Lab 13 Video on the eLearning system
- download lab13.zip from the eLearning system to your computer
- upload lab13.zip to student.cs.uni.edu using a secure ftp client (winSCP, FileZilla, scp, etc.)
- log-on to Blue Waters log-in node using Putty/ssh
- use scp to copy lab13.zip from student.cs.uni.edu Blue Waters
- unzip lab13.zip and cd lab13

NOTE: To use CUDA on Blue Waters you must first load the CUDA module into your environment by the command:

```
module load cudatoolkit
```

To see a list of the installed modules use: `module list`
To see a list of the modules available use: `module avail`

**Part A:** Using an editor open the file `lab13/enum_gpu.cu` which contains a CUDA program that queries to see the number of CUDA GPU cards installed on the host computer and their characteristics. Compile it by:

```
nvcc -o enum_gpu enum_gpu.cu
```

Run it using qsub similar to MPI by:

```
qsub qsub_enum_gpu
```

Answer the following questions about the Tesla K20X GPU cards:

a) What is the *Compute capability* (p. 165)?

b) Is the *Device copy overlap* enabled?

c) How much *Total global memory* exists?

d) What is the *Multiprocessor count*?

e) How much *Shared mem per mp*?

f) How many *Threads per warp*?

g) What is the *Max. threads per block*?

h) What is the *Max. thread dimensions* (i.e., “block” dimensions)? _______ x _______ x _______

i) What are the *Max. grid dimensions*? ______________ x ______________ x ______________

**Part B:** Using an editor open the file `lab13/addVectors.cu` which contains a CUDA program that allows the user to enter an integer command-line argument in the `qsub addVectors aprun` line: the length of the vectors. It creates three 1-dimensional arrays of that length (a, b, and c), fills arrays a and b with random floating-point numbers, and uses the GPU to sum the arrays. The vector sum is also calculated sequentially for comparison. Compile the program using:

```
nvcc -o addVectors addVectors.cu
```
Comp. Arch.

and run the program with lengths of 100000, 1000000, and 10000000 (i.e., uncommenting the appropriate qsub.addVectors aprun line) by: qsub qsub.addVectors

Complete the timing table:

<table>
<thead>
<tr>
<th>Vector Length</th>
<th>Sequential Vector Addition (in ms)</th>
<th>CUDA Vector Addition (in ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,000,000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10,000,000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a) Why is sequential vector addition faster for a length of 100,000?

b) As the length gets bigger, why is the CUDA version faster?

Recall the general CUDA programming steps:

main: (running on host)
1) Reads data from file (or generate it randomly) into host memory
2) Allocate global memory on device via cudaMalloc for data
3) Copy data from host memory to global memory via cudaMemcpy
4) Kernel launches a grid of thread blocks where:
   a) Threads within a block cooperate via shared memory, e.g., "__shared__" qualifier
   b) Threads within a block can synchronize, e.g., "__syncthreads();" -- barrier
   c) Threads in different blocks cannot cooperate much, e.g., atomic functions "atomicAdd()", __threadfence()
5) Copy results from global memory to host memory via cudaMemcpy

c) In the partial main function of addVectors.cu identify the above steps:

```c
// Copy a and b to device memory
size = length * sizeof(float);
cudaMalloc(void**) &d_a, size);
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);

cudaMalloc(void**) &d_b, size);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Allocate storage for c in device memory
cudaMalloc(void**) &d_c, size);

// Invoke kernel with 128 blocks, each with 128 threads
vectorAdditionKernel<<<128, 128>>>(length, d_a, d_b, d_c);

// Copy GPU calculated c back to host memory
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
```
Recall the general Kernel thread steps:

1) Thread determines its location within the grid and its block
   a) gridDim variable of type dim3 which contains the dimensions of the grid of blocks
   b) blockIdx variable of type uint3 which contains its block index(es) within the grid
   c) blockDim variable of type dim3 which contains the dimensions of each block
   d) threadIdx variable of type uint3 which contains its thread index(es) within its block
2) Calculate its index of its first data value and stride to its next
3) Loops over and processes the data determined in step (2)
4) Writes result(s) to global memory

d) In the Kernel function of vectorAdditionKernel identify the above steps:

```c
__global__ void vectorAdditionKernel(int length, float * a, float * b, float * c)
{
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  int stride = blockDim.x * blockDim.x;

  while ( tid < length )
  {
    c[tid] = a[tid] + b[tid];
    tid += stride;
  } // end while
} // end vectorAdditionKernel
```

c) Explain the tid and stride calculations.

---

**Part C:** Using an editor on fermilab.cern.edu open the file `lab13/multA.cu` which contains a CUDA program that allows the user to enter an integer command-line argument: the matrix size. It creates three "2-D matrices" (A, B, and C) embedded as 1-D row-order arrays, fills matrices A and B with random floating-point numbers, and uses the GPU to perform matrix multiplication, i.e., C = A x B. The product is also calculated sequentially for comparison.

a) Why are the 2-D matrices embedded as 1-D arrays?

Before launching the kernel the dimensions of the grid and blocks are set-up using variables of type `dim3` as:

```c
// Set-up dimensions of blocks and grid
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid((C.width+BLOCK_SIZE-1)/BLOCK_SIZE, (C.height + BLOCK_SIZE - 1)/BLOCK_SIZE);

// Invoke kernel
matrixMultKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
```

b) Since `BLOCK_SIZE` is defined to be 16, what are the dimensions of a block of threads?

c) The `dimGrid` dimensions are calculated to tile blocks over the whole C array. Since C's dimensions might not be a multiple of the `BLOCK_SIZE`, explain each of the `dimGrid` dimension calculations:

   `(C.width + BLOCK_SIZE - 1)/BLOCK_SIZE`

   `(C.height + BLOCK_SIZE - 1)/BLOCK_SIZE`
d) In the `matrixMultKernel` function explain the `row` and `col` calculations:

```c
int row = threadIdx.y + blockIdx.y * blockDim.y;

int col = threadIdx.x + blockIdx.x * blockDim.x;
```

d) In the `matrixMultKernel` function what is the purpose of the if-statement:

```c
if (row < C.height && col < C.width) {
```

e) Why is it faster to use a local variable `cValue` instead updating `C.elements[row * C.width + col]` directly inside the for-loop?

**Part D:** Using an editor on `fermi1.cs.uni.edu` open the file `lab13/mmultB.cu` which contains another CUDA program to perform matrix multiplication, i.e., `C = A \times B`. Start by timing `mmultA` and `mmultB` on 1024x1024 size matrices.

a) What are the times? `mmultA time_` and `mmultB time_`

To understand `mmultB.cu` consider a sub-matrix of `C`, `Csub`. It needed corresponding rows from `A` and columns from `B`. It further splits these rows and columns into sub-matrices to improve memory efficiency.

```
Matrix A

Sub-matrices of A
needed to calculate Csub

Matrix B

Sub-matrices of B
needed to calculate Csub

Matrix C

Csub
```

b) What is the purpose of the for-loop: `for (int m=0; m < (A.width/BLOCK_SIZE); m++)`?

c) Why is the purpose of the first `__syncthreads()` call?

d) Why is the purpose of the second `__syncthreads()` call?

Submit `lab13.zip` containing question answers on the eLearning system
Homework #10

Due: Wednesday, Dec 5 at 5 PM (two weeks)

Learning Objectives:
- Design an efficient data-decomposition (i.e., block vs. cyclic) for a CUDA program in C.
- Write correct C program using CUDA library commands to initialize blocks of threads and synchronize their operation.
- Time various sizes blocks of threads

Homework #10 Description:
You are to design and implement the 2D SOR algorithm using CUDA on the Blue Waters Tesla K20X GPU cards. Your answers to the discussion question for this week should help you with the design.

When you get your program working, time the 4096 x 4096 array (interior dimensions) with threshold = 0.0001. Vary the DIM (i.e., BLOCK_SIZE) from 8, 16, and 32.

