CUDA Tutorial

COMP5112 Assignment4

WANG, Lipeng

Outline

- CUDA Environment
- CUDA basics
- Assignment 4

CUDA Environment

CUDA Toolkit location on CS lab2 machines:

- /usr/local/cuda-8.0/
  - *bin/*
    the compiler executable and runtime libraries
  - *include/*
    the header files needed to compile CUDA programs
  - *lib64/*
    the library files needed to link CUDA programs
  - *samples/*
    CUDA sample code

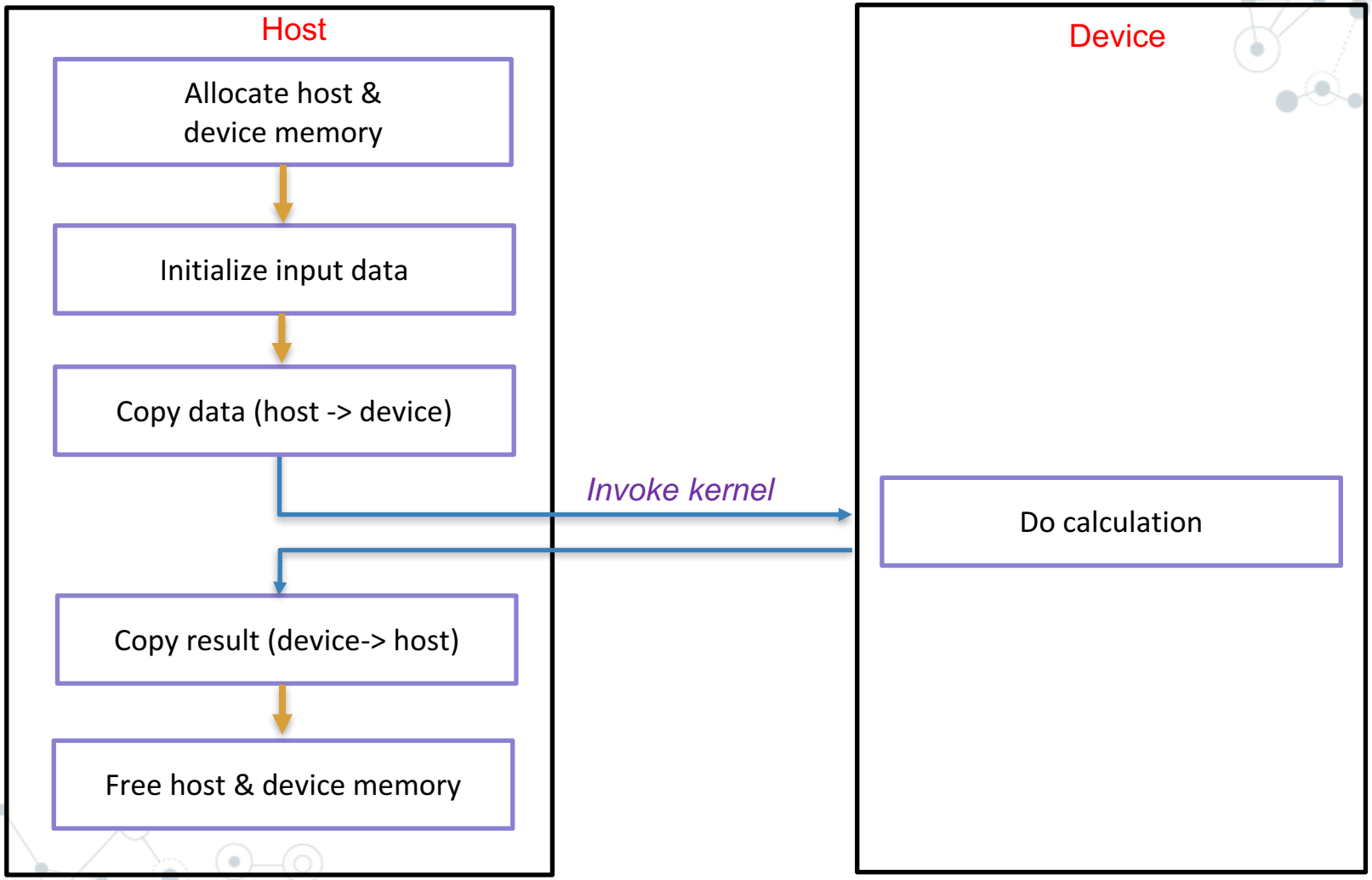
CUDA Environment cont.

