CUDA Tutorial

COMP5112 Assignment3

Outline

- CUDA Environment
- CUDA basics
- Assignment 3

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

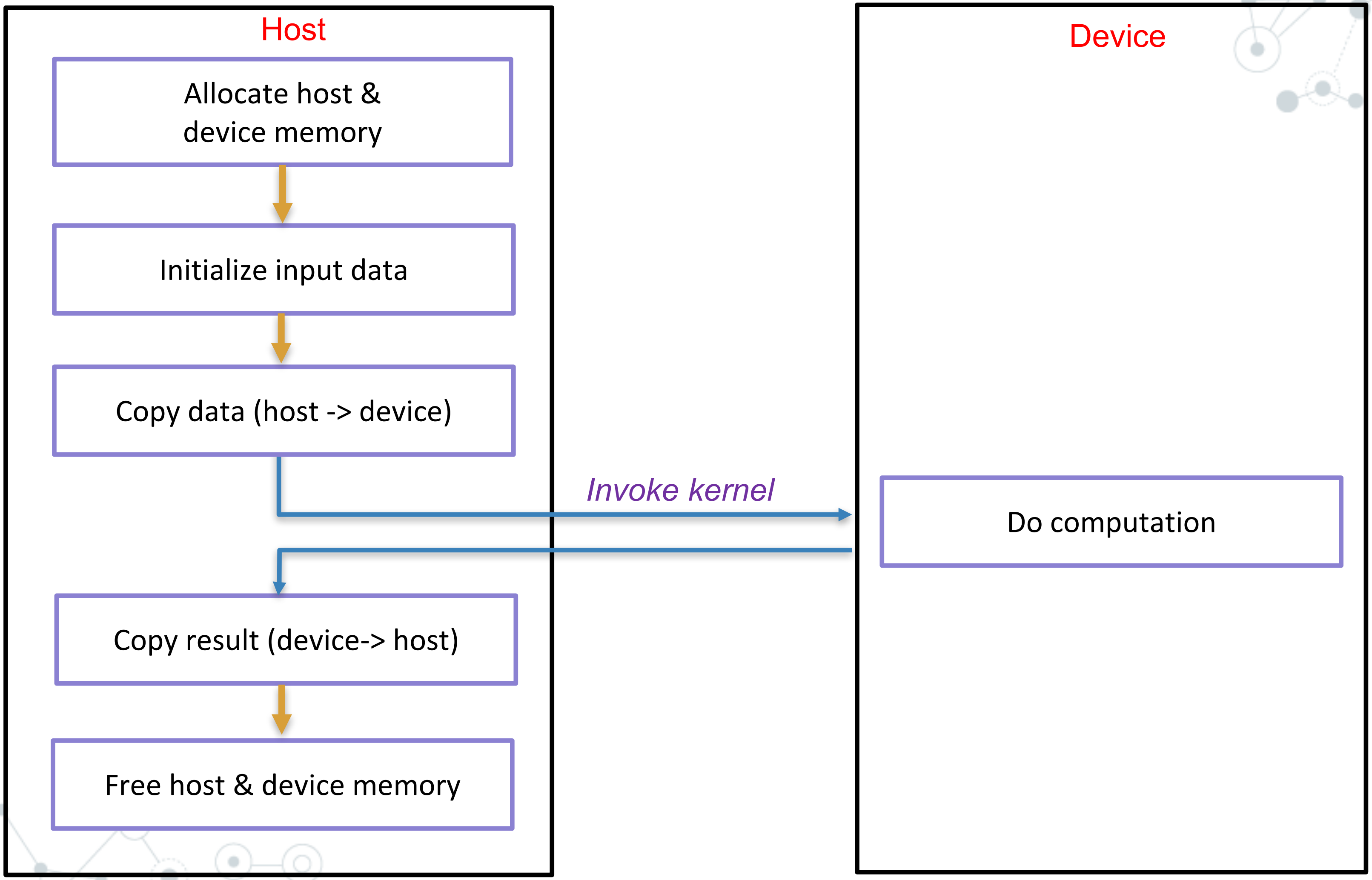
CUDA Environment cont.