<table>
<thead>
<tr>
<th>HW #10 Timings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time in seconds for 4096 x 4096 array with threshold = 0.0001</td>
</tr>
<tr>
<td>DIM = 8 (8x8 blocks)</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Tesla K20X GPU Card</td>
</tr>
</tbody>
</table>

Submit hw10.zip containing a completed timing table and completed program (hw10.cu) on the eLearning system.
In addition to your CUDA by Example textbook, you might also benefit from online resources:

Referenced NVIDIA slides on pages 1 - 24 from Getting Started with CUDA:
http://www.nvidia.com/content/cudazone/download/Getting_STARTED_w_CUDA_Training_NVISION08.pdf

Referenced slides on pages 1, 28, 166-168, 189-191, 212-249 from (Dan Negrut at UW-Madison):
http://shej.wisc.edu/Courses/ME964/2012/Lectures/cudaNegrutWisconsin.pdf

On fermil.cs.uni.edu, output of deviceQuery command:

```
$ deviceQuery
CUDA Device Query...
There are 3 CUDA devices.

CUDA Device #0
Major revision number: 2
Minor revision number: 0
Name: Tesla C2070
Total global memory: 1341587456
Total shared memory per block: 49152
Total registers per block: 32768
Warp size: 32
Maximum memory pitch: 2147483647
Maximum threads per block: 1024
Maximum dimension 0 of block: 1024
Maximum dimension 1 of block: 1024
Maximum dimension 2 of block: 64
Maximum dimension 0 of grid: 65535
Maximum dimension 1 of grid: 65535
Maximum dimension 2 of grid: 65535
Clock rate: 1147000
Total constant memory: 65536
Texture alignment: 512
Concurrent copy and execution: Yes
Number of multiprocessors: 14
Kernel execution timeout: No

CUDA Device #1
Major revision number: 2
Minor revision number: 0
Name: Tesla C2070
...

CUDA Device #2
Major revision number: 2
Minor revision number: 0
Name: Tesla C2070
...
```

A CUDA program can dynamically select the GPU device using the command "cudaSetDevice(device_num)".

Your account also get you on fermil2.cs.uni.edu which has additional CUDA devices.

<table>
<thead>
<tr>
<th>Directions</th>
<th>Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run deviceQuery command to see CUDA devices on host</td>
<td>deviceQuery</td>
</tr>
<tr>
<td>Compile the CUDA code using nvcc (.cu is the CUDA file extension)</td>
<td>nvcc -o count3s count3s.cu</td>
</tr>
<tr>
<td>Run code on fermil.cs.uni.edu</td>
<td>./count3s</td>
</tr>
</tbody>
</table>
General CUDA architecture and memory layout:

**GPU Device - Tesla C2070**

- **Grid (1D)**
- **Block (0)**
  - Shared Memory
  - Registers
  - Thread(0)
  - Thread(1)
  - Local Memory
- **Block (1)**
  - Shared Memory
  - Registers
  - Thread(0)
  - Thread(1)
  - Local Memory

- Global Memory (not cached, read/write)
- Constant Memory (cached, read-only)
- Texture Memory (cached "odd" shapes, read-only)

General CUDA programming steps:

- **main:** (running on host)
  1) Reads data from file into host memory (e.g., main memory of fermi.cs.uni.edu)
  2) Allocate global memory on device via cudaMalloc for data
  3) Copy data from host memory to global memory via cudaMemcpy
  4) Kernel launches a grid of thread blocks where:
     a) All threads can access data in global memory
     b) Threads within a block cooperate via shared memory, e.g., ".shared." qualifier
     c) Threads within a block can synchronize, e.g., "__syncthreads();" -- barrier
     d) Threads in different blocks cannot cooperate much, e.g., atomic functions "atomicAdd()", __threadfence()
  5) Copy results from global memory to host memory via cudaMemcpy

General Kernel thread steps:

1) Thread determines its location within the grid and its block
   a) gridDim variable of type dim3 which contains the dimensions of the grid of blocks
   b) blockIdx variable of type uint3 which contains its block index(es) within the grid
   c) blockDim variable of type dim3 which contains the dimensions of each block
   d) threadIdx variable of type uint3 which contains its thread index(es) within its block
2) Calculate its index of its first data value and stride to its next
3) Loops over and processes the data determined in step (2)
4) Writes result(s) to global memory
```c
#define STEN (512*512*32)
#include <stdio.h>
#include <cuda.h>

static void HandleError( cudaError_t err, const char *file, int line ) {
    if (err != cudaSuccess) {
        printf( "%s in %s at line %d\n", cudaGetErrorString(err), file, line );
        exit(EXIT_FAILURE);
    }
}

#define HANDLE_ERROR( err ) (HandleError( err, __FILE__, __LINE__ ))

// Takes 20.3 ms on Tesla C2070 (fermi device #0), 29.6 ms serially on host, and 201.1 ms on older Tesla C1060
__global__ void count3s_kernelA(int *dev_array, int length, int *devCount) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int stride = blockDim.x * gridDim.x;
    while (i < length) {
        if (dev_array[i] == 3) {
            atomicAdd(devCount, 1);
        }
        i = i + stride;
    }
}

// Counts 3s kernel B
__global__ void count3s_kernelB(int *dev_array, int length, int *devCount) {
    int i = threadIdx.x + blockDim.x * gridDim.x;
    int offset = blockDim.x * gridDim.x;
    while (i < length) {
        if (dev_array[i] == 3) {
            atomicAdd(&blockCount, 1);
        }
        i = i + offset;
    }
}

// Tries to reduce contention for the devCount by having a "local" count called blockCount in the
// the shared memory of a block. Requires 1.2 compute capability for atomicAdd to block memory
// Takes 10.4 ms on Tesla C2070 (fermi device #0), 29.6 ms serially on host, and 9.8 ms on older Tesla C1060
__global__ void count3s_kernelC(int *dev_array, int length, int *devCount) {
    int i = threadIdx.x + blockDim.x * gridDim.x;
    int offset = blockDim.x * gridDim.x;
    int threadCount = 0;
    while (i < length) {
        if (dev_array[i] == 3) {
            threadCount ++;
            i = i + offset;
        }
    }
    threadCounts[threadIdx.x] = threadCount;
    __syncthreads();
    // binary-tree reduction, threadsPerBlock must be a power of 2
    i = blockIdx.x/2;
    while (i != 0) {
        if (threadIdx.x < i) {
            threadCounts[threadIdx.x] += threadCounts[threadIdx.x + i];
        }
        __syncthreads();
        i = i / 2;
    }
    if (threadIdx.x == 0) { 
        atomicAdd(devCount, threadCounts[0]);
    }
}
```
int main(int argc, char* argv[]) {
    int sequentialCount, i, length;
    int * myArray;

    // fcm1 device #0 is Tesla C2070 has 2.0 Compute Capability on fcm1
    HANDLE_ERROR(cudaSetDeviceProperties(&prop, 0));
    HANDLE_ERROR(cudaSetDevice(0));

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    // Generate data array with 10% 3s
    length = SIZE;
    printf("%d\n", length);
    myArray = (int*) malloc(length * sizeof(int));
    srand(5);
    for (i=0; i < length; i++) {
        myArray[i] = rand() % 10;
    } // end for i
    //* Do the actual work sequentially */
    cudaEventRecord(start, 0);
    sequentialCount = 0;
    for (i=0; i < length; i++) {
        if (myArray[i] == 3) {
            sequentialCount++;
        }
    } // end if
    // end for i
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    float elapsedTime;
    cudaEventElapsedTime(&elapsedTime, start, stop);
    printf("Time to count 3s on host: %.1fs\n", elapsedTime);
    printf("Number of 3's: %d\n", sequentialCount);

    // Do the work on GPU
    // Allocate memory on the GPU for the data
    cudaEventRecord(start, 0);
    int * dev_array;
    int * dev_count;
    cudaMalloc((void**)&dev_array, length * sizeof(int));
    cudaMemcpy(dev_array, myArray, sizeOf(int) * length, cudaMemcpyHostToDevice);
    cudaMalloc((void**)&dev_count, sizeof(int));
    cudaMemcpy(dev_count, 0, sizeof(int));

    // Determine device properties
    int blocks = prop.multiProcessorCount;
    count3s kernel<<<blocks, threadsPerBlock>>>(dev_array, length, dev_count);
    int devCount;
    cudaMemcpy(&devCount, dev_count, sizeof(int), cudaMemcpyDeviceToHost);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&elapsedTime, start, stop);
    printf("Time to count 3s on CUDA device: %.1fs\n", elapsedTime);
    if (sequentialCount == devCount) {
        printf("Results match at %d 3s\n", devCount);
    } else {
        printf("Results wrong with seq. count %d and GPU count %d\n", sequentialCount, devCount);
    } // end if
    cudaEventDestroy(start);
    cudaEventDestroy(stop);
    cudaFree(dev_count);
    cudaFree(dev_array);
    free(myArray);

    return 0;
} /* end main */
addVectors.cu

/* Programmer:  Mark Fienup
   File:        addVectors.cu
   Compile As: nvcc -o addVectors addVectors.cu
   Run As:      ./addVectors <length>
   Description: A CUDA solution to the add two 1-D vectors. */