Check your CUDA environment first:

- Open your terminal application
- Use *nvcc --version* to check your CUDA environment
- If you cannot found *nvcc* command(or it is not CUDA 8.0), please add the CUDA toolkit installation path to the end of your *~/.cshrc* file
- Close your terminal application and re-open it (or re-login to this machine)
- Run *nvcc --version* to check again

```
[csl2wk01 ~]$nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2016 NVIDIA Corporation
Built on Tue_Jan_10_13:22:03_CST_2017
Cuda compilation tools, release 8.0, V8.0.61
[csl2wk01 ~]$
```

Typical CUDA programming model



Memory Allocation

- Host Memory
  - malloc
    - void* malloc(size_t size);

Parameters:
- *size : Size of the memory block, in bytes.*
  size_t is an unsigned integral type.

Returns:
- *On success, a pointer to the memory block allocated by the function.*

```
int *h_A, *d_A;
size_t size = 1024* sizeof(int);

//on host memory
h_A = (int*) malloc(size);
```

- Device Memory
  - cudaMalloc
    - cudaMalloc(void **ptr, size_t size);

Parameters:
- *devPtr* : Pointer to allocated device memory
- *size* : Requested allocation size in bytes

Returns:
- cudaSuccess, cudaErrorMemoryAllocation

```
//on device memory
cudaMalloc(&d_A, size);
```

Memory deallocation

- Host Memory
  - free
    - void* free(void* ptr);

Parameters:
- *ptr:* This is the pointer to a memory block previously allocated with malloc, calloc or realloc to be deallocated. If a null pointer is passed as argument, no action occurs.

Returns:
- This function does not return any value.

- Device Memory
  - cudaFree
    - cudaFree(void* devPtr);

Parameters:
- *devPtr* : Device pointer to memory to free

Returns:
- cudaSuccess, cudaErrorInvalidDevicePointer, cudaErrorInitializationError

```
int *h_A, *d_A;
size_t size = 1024* sizeof(int);

//allocate memory
h_A = (int*) malloc(size);
cudaMalloc(&d_A, size);

//free memory on host
free(h_A);

//free memoty on device
cudaFree(d_A);
```

Data transfer between host and device

```
cudaMemcpy(void*                    dst,
           const void*              src
           size_t                   count,
           enum cudaMemcpyKind      kind
          )
```

Copies `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `dst`, where kind is one of *cudaMemcpyHostToHost*, *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost*, or *cudaMemcpyDeviceToDevice*, and specifies the direction of the copy. The memory areas may not overlap. Calling *cudaMemcpy()* with `dst` and `src` pointers that do not match the direction of the copy results in an undefined behavior.

Parameters:

dst	-	Destination memory address
src	-	Source memory address
count	-	Size in bytes to copy
kind	-	Type of transfer

```
//host -> device
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice)
//device -> host
cudaMemcpy(h_A, d_A, size, cudaMemcpyDeviceToHost)
```

CUDA kernel invocation

- A kernel function has the prefix __global__ , return type void
  - `__global__ void kernelName (param1, …)`
- `kernelName<<<#block, #thread, shared_size, s>>>(par1,…)`
- Most cases
  - `kernelName<<<#block, #thread>>>(par1,…)`
    - #block: number of blocks in a grid
    - #thread: number of threads per block
- E.g.:
  - `addKernel<<<1, size>>>(d_c, d_a, d_b);`

Built-in variable *dim3*

- dim3 is an integer vector type that can be used in CUDA code. Its most common application is to pass the grid and block dimensions in a kernel invocation. It can also be used in any user code for holding values of 3 dimensions.