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.*

- 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

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

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

//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`

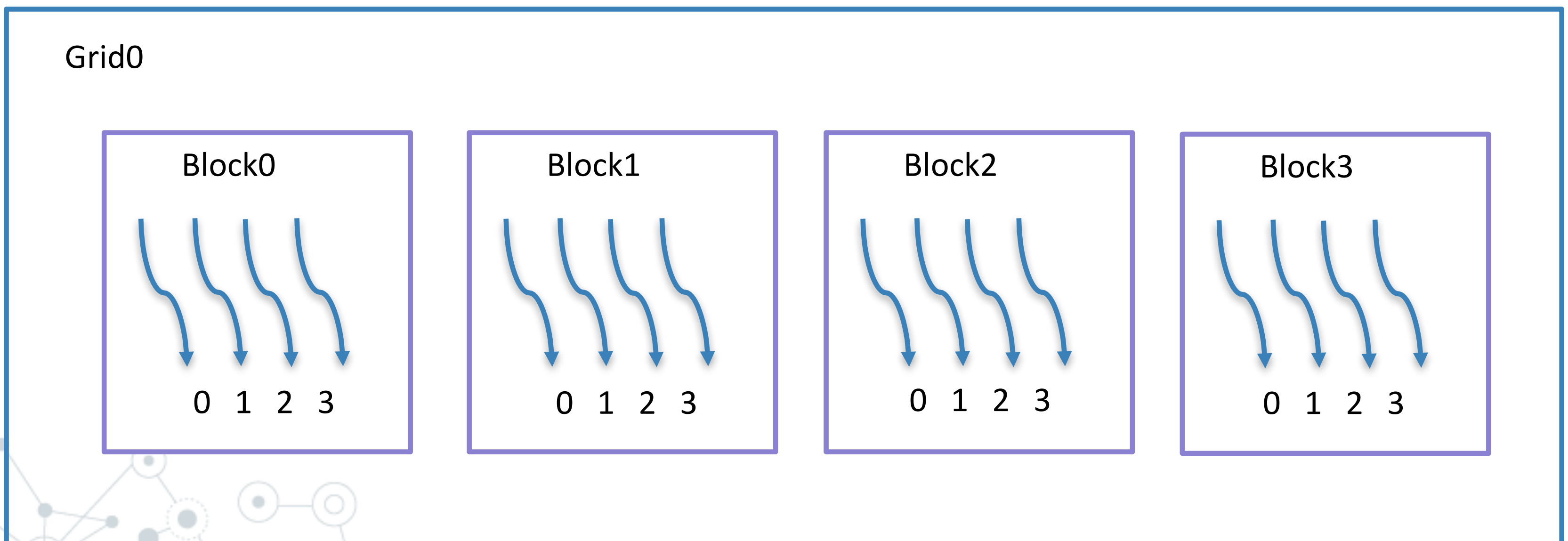
- dim3 has 3 elements: x, y, z
  - C code initialization: `dim3 grid = {512, 512, 1};`
  - C++ code initialization: `dim3 grid(512,512,1);`

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;
```

Assignment 3 - Hints

- Finish *bellman-ford* function
- Write your own kernel function(s)
- Set CUDA kernel configurations correctly
- Use coalesced memory access
- Read last year's solution code first if you have no idea to start

Assignment 3 – Helper function

Error checking

```
/*
 * This is a CHECK function to check CUDA calls
 */
#define CHECK(call)
 {
     const cudaError_t error = call;
     if (error != cudaSuccess)
     {
         fprintf(stderr, "Error: %s:%d, ", __FILE__, __LINE__);
         fprintf(stderr, "code: %d, reason: %s\n", error,
                 cudaGetErrorString(error));
         exit(1);
     }
 }

```

Assignment 3 – Main function

```
int main(int argc, char **argv) {
    if (argc <= 1) {
        utils::abort_with_error_message("INPUT FILE WAS NOT FOUND!");
    }
    if (argc <= 3) {
        utils::abort_with_error_message("blocksPerGrid or threadsPerBlock WAS NOT FOUND!");
    }
    string filename = argv[1];
    int blockPerGrid = atoi(argv[2]);
    int threadsPerBlock = atoi(argv[3]);

    int *dist;
    bool has_negative_cycle = false;
    assert(utils::read_file(filename) == 0);
    dist = (int *) calloc(sizeof(int), utils::N);
    //time counter
    timeval start_wall_time_t, end_wall_time_t;
    float ms_wall;
    cudaDeviceReset();
    //start timer
    gettimeofday(&start_wall_time_t, nullptr);
    //bellman-ford algorithm
    bellman_ford(blockPerGrid, threadsPerBlock, utils::N, utils::mat, dist, &has_negative_cycle);
    CHECK(cudaDeviceSynchronize());
    //end timer
    gettimeofday(&end_wall_time_t, nullptr);
    ms_wall = ((end_wall_time_t.tv_sec - start_wall_time_t.tv_sec) * 1000 * 1000
            + end_wall_time_t.tv_usec - start_wall_time_t.tv_usec) / 1000.0;
    std::cerr.setf(std::ios::fixed);
    std::cerr << std::setprecision(6) << "Time(s): " << (ms_wall/1000.0) << endl;
    utils::print_result(has_negative_cycle, dist);
    free(dist);
    free(utils::mat);
    return 0;
}
```

Thanks!

Q&A