#include <stdlib.h>
#include <stdio.h>
#include <cuda.h>

#define BLOCK_SIZE 16
#define TRUE 1
#define FALSE 0
#define BOOL int

static void HandleError( cudaError_t err, 
                          const char *file, 
                          int line ) {
    if (err != cudaSuccess) {
        printf( "%s in %s at line %d\n", cudaGetErrorString( err ), 
                 file, line );
        exit( EXIT_FAILURE );
    }
}
#define HANDLE_ERROR( err ) (HandleError( err, __FILE__, __LINE__))

// function prototypes
float* initializeVector(int length, float min, float max);
void printVector(int length, float *);
BOOL equalVectors(int length, float * vector1, float * vector2, float tolerance);
void seqVectorAddition(int length, float * a, float * b, float * c);

__global__ void vectorAdditionKernel(int length, float * a, float * b, float * c);

int main(int argc, char* argv[]) {
    float * a;
    float * b;
    float * c;
    float * seqC;
    int length;  // assume square
    size_t size;

    cudaDeviceProp prop;
    // ferml device #0 is Tesla C2070 has 2.0 Compute Capability
    HANDLE_ERROR(cudaGetDeviceProperties( &prop, 0));
    HANDLE_ERROR(cudaSetDevice(0));

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    if (argc != 2) {
        printf("Usage: %s <length>\n", argv[0]);
        exit(-1);
    }  // end if
sscanf(argv[1], "%d", &length);
printf("vector length = %d\n", length);

a = initializeVector(length, +1.0, -1.0);
b = initializeVector(length, +1.0, -1.0);
c = initializeVector(length, +1.0, -1.0);
seqC = initializeVector(length, +1.0, -1.0);

/* Do the actual work sequentially */
cudaEventRecord(start, 0);

seqVectorAddition(length, a, b, seqC);

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
printf("Time perform seq. vector addition on host: %3.1f ms\n", elapsedTime);

// Do the work on GPU
cudaEventRecord(start, 0);
float * d_a;
float * d_b;
float * d_c;

// Copy a and b to device memory
size = length * sizeof(float);
cudaMalloc((void**)&d_a, size);
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);

cudaMalloc((void**)&d_b, size);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Allocate storage for c in device memory
cudaMalloc((void**)&d_c, size);

// Invoke kernel with 128 blocks, each with 128 threads
vectorAdditionKernel<<<128, 128>>>(length, d_a, d_b, d_c);

// Copy GPU calculated c back to host memory
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);

cudaEventElapsedTime(&elapsedTime, start, stop);
printf("Time perform vector addition on CUDA device: %3.1f ms\n", elapsedTime);

if (equalVectors(length, c, seqC, 0.001)) {
  printf("Results match within a tolerance of %f\n", 0.001);
} else {
  printf("Results wrong: tolerance used %f\n", 0.001);
} // end if

// print if small enough
addVectors.cu

if (length < 10) {
    printf("\nc from CUDA: \\
    printVector(length, c);
    printf("\nseqC from host: \\
    printVector(length, seqC);
} // end if

cudaEventDestroy( start );
cudaEventDestroy( stop );
cudaFree( d_a );
cudaFree( d_b );
cudaFree( d_c );

free(a);
free(b);
free(c);
free(seqC);

return 0;
} /* end main */

// Each thread might compute many elements with each stride apart
__global__ void vectorAdditionKernel(int length, float * a, float * b, float * c) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = gridDim.x * blockDim.x;

    while ( tid < length ) {
        c[tid] = a[tid] + b[tid];
        tid += stride;
    } // end while
} // end vectorAdditionKernel

/************************************************************
Allocate vector of floats and initialized elements randomly.
 ************************************************************/
float* initializeVector(int length, float min, float max) {
    int i;
    float range, div;
    float * temp;

temp = (float *) malloc(sizeof(float) * length);

for (i=0; i < length; i++) {
    range = max - min;
    div = RAND_MAX / range;
    temp[i] = min + (rand() / div);
} // end for

return temp;
} // end initializeVector

/************************************************************
Prints vector to screen
addVectors.cu

void printVector(int length, float * v) {
    int i;

    for (i=0; i < length; i++) {
        printf("%8.4f ", v[i]);
    } // end for
    printf("\n");
} // end printVector

BOOL equalVectors(int length, float * vector1, float * vector2, float tolerance) {
    int i;

    for (i=0; i < length; i++) {
        if (fabsf(vector1[i] - vector2[i]) > tolerance) {
            printf("MISMATCH VALUES: %f %f\n", vector1[i], vector2[i]);
            return FALSE;
        } // end if
    } // end for
    return TRUE;
} // end equalVectors

void seqVectorAddition(int length, float * a, float * b, float * c) {
    int i;

    for (i=0; i < length; i++) {
        c[i] = a[i] + b[i];
    } /* end for (i */
} // end seqVectorAddition
mmultA.cu

/*
 * Programmer: Mark Fienup
 * File: mmultA.cu
 * Compile As: nvcc -o mmultA mmultA.cu
 * Run As: ./mmultA <matrix size>
 * Description: A CUDA solution to the matrix multiplication
 * Stores matrices in 1-D arrays in row-order.
 */

#include <stdlib.h>
#include <stdio.h>
#include <cuda.h>

#define BLOCK_SIZE 16
#define TRUE 1
#define FALSE 0
#define BOOL int

static void HandleError( cudaError_t err,
                        const char *file,
                        int line ) {
  if (err != cudaSuccess) {
    printf("%s in %s at line %d\n", cudaGetErrorString(err),
           file, line);
    exit(EXIT_FAILURE);
  }
}
#define HANDLE_ERROR(err) (HandleError(err, __FILE__, __LINE__ ))

typedef struct {
  int width;
  int height;
  float *elements;
} Matrix;

// function prototypes
Matrix initializeMatrix(int rows, int columns, float min, float max);
void printMatrix(Matrix M);
BOOL equalMatrices(const Matrix M1, const Matrix M2, float tolerance);
void segMatrixMult(const Matrix A, const Matrix B, Matrix C);

__global__ void matrixMultKernel(const Matrix A, const Matrix B, Matrix C);

int main(int argc, char* argv[]) {
  Matrix A, B, C, seqC;
  int n; // assume square
  size_t size;

  cudaDeviceProp prop;
  // fermi device #0 is Tesla C2070 has 2.0 Compute Capability
  HANDLE_ERROR(cudaGetDeviceProperties( &prop, 0));
  HANDLE_ERROR(cudaSetDevice(0));

  cudaEvent_t start, stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);

  if (argc != 2) {
    printf("Usage: %s <matrix size>\n", argv[0]);
    exit(-1);
    } // end if

    sscanf(argv[1], "%d", &n);
    printf("n = %d\n", n);

    A = initializeMatrix(n, n, +1.0, -1.0);
    B = initializeMatrix(n, n, +1.0, -1.0);
    C = initializeMatrix(n, n, +1.0, -1.0);
    seqC = initializeMatrix(n, n, +1.0, -1.0);

    /* Do the actual work sequentially */
    cudaMemcpy( A, B, seqC);
    cudaMemcpy( start, 0);
    cudaEventRecord( stop, 0);
    cudaEventSynchronize( stop);
    float elapsedTime;
    cudaEventElapsedTime( &elapsedTime, start, stop);
    printf("Time perform seq. Matrix Multiplication on host: %3.1f ms\n", elapsedTime);

    // Do the work on GPU
    cudaMemcpy( start, 0);
    Matrix d_A, d_B, d_C;

    // Copy A and B to device memory
    d_A = A;
    size = A.width * A.height * sizeof(float);
    cudaMemcpy((void**)&d_A.elements, A.elements, size,
                  cudaMemcpyHostToDevice);

    d_B = B;
    size = B.width * B.height * sizeof(float);
    cudaMemcpy((void**)&d_B.elements, B.elements, size,
                  cudaMemcpyHostToDevice);

    // Allocate storage for C in device memory
    d_C = C;
    size = C.width * C.height * sizeof(float);
    cudaMemcpy((void**)&d_C.elements, C.elements, size);

    // Set-up dimensions of blocks and grid
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid((C.width + BLOCK_SIZE - 1) / BLOCK_SIZE, (C.height + BLOCK_SIZE - 1) / BLOCK_SIZE);

    // Invoke kernel
    matrixMultKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

    // Copy GPU calculated C back to host memory
    cudaMemcpy(C.elements, d_C.elements, size, cudaMemcpyDeviceToHost);

    cudaMemcpy( stop, 0);
    cudaEventSynchronize( stop);
    cudaMemcpy( &elapsedTime, start, stop);
printf( "Time perform Matrix Multiplication on CUDA device: \%3.1f ms\n", elapsedTime);

if (equalMatrices(C, seqC, 0.001)) {
    printf("Results match within a tolerance of \%f\n", 0.001);
} else {
    printf("Results wrong: tolerance used \%f\n", 0.001);
} // end if