- dim3 is a simple structure that is defined in %CUDA_INC_PATH%/vector_types.h

- dim3 has 3 elements: x, y, z
  - C code initialization: `dim3 grid = {512, 512, 1};`
  - C++ code initialization: `dim3 grid(512,512,1);`

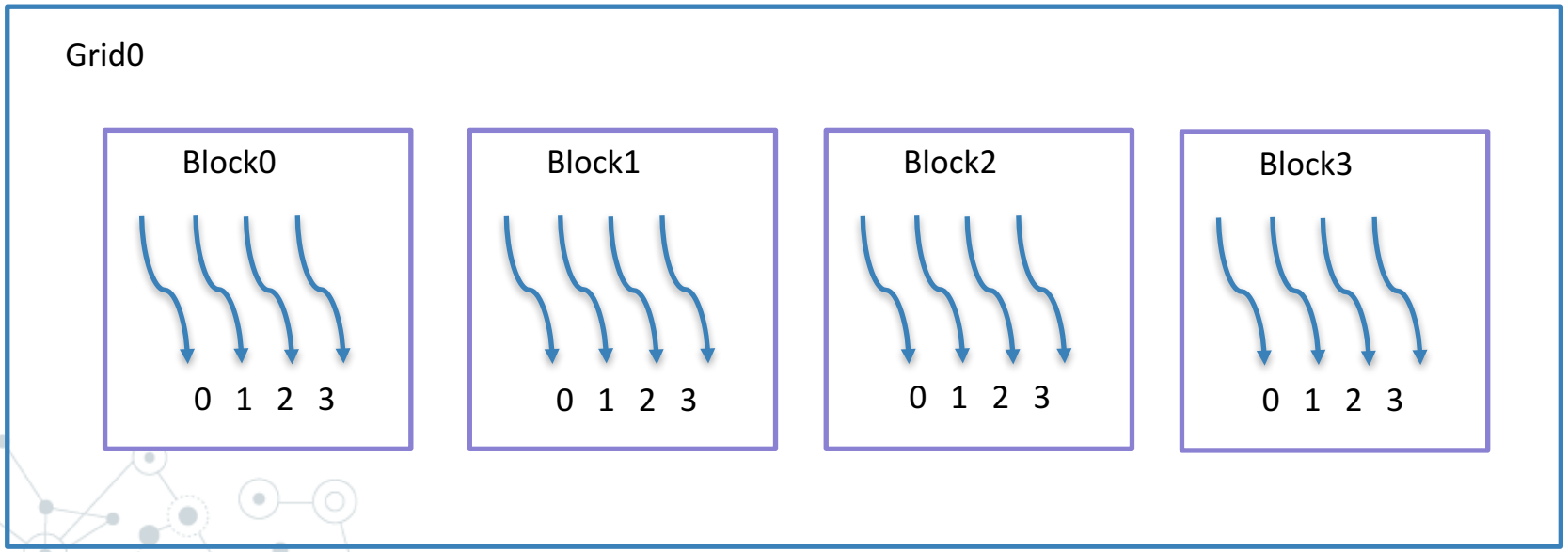
Built-in variable *dim3* (cont..)

- Not all three elements need to be provided
  - Any element not provided during initialization is initialized to 1, not 0!

- Examples
  - `dim3 block(512); // 512 * 1 * 1`
  - `dim3 thread(512, 2) // 512 * 2 * 1`
  - `fooKernel<<< block, thread>>> ();`

Dim3 example

```
// 1 grid -> 4 blocks -> 4 threads/block
dim3 block(4,1,1);
dim3 thread(4,1,1);
addKernel<<<block, thread>>>(d_c, d_a, d_b);
```



Thread index calculation

- Built-in variables which can be used in device code

- grid
  - `gridDim.x`
  - `gridDim.y`
  - `gridDim.z`

- block
  - `blockDim.x`
  - `blockDim.y`
  - `blockDim.z`

Thread index calculation (cont.)

- 1D grid of 1D blocks

```
//1D * 1D
threadID = blockDim.x * blockIdx.x + threadIdx.x;
```

- 1D grid of 2D blocks

```
//1D * 2D
threadID = blockDim.x * blockDim.y * blockIdx.x +
           blockDim.x * threadIdx.y +
           threadIdx.x;
```

- 1D grid of 3D blocks

```
//1D * 3D
threadID = blockDim.x * blockDim.y * blockDim.z * blockIdx.x +
           (blockDim.x * blockDim.y) * threadIdx.z +
           blockDim.x * threadIdx.y +
           threadIdx.x;
```

Thread index calculation (cont.)