// print if small enough
if (C.width < 10 && C.height < 10) {
    printf("\nC from CUDA:\n");
    printMatrix(C);
    printf("nseqC from host:\n");
    printMatrix(seqC);
} // end if

cudaEventDestroy( start );
cudaEventDestroy( stop );
cudaFree( d_A.elements );
cudaFree( d_B.elements );
cudaFree( d_C.elements );

free(A.elements);
free(B.elements);
free(C.elements);
free(seqC.elements);

return 0;
} /* end main */

// Each thread computes one element of C by accumulating results in local CVal
__global__ void matrixMultKernel(const Matrix A, const Matrix B, Matrix C) {
    int row = threadIdx.y + blockIdx.y * blockDim.y;
    int col = threadIdx.x + blockIdx.x * blockDim.x;
    int k;
    float CValue;

    if (row < C.height && col < C.width) {
        CValue = 0.0;
        for (k = 0; k < A.width; k++) {
            CValue += A.elements[row * A.width + k] * B.elements[k * B.width + col];
        } // end for
        C.elements[row * C.width + col] = CValue;
    } // end if
} // end matrixMultKernel

/********************************************
Allocate matrix of floats at 1D array and initialized elements randomly.
*********************************************/
Matrix initializeMatrix(int rows, int columns, float min, float max) {
    int i;
    float range, div;
    Matrix M;

M.width = columns;
M.height = rows;
M.elements = (float *) malloc(sizeof(float) * rows * columns);

for (i=0; i < rows*columns; i++) {
    range = max - min;
    div = RAND_MAX / range;
    M.elements[i] = min + (rand() / div);
} // end for

return M;
} // end initializeMatrix

/*******************************************************************************/
/* Prints matrix to screen */
/*******************************************************************************/
void printMatrix(Matrix M) {
    int r, c;

    for (r=0; r < M.height; r++) {
        for (c=0; c < M.width; c++) {
            printf("%8.4f ", M.elements[r * M.width + c]);
        } // end for (c...
        printf("\n");
    } // end for (r...
} // end printMatrix

/*******************************************************************************/
/* Compares elements of M1 and M2 to see whether all match within the given */
/* tolerance. Returns TRUE or FALSE accordingly. */
/*******************************************************************************/
BOOL equalMatrices(const Matrix M1, const Matrix M2, float tolerance) {
    int i;

    if (M1.width != M2.width || M1.height != M2.height) {
        return FALSE;
    } // end if

    for (i=0; i < M1.width*M1.height; i++) {
        if (fabsf(M1.elements[i] - M2.elements[i]) > tolerance) {
            printf("MISMATCH VALUES: %f %f\n", M1.elements[i], M2.elements[i]);
            return FALSE;
        } // end if
    } // end for

    return TRUE;
} // end equalMatrices

/*******************************************************************************/
/* Sequentially computes matrix multiplication of C = A * B with */
/* C being returned. */
/*******************************************************************************/
void seqMatrixMult(const Matrix A, const Matrix B, Matrix C) {
int i, j, k;
float CValue;

for (i=0; i < C.height; i++) {
    for (j=0; j < C.width; j++) {
        CValue = 0.0;
        for (k=0; k < A.width; k++) {
            CValue += A.elements[i*A.width + k] * B.elements[k*B.width + j];
        } /* end for (k */
        C.elements[i*C.width + j] = CValue;
    } /* end for (j */
} /* end for (i */

} // end seqMatrixMult
mmultB.cu

/*
 * Programmer:  Mark Fienup
 * File:        mmultB.cu
 * Compile As:  nvcc -o mmultB mmultB.cu
 * Run As:      ./mmultB <matrix size>
 * Description: A CUDA solution to the matrix multiplication
 *               Stores matrices in 1-D arrays in row-order.
 */

#include <stdlib.h>
#include <stdio.h>
#include <cuda.h>

#define BLOCK_SIZE 16
#define TRUE 1
#define FALSE 0
#define BOOL int

static void HandleError( cudaError_t err,
                          const char *file,
                          int line ) {
    if (err != cudaSuccess) {
        printf( "%s in %s at line %d\n", cudaGetErrorString( err ),
                file, line );
        exit( EXIT_FAILURE );
    }
}
#define HANDLE_ERROR( err ) (HandleError( err, __FILE__, __LINE__ ))

typedef struct {
    int width;
    int height;
    int stride;
    float * elements;
} Matrix;

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

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

// Get the BLOCK_SIZE x BLOCK_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
__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];
mmultB.cu

return Asub;
} // end GetSubMatrix

// function prototypes
Matrix initializeMatrix(int rows, int columns, float min, float max);
void printMatrix(Matrix M);
BOOL equalMatrices(const Matrix M1, const Matrix M2, float tolerance);
void seqMatrixMult(const Matrix A, const Matrix B, Matrix C);

__global__ void matrixMultKernel(const Matrix A, const Matrix B, Matrix C);

int main(int argc, char** argv) {
    Matrix A, B, C, seqC;
    int n; // assume square
    size_t size;

cudaDeviceProp prop;
    // fermi device #0 is Tesla C2070 has 2.0 Compute Capability
    HANDLE_ERROR(cudaGetDeviceProperties(&prop, 0));
    HANDLE_ERROR(cudaSetDevice(0));

cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    if (argc != 2) {
        printf("Usage: %s <matrix size>
", argv[0]);
        exit(-1);
    } // end if

    sscanf(argv[1], "%d", &n);
    printf("n = %d\n", n);
    A = initializeMatrix(n, n, +1.0, -1.0);
    B = initializeMatrix(n, n, +1.0, -1.0);
    C = initializeMatrix(n, n, +1.0, -1.0);
    seqC = initializeMatrix(n, n, +1.0, -1.0);

    /* Do the actual work sequentially */
    cudaEventRecord(start, 0);

    seqMatrixMult(A, B, seqC);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    float elapsedTime;
    cudaEventElapsedTime(&elapsedTime, start, stop);
    printf("Time perform seq. Matrix Multiplication on host: %3.1f ms\n", elapsedTime);

    // Do the work on GPU
    cudaEventRecord(start, 0);
    Matrix d_A, d_B, d_C;

    // Copy A and B to device memory
    d_A = A;
    d_A.stride = A.width;
size = A.width * A.height * sizeof(float);
cudaMallocVoid** &d_A.elements, size);
cudaMemcpy(d_A.elements, A.elements, size, cudaMemcpyHostToDevice);

d_B = B;
d_B.stride = B.width;
size = B.width * B.height * sizeof(float);
cudaMallocVoid** &d_B.elements, size);
cudaMemcpy(d_B.elements, B.elements, size, cudaMemcpyHostToDevice);

// Allocate storage for C in device memory
d_C = C;
d_C.stride = C.width;
size = C.width * C.height * sizeof(float);
cudaMallocVoid** &d_C.elements, size);

// Set-up dimensions of blocks and grid
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid((B.width + BLOCK_SIZE - 1) / BLOCK_SIZE, (A.height + BLOCK_SIZE - 1) / BLOCK_SIZE);

// Invoke kernel
matrixMultKernel<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

// Copy GPU calculated C back to host memory
cudaMemcpy(C.elements, d_C.elements, size, cudaMemcpyDeviceToHost);

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);

cudaEventElapsedTime(&elapsedTime, start, stop);
printf("Time perform Matrix Multiplication on CUDA device: %3.1f ms\n", elapsedTime);

if (equalMatrices(C, seqC, 0.001)) {
    printf("Results match within a tolerance of %f\n", 0.001);
} else {
    printf("Results wrong: tolerance used %f\n", 0.001);
} // end if

if (C.width < 10 && C.height < 10) {
    printf("C from CUDA: \n");
    printMatrix(C);
    printf("seqC from host: \n");
    printMatrix(seqC);
} // end if

cudaEventDestroy(start);
cudaEventDestroy(stop);
cudaFree(d_A.elements);
cudaFree(d_B.elements);
cudaFree(d_C.elements);

free(A.elements);
free(B.elements);
free(C.elements);
free(seqC.elements);
return 0;
} /* end main */

// Each thread computes one element of C by accumulating results in local CValue
__global__ void matrixMultKernel(const Matrix A, const Matrix B, Matrix C) {

// 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 = 0.0;

// Thread row and column within Csub
int row = threadIdx.y;
int col = threadIdx.x;

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

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

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

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

// Load Asub and Bsub from device memory to 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);

// Synchronize to make sure the sub-matrices are loaded
// before starting to computation of Csub
__syncthreads();