- 2D grid of 1D blocks

```
//2D * 1D
blockID = gridDim.x * blockIdx.y + blockIdx.x;
threadID = blockID * blockDim.x + threadIdx.x
```

- 2D grid of 2D blocks

```
//2D * 2D
blockID = gridDim.x * blockIdx.y + blockIdx.x;
threadID = blockID * (blockDim.x * blockDim.y) +
           blockDim.x * threadIdx.y +
           threadIdx.x;
```

- 2D grid of 3D blocks

```
//2D * 3D
blockID = gridDim.x * blockIdx.y + blockIdx.x;
threadID = blockID * (blockDim.x * blockDim.y * blockDim.z) +
           (blockDim.x * blockDim.y) * threadIdx.z +
           blockDim.x * threadIdx.y +
           threadIdx.x;
```

Thread index calculation (cont.)

- 3D grid of 1D blocks

```
//3D * 1D
blockID = gridDim.x * gridDim.y * blockIdx.z +
          gridDim.x * blockIdx.y +
          blockIdx.x;
```

- 3D grid of 2D blocks

```
//3D * 2D
blockID = gridDim.x * gridDim.y * blockIdx.z +
          gridDim.x * blockIdx.y +
          blockIdx.x;
threadID = blockID * (blockDim.x * blockDim.y) +
           blockDim.x * threadIdx.y +
           threadIdx.x;
```

- 3D grid of 3D blocks

```
//3D * 3D
blockID = gridDim.x * gridDim.y * blockIdx.z +
          gridDim.x * blockIdx.y +
          blockIdx.x;
threadID = blockID * (gridDim.x * gridDim.y * gridDim.z) +
           (gridDim.x * gridDim.y) * threadIdx.z +
           gridDim.x * threadIdx.y +
           threadIdx.x;
```

Take Parallel-Reduction as an example

- Given an integer array, calculate the summation of this array

- Need to be able to use multiple thread blocks
  - To process very large arrays
  - To keep all multiprocessors on the GPU busy
  - Each thread block reduces a portion of the array to a single value

Take Parallel-Reduction as an example

- If we could synchronize across all thread blocks, could easily reduce very large arrays
  - Global Sync after each block produces its result
  - Once all blocks reach sync, reduce the results of all blocks
- But CUDA has no global synchronization between blocks
  - Only synchronization between threads in a single block
- Solution: decompose into multiple kernels

Take Parallel-Reduction as an example

Kernel configuration

- Restrict blocks/grid to 8
- Restrict threads/block to 1024

```
//kernel configuration
dim3 blocks(8);
dim3 threads(1024);
```

Take Parallel-Reduction as an example

- Two kernels

```
reduction1 <<<blocks, threads>>> (d_input_data, d_block_local_sum, N);
reduction2 <<<blocks, threads>>> (d_block_local_sum, d_sum);
```

Take Parallel-Reduction as an example

```
__global__ void reduction1(int *d_input_data, int *block_local_sum, int N){
    int element_per_thread = (int) ceil(N * 1.0 / (blockDim.x * gridDim.x));

    //store the sum for each thread
    __shared__ int thread_local_sum[1024];

    int my_start = (blockIdx.x * blockDim.x + threadIdx.x) * element_per_thread;
    int my_end = my_start + element_per_thread;
    if(my_end >= N){
        my_end = N;
    }

    int my_sum = 0;
    for(int i = my_start ; i < my_end; i++){
        my_sum += d_input_data[i];
    }
    //store my result to shared memory
    thread_local_sum[threadIdx.x] = my_sum;
    __syncthreads();//synchronization, make sure every threads has done their summation

    //use the first thread in this block to calculate the local summation of this block
    if(threadIdx.x == 0){
        int this_block_sum = 0;
        for (int i = 0 ; i < blockDim.x; i++){
            this_block_sum += thread_local_sum[i];
        }

        //store this block summation to global memory address
        block_local_sum[blockIdx.x] = this_block_sum;
    }
}

__global__ void reduction2(int *block_local_sum, int *d_sum){
    if(threadIdx.x == 0 && blockIdx.x == 0){
        int all_sum = 0;
        //sum all block's result up to get the global summation
        for(int i = 0; i < gridDim.x; i++) {
            all_sum += block_local_sum[i];
        }

        d_sum[0] = all_sum;
    }
}
```

Assignment 4 – Your tasks

- You need to implement 3 kernel functions

- Kernel configuration
  - Restrict blocks/grid to 8
  - threads/block can be changed ($32 \leq p \leq 1024$ & p is power of 2)

```
//we restrict this value to 8, DO NOT change it!
int blocksPerGrid = 8;

//NOTICE: (p * 8) may LESS THAN N
int threadsPerBlock = p;

dim3 blocks(blocksPerGrid);
dim3 threads(threadsPerBlock);
```

Assignment 4 – Helper functions

Error checking

```
40  /*
41   * This is a CHECK function to check CUDA calls
42   */
43  #define CHECK(call)
44   {
45       const cudaError_t error = call;
46       if (error != cudaSuccess)
47       {
48           fprintf(stderr, "Error: %s:%d, ", __FILE__, __LINE__);
49           fprintf(stderr, "code: %d, reason: %s\n", error,
50                   cudaGetErrorString(error));
51           exit(1);
52       }
53   }
54
```

Assignment 4 – Main function

```
264 int main(int argc, char **argv) {
265     assert(utils::parse_args(argc, argv) == 0);
266     assert(utils::read_file(utils::filename) == 0);
267 
268     //`all_dist` stores the distances and `all_pred` stores the predecessors
269     int *all_dist;
270     int *all_pred;
271     all_dist = (int *) calloc(utils::N, sizeof(int));
272     all_pred = (int *) calloc(utils::N, sizeof(int));
273 
274     //time counter
275     timeval start_wall_time_t, end_wall_time_t;
276     float ms_wall;
277 
278     cudaDeviceReset();
279 
280     //start timer
281     gettimeofday(&start_wall_time_t, nullptr);
282     dijkstra(utils::N, utils::num_threads, utils::mat, all_dist, all_pred);
283     CHECK(cudaDeviceSynchronize());
284 
285     //end timer
286     gettimeofday(&end_wall_time_t, nullptr);
287     ms_wall = ((end_wall_time_t.tv_sec - start_wall_time_t.tv_sec) * 1000 * 1000
288                + end_wall_time_t.tv_usec - start_wall_time_t.tv_usec) / 1000.0;
289 
290     std::cerr << "Time(ms): " << ms_wall << endl;
291 
292     utils::print_result(all_dist, all_pred);
293 
294     free(utils::mat);
295     free(all_dist);
296     free(all_pred);
297 
298     return 0;
299 }
```

Assignment 4 – Dijkstra function

```
191 //Do not change anything below this line
192 void dijkstra(int N, int p, int *mat, int *all_dist, int *all_pred) {
193
194     //threads number for each block should smaller than or equal to 1024
195     assert(p <= 1024);
196
197     //we restrict this value to 8, DO NOT change it!
198     int blocksPerGrid = 8;
199
200     //NOTICE: (p * 8) may LESS THAN N
201     int threadsPerBlock = p;
202
203     dim3 blocks(blocksPerGrid);
204     dim3 threads(threadsPerBlock);
205
206
207     //allocate memory
208     int *h_visit;
209     int *d_mat, *d_visit, *d_all_dist, *d_all_pred, *d_local_min, *d_local_min_index;
210     int *d_global_min, *d_global_min_index;
211
212     h_visit = (int *) calloc(N, sizeof(int));
213     cudaMalloc(&d_mat, sizeof(int) * N * N);
214     cudaMalloc(&d_visit, sizeof(int) * N);
215     cudaMalloc(&d_all_dist, sizeof(int) * N);
216     cudaMalloc(&d_all_pred, sizeof(int) * N);
217     cudaMalloc(&d_local_min, sizeof(int) * blocksPerGrid);
218     cudaMalloc(&d_local_min_index, sizeof(int) * blocksPerGrid);
219     cudaMalloc(&d_global_min, sizeof(int));
220     cudaMalloc(&d_global_min_index, sizeof(int));
221
222     //initialization and copy data from host to device
223     for (int i = 0; i < N; i++) {
224         all_dist[i] = mat[i];
225         all_pred[i] = 0;
226         h_visit[i] = 0;
227     }
228     h_visit[0] = 1;
```