// Multiply Asub and Bsub together
for (int k = 0; k < BLOCK_SIZE; k++) {
    CValue += As[row][k] * Bs[k][col];
} /* end for */

// Synchronize to make sure that the preceeding
// computation is done before loading two new
// sub-matrices of A and B in the next iteration
__syncthreads();
mmultB.cu

} // end for (m

// Write Csub to device memory
// Each thread writes one element
SetElement(Csub, row, col, CValue);

} // end matrixMultKernel

/************************************************************
Allocate matrix of floats at 1D array and initialized elements randomly.
************************************************************/
Matrix initializeMatrix(int rows, int columns, float min, float max) {
    int i;
    float range, div;
    Matrix M;

    M.width = columns;
    M.height = rows;
    M.elements = (float *) malloc(sizeof(float) * rows * columns);

    for (i=0; i < rows*columns; i++) {
        range = max - min;
        div = RAND_MAX / range;
        M.elements[i] = min + (rand() / div);
    } // end for

    return M;
} // end initializeMatrix

/************************************************************
Prints matrix to screen
************************************************************/
void printMatrix(Matrix M) {
    int r, c;

    for (r=0; r < M.height; r++) {
        for (c=0; c < M.width; c++) {
            printf("%8.4f ", M.elements[r * M.width + c]);
        } // end for (c...
        printf("\n");
    } // end for (r...

} // end printMatrix

/************************************************************
Compares elements of M1 and M2 to see whether all match within
the given tolerance. Returns TRUE or FALSE accordingly.
************************************************************/
BOOL equalMatrices(const Matrix M1, const Matrix M2, float tolerance) {
    int i;

    if (M1.width != M2.width || M1.height != M2.height) {
        return FALSE;
    } // end if
for (i=0; i < M1.width*M1.height; i++) {
    if (fabsf(M1.elements[i] - M2.elements[i]) > tolerance) {
        printf("MISMATCH VALUES: %f %f\n", M1.elements[i], M2.elements[i]);
        return FALSE;
    } // end if
} // end for
return TRUE;
}

**************************************************************************
Sequentially computes matrix multiplication of C = A * B with
C being returned.
**************************************************************************
void seqMatrixMult(const Matrix A, const Matrix B, Matrix C) {
    int i, j, k;
    float CValue;

    for (i=0; i < C.height; i++) {
        for (j=0; j < C.width; j++) {
            CValue = 0.0;
            for (k=0; k < A.width; k++) {
                CValue += A.elements[i*A.width + k] * B.elements[k*B.width + j];
            } // end for (k */
            C.elements[i*C.width + j] = CValue;
        } // end for (j */
    } // end for (i */
} // end seqMatrixMult
Learning Objectives:
- Apply the CUDA `atomicAdd` command to perform mutually exclusive update of a shared variable
- Apply CUDA command `__syncthreads()` to perform a barrier-synchronization among a thread block
- Time CUDA events to evaluate performance of various levels of CUDA memory

To start the lab:
- read Chapter 9 of the CUDA by Example textbook
- watch the Lab 14 Video on the eLearning system
- download `lab14.zip` from the eLearning system to your computer
- upload `lab14.zip` to student.cs.uni.edu using a secure ftp client (winSCP, FileZilla, scp, etc.)
- log-on to Blue Waters log-in node using Putty/ssh
- use scp to copy `lab14.zip` from student.cs.uni.edu Blue Waters
- unzip `lab14.zip` and cd `lab14`

**NOTE:** To use CUDA on Blue Waters you must first load the CUDA module into your environment by the command: `module load cudatoolkit`

**Part A:** Using an editor open the file `lab14/count3sA.cu` which contains a CUDA program that allows the user to enter an integer command-line argument in `qsub count3sA aprun` line: the length of the array. It creates a 1-dimensional array of that length (myArray), fills the array with random integers between 0 - 9, and uses the GPU to count the occurrences of 3 in the arrays. The count of 3’s is also calculated sequentially for comparison.

Compile the program using: `nvcc -o count3sA count3sA.cu` and run the program with lengths of 100000, 1000000, and 10000000 (i.e., uncommenting the appropriate `qsub.count3sA aprun` line) by: `qsub qsub.count3sA`

Complete the timing table:

<table>
<thead>
<tr>
<th>Array Length</th>
<th>Sequential Count of 3's (in ms)</th>
<th>CUDA Count of 3's Kernel A (in ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,000,000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10,000,000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The CUDA kernel A that counts the 3’s is defined as:

```
__global__ void count3s_kernelA(int * dev_array, int length, int * devCount) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;
    while (i < length) {
        if (dev_array[i] == 3) {
            atomicAdd(devCount, 1);
        }
    }
    i = i + stride;
}
```

Recall the general CUDA architecture and memory layout:
a) Why is sequential vector addition faster for a length of 100,000?

b) As the length gets bigger, why is the CUDA version faster?

c) The `atomicAdd( devCount, 1 );` increments the `devCount` variable mutually exclusive across all threads. In what level of CUDA memory (host memory, device global memory, shared memory of a block, local memory of a thread) is `devCount` located?

d) How much contention is there for `devCount` (i.e., which threads are trying to increment it)?

**Part B:** Using an editor open the file `lab14/count3sB.cu` which contains another CUDA program that counts the occurrences of 3 in the arrays.

Compile the program using: `nvcc -o count3sB count3sB.cu`

and run the program with lengths of 100000, 1000000, and 10000000 (i.e., uncommenting the appropriate `qsub.count3sB aprun` line) by: `qsub qsub.count3sB`

Complete the timing table:

<table>
<thead>
<tr>
<th>Array Length</th>
<th>Sequential Count of 3's (in ms)</th>
<th>CUDA Count of 3's <strong>Kernel B</strong> (in ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,000,000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10,000,000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The CUDA kernel B that counts the 3's is defined as:

```c
__global__ void count3s_kernelB(int * dev_array, int length, int * devCount) {
    __shared__ int blockCount;
    if (threadIdx.x == 0) {
        blockCount = 0;
    } // end if
    __syncthreads();

    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;

    while (i < length) {
        if (dev_array[i] == 3) {
            atomicAdd( &blockCount, 1);
        } // end if
        i = i + stride;
    } // end while
    __syncthreads();

    if (threadIdx.x == 0) {
        atomicAdd(devCount, blockCount);
    } // end if
} // end count3s_kernelB
```

a) The `atomicAdd( &blockCount, 1 );` increments the `blockCount` variable mutually exclusive across all threads. In what level of CUDA memory (host memory, device global memory, shared memory of a block, local memory of a thread) is `blockCount` located?
b) How much contention is there for blockCount (i.e., which threads are trying to increment it)?

c) The CUDA command __syncthreads() perform a barrier-synchronization among all the thread in a block. Explain the purpose of each __syncthreads() in Kernel B.

d) Explain why Kernel B is faster than Kernel A.

Part C: Using an editor on fermil.cs.uni.edu open the file lab14/count3sC.cu which contains yet another CUDA program that counts the occurrences of 3 in the arrays.

Compile the program using: nvcc -o count3sC count3sC.cu

and run the program with lengths of 100000, 1000000, and 10000000 (i.e., uncommenting the appropriate qsub.count3sC aprun line) by: qsub qsub.count3sC

Complete the timing table:

<table>
<thead>
<tr>
<th>Array Length</th>
<th>Sequential Count of 3's (in ms)</th>
<th>CUDA Count of 3's Kernel C (in ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,000,000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10,000,000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The CUDA kernel C that counts the 3's is defined as:

```c
__global__ void count3s_kernelC(int * dev_array, int length, int * devCount) {
    __shared__ int threadCounts[threadsPerBlock];

    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;
    int threadCount = 0;

    while (i < length) {
        if (dev_array[i] == 3) {
            threadCount += 1;
        } // end if
        i = i + stride;
    } // end while

    threadCounts[threadIdx.x] = threadCount;
    __syncthreads();

    // binary-tree reduction, threadsPerBlock must be a power of 2
    i = blockDim.x/2;
    while (i != 0) {
        if (threadIdx.x < i) {
            threadCounts[threadIdx.x] += threadCounts[threadIdx.x + i];
        } // end if
        __syncthreads();
        i = i / 2;
    } // end while

    if (threadIdx.x == 0) {
        atomicAdd(devCount, threadCounts[0]);
    } // end if
} // end count3s_kernelC
```
a) The top while-loop increments the `threadCount` variable without using an `atomicAdd`. Why is an `atomicAdd` not needed here? (Hint: Consider which level of CUDA memory holds `threadCount`)

b) How much contention is there for `threadCount` (i.e., which threads are trying to increment it)?

c) Explain the purpose of the `__syncthreads()` immediately after:

   ```
   threadCounts[threadIdx.x] = threadCount;
   ```

d) Explain the purpose of the `__syncthreads()` within the while-loop that performs the binary-tree reduction.

e) How much contention is there for `devCount` (i.e., which/how many threads are trying to increment it)?

f) In summary, explain why Kernel C is faster than Kernel B.

**Part D:** Using an editor open the file `lab14/count3D.cu` which contains a partial Kernel D (top “half” of Kernel C). You need to complete this Kernel D by:
- using an `atomicAdd` to update `blockCount` by each thread’s `threadCounts`
- using an `atomicAdd` to update `devCount` by each block’s `blockCounts`
- using `__syncthreads()` (barrier-synchronization) at needed

a) What is the time of your Kernel D on an array of length 10,000,000?

b) Explain why Kernel D is faster/slower than Kernel C.

Submit `lab14.zip` containing question answers and completed program on the e-Learning system.
Comp. Arch.  

Week 14 Display Video

(Appendix A Material -- creation of a mutex lock what can be used to update a shared variable in global memory by all threads)

We’ll use the CUDA Compare-And-Swap atomic command:

```c
int atomicCAS(int * address, int compare, int val);
```

This atomic operation performs:
1. reads the value pointed at by address
2. compares this value to see if it is equal to the parameter compare. If they are equal, then change the memory pointed at by address to val; otherwise don’t change the value pointed at by the address.
3. returns the initial value read in step (1)

We can summarize the atomicCAS’s atomic operation by psuedo-code:

```c
old = *address
if *address == compare then
    *address = val
return old
```

To implement a mutex lock operation, we’ll initial a global memory location pointed at by mutex to 0 (“unlocked”).

```
mutex

0
```

To lock the mutex all threads can perform the code:

```c
while( atomicCAS( mutex, 0, 1 ) != 0 );
```

The first thread to perform the lock will compare the mutex value of 0 to its compare parameter of 0 which will replace the mutex value by 1 with atomicCAS returning 0. Thus, it will drop out of the while-loop.

After the mutex is locked (i.e., has the value 1), other threads performing the atomicCAS will get a return value of 1 and busy-wait inside the while-loop.

We’ll use the CUDA “atomic exchange” command when unlocking the mutex. The syntax of this operation is:

```c
int atomicExch(int * address, int val);
```

We can summarize the atomicExch semantics by psuedo-code:

```c
old = *address
*address = val
return old
```

The thread holding the mutex lock can unlock the mutex (i.e. set it back to 0) by performing the code:

```c
atomicExch( mutex, 0 );
```

We can create a Lock struct with functions: lock and unlock. The following example uses a Lock struct to complete the summation of a 1D array of floats.
/* Programmer: Mark Fienup
 File: sum1D_Array_Floats.cu
 Compile As: nvcc -o sum1D_Array_Floats.cu
 Run As: ./sum1D

 Description: A CUDA solution to sum a 1D array of floats.
 Uses a user-defined lock structure with lock and unlock functions.
 */

#define SIZE (512*512*32)
#define threadsPerBlock 512

#include <stdlib.h>
#include <stdio.h>
#include <cuda.h>

static void HandleError( cudaError_t err, const char *file, int line )
{
    if (err != cudaSuccess) {
        printf( "%s in %s at line %d\n", cudaGetErrorMessage( err ), file, line );
        exit( EXIT_FAILURE );
    }
}

#define HANDLE_ERROR( err ) (HandleError( err, __FILE__, __LINE__ ))

struct Lock {
    int *mutex;
    Lock( void ) {
        HANDLE_ERROR( cudaMalloc( (void**)&mutex, sizeof(int) ) );
        HANDLE_ERROR( cudaMemcpy( mutex, 0, sizeof(int) ) );
    }

    ~Lock( void ) {
        cudaFree( mutex );
    }

    __device__ void lock( void ) {
        while( atomicCAS( mutex, 0, 1 ) != 0 );
        __threadfence();
    }

    __device__ void unlock( void ) {
        __threadfence();
        atomicExch( mutex, 0 );
    }
};

__global__ void sum1D_floats(Lock lock, float * dev_array, int length, float * devSum) {
    __shared__ float threadSums[threadsPerBlock];

    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;
    float threadSum = 0.0;

    while (i < length) {
        threadSum += dev_array[i];
        i = i + stride;
    } // end while

    threadSums[threadIdx.x] = threadSum;
    __syncthreads();

    // binary-tree reduction, threadsPerBlock must be a power of 2
    i = blockDim.x/2;
while (i != 0) {
    if (threadIdx.x < i) {
        threadSums[threadIdx.x] += threadSums[threadIdx.x + i];
    } // end if
    __syncthreads();
    i = i / 2;
} // end while

if (threadIdx.x == 0) {
    lock.lock(); /* wait until lock acquired */
    *devSum += threadSums[0];
    lock.unlock(); /* release the lock after updating devSum */
} // end if
} // end sum1D_floats

int main(int argc, char* argv[]) {
    float sequentialSum;
    int i, length;
    float * myHarray;

    cudaDeviceProp prop;

    // force device #2 is Tesla C2070 has 2.0 Compute Capability on fermi
    HANDLE_ERROR(cudaGetDeviceProperties( &prop, 2));
    HANDLE_ERROR(cudaSetDevice(2));

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    if (argc != 2) {
        printf("Usage: %s <array size>\n", argv[0]);
        exit(-1);
    } // end if

    // Generate data array with 10% 3s
    sscanf(argv[1], "%d", &length);
    printf("Array length = %d\n", length);
    myArray=(float *) malloc(length*sizeof(float));
    srand(5);
    for (i=0; i < length; i++) {
        myArray[i] = float( rand() % 10 );
    } // end for i

    /* Do the actual work sequentially */
    cudaEventRecord(start, 0);
    sequentialSum = 0;
    for (i=0; i < length; i++) {
        sequentialSum += myArray[i];
    } // end for i
    cudaEventRecord(stop, 0);
    cudaEventSynchronize( stop);
    float elapsedTime;
    cudaEventElapsedTime( &elapsedTime, start, stop);
    printf("Time to count 3s on host: %.3f ms\n", elapsedTime);
    printf("Array sum sequentially %f\n", sequentialSum);

    // Do the work on GPU
    // allocate memory on the GPU for the data
    cudaEventRecord(start, 0);
    float * dev_array;
    float * dev_sum;
    cudaMalloc((void**) &dev_array, length*sizeof(float));
    cudaMemcpy(dev_array, myArray, sizeof(float)*length, cudaMemcpyHostToDevice);
    cudaMalloc((void**) &dev_sum, sizeof(float));
cudaMemset(dev_sum, 0, sizeof(float));

// Determine device properties
Lock lock;
int blocks = prop.multiProcessorCount;
sum1D_floats<<blocks*2,threadsPerBlock>>>(lock, dev_array, length, dev_sum);

float devSum;
cudaMemcpy(&devSum, dev_sum, sizeof(float), cudaMemcpyDeviceToHost);

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);

cudaEventElapsedTime(&elapsedTime, start, stop);
printf( "Time to sum array on CUDA device: %3.1f ms\n", elapsedTime);

if (fabs(sequentialSum - devSum) < 0.001) {
    printf("Results match at %f 3s\n", devSum);
} else {
    printf("Results wrong with seq. sum %f and GPU sum %f.\n", sequentialSum, devSum);
} // end if

cudaEventDestroy(start);
cudaEventDestroy(stop);
cudaFree(dev_sum);
cudaFree(dev_array);

free(myArray);

return 0;
} /* end main */
A CUDA stream represents a FIFO queue of GPU operations:
- kernel launches
- memory copies between host and device memories
- event starts and stops
Think of each stream as a task on the GPU

Streams allow parallel tasks to run simultaneously on a GPU. Speedup usually achieved by overlapping:
- kernel computation on the GPU by one stream
- the copying of data between the host-memory and device-memory by another stream
(NOTE: GPU card must support “Concurrent copy and execution” with deviceQuery command. Our Tesla C2070 GPUs support this.)

To utilize “concurrent copy” the data array in the host-memory must be allocated using instead of C’s malloc command. The cudaHostAlloc() function ensures that the memory is page-locked/pinned so the OS cannot swap/page it out to disk. Thus, the GPU can use DMA to copy data to/from the host-memory.

Even without streams, the book’s copy_time.cu (see attached) program on fermi1.cs.uni.edu shows an improvement using arrays that have been allocated using cudaHostAlloc().