Assignment 4 – Dijkstra function (cont.)

```
229
230     cudaMemcpy(d_mat, mat, sizeof(int) * N * N, cudaMemcpyHostToDevice);
231     cudaMemcpy(d_all_dist, all_dist, sizeof(int) * N, cudaMemcpyHostToDevice);
232     cudaMemcpy(d_all_pred, all_pred, sizeof(int) * N, cudaMemcpyHostToDevice);
233     cudaMemcpy(d_visit, h_visit, sizeof(int) * N, cudaMemcpyHostToDevice);
234
235     //dijkstra iterations
236     for (int iter = 1; iter < N; iter++) {
237         FindLocalMin <<< blocks, threads >>> (N, d_visit, d_all_dist, d_local_min, d_local_min_index);
238         //CHECK(cudaDeviceSynchronize()); //only for debug
239         UpdateGlobalMin <<< blocks, threads >>>
240                                     (d_global_min, d_global_min_index, d_local_min, d_local_min_index, d_visit);
241         //CHECK(cudaDeviceSynchronize()); //only for debug
242         UpdatePath << < blocks, threads >> >
243                                 (N, d_mat, d_visit, d_all_dist, d_all_pred, d_global_min, d_global_min_index);
244         //CHECK(cudaDeviceSynchronize()); //only for debug
245     }
246
247     //copy results from device to host
248     cudaMemcpy(all_dist, d_all_dist, sizeof(int) * N, cudaMemcpyDeviceToHost);
249     cudaMemcpy(all_pred, d_all_pred, sizeof(int) * N, cudaMemcpyDeviceToHost);
250
251     //free memory
252     free(h_visit);
253     cudaFree(d_mat);
254     cudaFree(d_visit);
255     cudaFree(d_all_dist);
256     cudaFree(d_all_pred);
257     cudaFree(d_local_min);
258     cudaFree(d_local_min_index);
259     cudaFree(d_global_min);
260     cudaFree(d_global_min_index);
261
262 }
```

Assignment 4 – Dijkstra iterations

```
//dijkstra iterations
for (int iter = 1; iter < N; iter++) {
    FindLocalMin <<< blocks, threads >>> (N, d_visit, d_all_dist, d_local_min, d_local_min_index);
    //CHECK(cudaDeviceSynchronize()); //only for debug
    UpdateGlobalMin <<< blocks, threads >>>
                               (d_global_min, d_global_min_index, d_local_min, d_local_min_index, d_visit);
    //CHECK(cudaDeviceSynchronize()); //only for debug
    UpdatePath << < blocks, threads >> >
                           (N, d_mat, d_visit, d_all_dist, d_all_pred, d_global_min, d_global_min_index);
    //CHECK(cudaDeviceSynchronize()); //only for debug
}
```

Assignment 4 – CUDA Kernels

```cuda
/*
 * function: find the local minimum for each block and store them to d_local_min and d_local_min_index
 * parameters: N: input size, *d_visit: array to record which vertex has been visited, *d_all_dist: array to store the distance,
 *        *d_local_min: array to store the local minimum value for each block, *d_local_min_index: array to store the local minimum index for each block
 */
__global__ void FindLocalMin(int N, int *d_visit, int *d_all_dist, int *d_local_min, int *d_local_min_index) {

}

/*
 * function: update the global minimum value(and index), store them to a global memory address
 * parameters: *global_min: memory address to store the global min value, *global_min_index: memory address to store the global min index
 *        *d_local_min: array stores the local min value od each block, *d_local_min_index: array stores the local min index of each block
 *        *d_visit: array stores the status(visited/un-visited) for each vertex
 */
__global__ void
UpdateGlobalMin(int *global_min, int *global_min_index, int *d_local_min, int *d_local_min_index, int *d_visit) {

}

/*
 * function: update the shortest path for every un-visited vertices
 * parameters: N: input size, *mat: input matrix, *d_visit: array stores the status(visited/un-visited) for each vertex
 *             *d_all_dist: array stores the shortest distance for each vertex, *d_all_pred: array stores the predecessors
 *             *global_min: memory address that stores the global min value, *global_min_index: memory address that stores the global min index
 */
__global__ void
UpdatePath(int N, int *mat, int *d_visit, int *d_all_dist, int *d_all_pred, int *global_min, int *global_min_index) {

}
```

Thanks!
Q&A