<table>
<thead>
<tr>
<th>Direction of the Copy</th>
<th>Copy Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>array allocated using malloc</td>
</tr>
<tr>
<td>copy “up” from host to device memory</td>
<td>4856.9 MB/sec</td>
</tr>
<tr>
<td>copy “down” from device to host memory</td>
<td>4170.6 MB/sec</td>
</tr>
</tbody>
</table>

The textbook stream example uses two 1D array of (random) floats a and b to calculate 1D array c:

Our previous CUDA approach as a single stream was:
1. whole copy of a to device memory
2. whole copy of b to device memory
3. kernel computation of c
4. whole copy of c to host memory

To achieve parallelism using streams, the copying and computation are partitioned into 20 blocks:

Each stream receives 20 blocks of work. Stream 0 gets the even blocks and stream 1 gets the odd blocks. Two versions of the program are discussed which differ by the order of GPU operations are issued to the streams:
basic_double_streams.cu algorithm:
for (i = 0; i < 20; i = i + 2) do
  copy a_i to device memory by stream 0
  copy b_i to device memory by stream 0
  kernel to calculate c_i by stream 0
  copy c_i to host memory by stream 0
  copy a_{i+1} to device memory by stream 1
  copy b_{i+1} to device memory by stream 1
  kernel to calculate c_{i+1} by stream 1
  copy c_{i+1} to host memory by stream 1
end for

basic_double_streams_correct.cu algorithm:
for (i = 0; i < 20; i = i + 2) do
  copy a_i to device memory by stream 0
  copy a_{i+1} to device memory by stream 1
  copy b_i to device memory by stream 0
  copy b_{i+1} to device memory by stream 1
  kernel to calculate c_i by stream 0
  kernel to calculate c_{i+1} by stream 1
  copy c_i to host memory by stream 0
  copy c_{i+1} to host memory by stream 1
end for

Textbook reported timings on GeForce GTX 285:
- single stream 62 ms
- basic_double_streams.cu 61 ms
- basic_double_streams_correct.cu 48 ms

Authors explain the results based on the order that stream operations enter to GPU queues to handle:
- Copy Engine queue of pending cudaMemcpyAsync() calls
- Kernel Engine queue of pending kernel calls

Figure 10.2 shows the mapping of basic_double_streams.cu into GPU queues.

Since the cudaMemcpy C of stream 0 is in the Copy Engine queue before the cudaMemcpy A and cudaMemcpy B of stream 1, then the kernel A of stream 1 cannot be run in parallel. Figure 10.3 shows the execution timeline for this example.
Figure 10.4 shows the execution timeline of basic_double_streams.cu into GPU queues. The arrows show the dependencies between the Copy Engine and the Kernel Engine.

The alternative basic_double_streams_correct.cu program produces an execution timeline as shown in Figure 10.5.

Notice the overlap of the memory copies for one stream with the computation of the other stream.

This all seems to make perfect sense, except timing on the Tesla C2070 GPU cards on fermi1.cs.uni.edu are:
- single stream 58 ms
- basic_double_streams.cu 44 ms
- basic_double_streams_correct.cu 53 ms

I'm assuming that "improvements" in the hardware queuing on the Tesla C2070 GPU card was able to find more task level parallelize between the two streams. (This is just a conjecture on my part!)
#include "book.h"

#define SIZE (64*1024*1024)

float cuda_malloc_test( int size, bool up ) {
    cudaEvent_t start, stop;
    int *a, *dev_a;
    float elapsedTime;

    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );

    a = (int*)malloc( size * sizeof( *a ) );
    HANDLE_NULL( a );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
                       size * sizeof( *dev_a ) ) );

    HANDLE_ERROR( cudaEventRecord( start, 0 ) );
    for (int i=0; i<100; i++) {
        if (up)
            HANDLE_ERROR( cudaMemcpy( dev_a, a,
                           size * sizeof( *dev_a ),
                           cudaMemcpyHostToDevice ) );
        else
            HANDLE_ERROR( cudaMemcpy( a, dev_a,
                           size * sizeof( *dev_a ),
                           cudaMemcpyDeviceToHost ) );
    }
    HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
    HANDLE_ERROR( cudaEventSynchronize( stop ) );
    HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                      start, stop ) );

    free( a );
    HANDLE_ERROR( cudaFree( dev_a ) );
    HANDLE_ERROR( cudaEventDestroy( start ) );
    HANDLE_ERROR( cudaEventDestroy( stop ) );

    return elapsedTime;
}

float cuda_host_alloc_test( int size, bool up ) {
    cudaEvent_t start, stop;
    int *a, *dev_a;
    float elapsedTime;

    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );

    HANDLE_ERROR( cudaHostAlloc( (void**)&a,
                       size * sizeof( *a ),
                       cudaMemcpyHostAllocDefault ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
                       size * sizeof( *dev_a ) ) );
HANDLE_ERROR( cudaEventRecord( 'start', 0 ) );
for (int i=0; i<100; i++) {
    if (up)
        HANDLE_ERROR( cudaMemcpy( dev_a, a,
                                    size * sizeof( *a ),
                                    cudaMemcpyHostToDevice ) );
    else
        HANDLE_ERROR( cudaMemcpy( a, dev_a,
                                    size * sizeof( *a ),
                                    cudaMemcpyDeviceToHost ) );
}
HANDLE_ERROR( cudaEventRecord( 'stop', 0 ) );
HANDLE_ERROR( cudaEventSynchronize( 'stop' ) );
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                      start, stop ) );
HANDLE_ERROR( cudaFreeHost( a ) );
HANDLE_ERROR( cudaFree( dev_a ) );
HANDLE_ERROR( cudaEventDestroy( start ) );
HANDLE_ERROR( cudaEventDestroy( stop ) );

return elapsedTime;

int main( void ) {
    float elapsedTime;
    float MB = (float)100*SIZE*sizeof(int)/1024/1024;

    // try it with cudaMalloc
    elapsedTime = cuda_malloc_test( SIZE, true );
    printf( "Time using cudaMalloc: %3.1f ms\n",
            elapsedTime );
    printf( "\tMB/s during copy up: %3.1f\n",
            MB/(elapsedTime/1000) );
    elapsedTime = cuda_malloc_test( SIZE, false );
    printf( "Time using cudaMalloc: %3.1f ms\n",
            elapsedTime );
    printf( "\tMB/s during copy down: %3.1f\n",
            MB/(elapsedTime/1000) );

    // now try it with cudaHostAlloc
    elapsedTime = cuda_host Alloc_test( SIZE, true );
    printf( "Time using cudaHostAlloc: %3.1f ms\n",
            elapsedTime );
    printf( "\tMB/s during copy up: %3.1f\n",
            MB/(elapsedTime/1000) );
    elapsedTime = cuda_host Alloc_test( SIZE, false );
    printf( "Time using cudaHostAlloc: %3.1f ms\n",
            elapsedTime );
    printf( "\tMB/s during copy down: %3.1f\n",
            MB/(elapsedTime/1000) );
}
#include "book.h"

#define N    (1024*1024)
#define FULL_DATA_SIZE  (N*20)

__global__ void kernel( int *a, int *b, int *c ) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < N) {
    int idx1 = (idx + 1) % 256;
    int idx2 = (idx + 2) % 256;
    float as = (a[idx] + a[idx1] + a[idx2]) / 3.0f;
    float bs = (b[idx] + b[idx1] + b[idx2]) / 3.0f;
    c[idx] = (as + bs) / 2;
  }
}

int main( void ) {
  cudaDeviceProp prop;
  int whichDevice;
  HANDLE_ERROR( cudaGetDevice( &whichDevice ) );
  HANDLE_ERROR( cudaGetDeviceProperties( &prop, whichDevice ) );
  if (!prop.deviceOverlap) {
    printf( "Device will not handle overlaps, so no speed up from streams\n" );
    return 0;
  }

  cudaEvent_t start, stop;
  float elapsedTime;

  cudaStream_t stream0, stream1;
  int *host_a, *host_b, *host_c;
  int *dev_a0, *dev_b0, *dev_c0;
  int *dev_a1, *dev_b1, *dev_c1;

  // start the timers
  HANDLE_ERROR( cudaEventCreate( &start ) );
  HANDLE_ERROR( cudaEventCreate( &stop ) );

  // initialize the streams
  HANDLE_ERROR( cudaStreamCreate( &stream0 ) );
  HANDLE_ERROR( cudaStreamCreate( &stream1 ) );

  // allocate the memory on the GPU
  HANDLE_ERROR( cudaMalloc( (void**)&dev_a0, N * sizeof(int) ) );
  HANDLE_ERROR( cudaMalloc( (void**)&dev_b0, N * sizeof(int) ) );
  HANDLE_ERROR( cudaMalloc( (void**)&dev_c0, N * sizeof(int) ) );
  HANDLE_ERROR( cudaMalloc( (void**)&dev_a1, N * sizeof(int) ) );
  HANDLE_ERROR( cudaMalloc( (void**)&dev_b1, N * sizeof(int) ) );
  HANDLE_ERROR( cudaMalloc( (void**)&dev_c1, N * sizeof(int) ) );
}
N * sizeof(int) );

// allocate host locked memory, used to stream
HANDLE_ERROR( cudaMemcpy( host_a,
    FULL_DATA_SIZE * sizeof(int),
    cudaMemcpyHostToDevice ) );

HANDLE_ERROR( cudaMemcpy( host_b,
    FULL_DATA_SIZE * sizeof(int),
    cudaMemcpyHostToDevice ) );

HANDLE_ERROR( cudaMemcpy( host_c,
    FULL_DATA_SIZE * sizeof(int),
    cudaMemcpyHostToDevice ) );

for (int i=0; i<FULL_DATA_SIZE; i++) {
    host_a[i] = rand();
    host_b[i] = rand();
}

HANDLE_ERROR( cudaMemcpy( dev_a0, host_a+i,
    N * sizeof(int),
    cudaMemcpyHostToDevice, stream0 ) );

HANDLE_ERROR( cudaMemcpy( dev_b0, host_b+i,
    N * sizeof(int),
    cudaMemcpyHostToDevice, stream0 ) );

kernel<<<N/256,256,0,stream0>>>( dev_a0, dev_b0, dev_c0 );

// copy the data from device to locked memory
HANDLE_ERROR( cudaMemcpy( host_c+i, dev_c0,
    N * sizeof(int),
    cudaMemcpyDeviceToHost, stream0 ) );

// copy the locked memory to the device, async
HANDLE_ERROR( cudaMemcpyAsync( dev_a1, host_a+i+N,
    N * sizeof(int),
    cudaMemcpyHostToDevice, stream1 ) );

HANDLE_ERROR( cudaMemcpyAsync( dev_b1, host_b+i+N,
    N * sizeof(int),
    cudaMemcpyHostToDevice, stream1 ) );

kernel<<<N/256,256,0,stream1>>>( dev_a1, dev_b1, dev_c1 );

// copy the data from device to locked memory
HANDLE_ERROR( cudaMemcpy( host_c+i+N, dev_c1,
    N * sizeof(int),
    cudaMemcpyDeviceToHost, stream1 ) );
}
HANDLE_ERROR( cudaStreamSynchronize( stream0 ) );
HANDLE_ERROR( cudaStreamSynchronize( stream1 ) );

HANDLE_ERROR( cudaEventRecord ( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize ( stop ) );
HANDLE_ERROR( cudaEventElapsedTime ( &elapsedTime, start, stop ) );
printf("Time taken: %3.1f ms\n", elapsedTime);

// cleanup the streams and memory
HANDLE_ERROR( cudaFreeHost( host_a ) );
HANDLE_ERROR( cudaFreeHost( host_b ) );
HANDLE_ERROR( cudaFreeHost( host_c ) );
HANDLE_ERROR( cudaFree( dev_a0 ) );
HANDLE_ERROR( cudaFree( dev_b0 ) );
HANDLE_ERROR( cudaFree( dev_c0 ) );
HANDLE_ERROR( cudaFree( dev_a1 ) );
HANDLE_ERROR( cudaFree( dev_b1 ) );
HANDLE_ERROR( cudaFree( dev_c1 ) );
HANDLE_ERROR( cudaStreamDestroy( stream0 ) );
HANDLE_ERROR( cudaStreamDestroy( stream1 ) );

return 0;
}
1. For a very large parallel problem how might we combine these parallel computing paradigms (MPI, pthreads, CUDA)?

Recall the n-body problem from Chapter 6 where we calculate the movement of n-bodies (e.g., n objects/planets in space, or n particles in a contain) over time. To be concrete, the book considers the motion of planets or stars in a 2D space.

Serial code:

Get input data;
for each timestep {
    if (timestep output) Print positions and velocities of particles;
    for each particle q
        Compute total force on q;
    for each particle q
        Compute position and velocity of q;
}
Print positions and velocities of particles:

The for each particle q:
    Computer total force on q”
code needs to perform individual force calculations:

Nodes:
- row 0 are the forces on particle 0 by other particles, etc.
- matrix is “symmetric”, except opposite forces are negated
  (two versions: basic and reduced utilizing symmetry)

2. Consider how we might begin to parallelize the n-body problem using CUDA on the GPU by answering the following questions:

a) Where (i.e., in what type of CUDA memory) should we store the particle masses?

b) Where (i.e., in what type of CUDA memory) should we store the particle positions and velocities?

c) How would you allocate the computation of individual force calculations (e.g., \( f_{01}, f_{02}, \text{etc.} \)) to CUDA threads?

d) How would you compute the total force on an individual particle q?

e) How would you allocate the computation of updating particle positions and velocities to CUDA threads?

f) How would you synchronize the outer-loop to complete a time-step before starting the next?
Week 15 Discussion Questions

I’ve thought some about implementing TSP on GPU. In fact I presented a poster paper at PDPTA’13 (International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV, July 22-25, 2013) (Partially paper below)

Abstract: Depth-first search (DFS) tree searching algorithms are a common implementation approach for many NP-complete optimization algorithms (traveling salesperson problem (TSP), 0-1 Knapsack problem, etc.). In a GPU environment the host computer typically performs a breadth-first expansion of the top of the search tree to determine subtrees that can be assigned to GPU threads, then these subtrees are transferred from the host memory to the GPU-device global memory before the GPU threads can start their search. This paper describes a parallel algorithm that allows the GPU threads to determine their initial subtrees.

Background: Depth-first search (DFS) tree searching algorithms are a common implementation approach for many NP-complete optimization algorithms [2]. Consider the traveling salesperson problem (TSP) for the below graph, where a salesperson starting at her hometown (say v₁) wants to visit every other city exactly once before return to her hometown (called a tour) using a minimum total cost. For this toy example, the minimum tour is [v₁,v₂,v₃,v₄,v₂,v₁] with a total cost of 21.

Parallelization Algorithm

In a GPU environment the host computer typically performs a breadth-first expansion of the top of the search tree to determine subtrees that can be assigned to GPU threads, then these subtrees are transferred from the host memory to the GPU-device global memory before the GPU threads can start their search on the subtrees.

The following parallel algorithm allows the t (a power of 2) GPU threads to determine their initial subtrees. Consider a search tree with a branching factor of n at each level, we can visualize the total work of the search tree using two arrays Start and End as in (a) below. To split the work into two halves, half of level 1's values can be split off as in (b) by thread 0.

```
// binary-tree scatter of work, threadsPerBlock must be a power of 2
i = blockIdx.x;
while (i > 0) {
    if (threadIdx.x % i == 0) {
        for (level = 2; level < n; level++) {
            if (Start[threadIdx.x][level-2] == End[threadIdx.x][level-2]) {
                Start[threadIdx.x][level-2] = Start[threadIdx.x][level-2] + End[threadIdx.x][level-2];
                Start[threadIdx.x][level-2] = Start[threadIdx.x][level-2];
            } else {
                mid = (Start[threadIdx.x][level-2] + End[threadIdx.x][level-2]) / 2;
                Start[threadIdx.x][level-2] = mid;
                End[threadIdx.x][level-2] = mid;
            }
        } // end if
    } // end for
} // end while
```

This binary-tree scattering of work among the threads would look like the following in CUDA pseudo-code:

Week 15 Discussion Questions Page 2
Week 15 Discussion Questions

Some NP-complete problems search for optimal subsets of items from a set of size n (e.g., 0-1 Knapsack problem, subset-sum problem, etc.). In these cases, the above binary-tree scattering of work is not even needed since each of the threads (a power of $2^k$) GPU thread Id's if thought of as k-bit binary numbers represent the starting subtrees level k in the search tree. Consider the small example of $t = 8 = 2^3$ with binary thread Ids: 000, 001, 010, 011, 100, 101, 110, 111.

After a thread is assigned an initial subtree (regardless of the type of search tree), its initial state and feasibility must be evaluated if a promising function is being used to prune branches of the search tree.

2. Answer the following questions about my above paper.

a) Why is it bad to have the host computer perform a breadth-first expansion of the top of the search tree to determine which subtrees can be assigned to GPU threads? (i.e., why is it better if we can parallelize the assignment of subtrees to GPU threads?)

b) My TSP approach for the GPU as present has a serious flaw. What is it?

c) My approach for NP-complete problems that search for optimal subsets of items has the same problem, but it can be "salvaged" by a relatively simple modification. What is the modification?