CUDA Tutorial

COMP5112 Assignment3

Outline

- CUDA Environment
- CUDA programming basics
- Assignment 3

CUDA Environment on CS Lab2

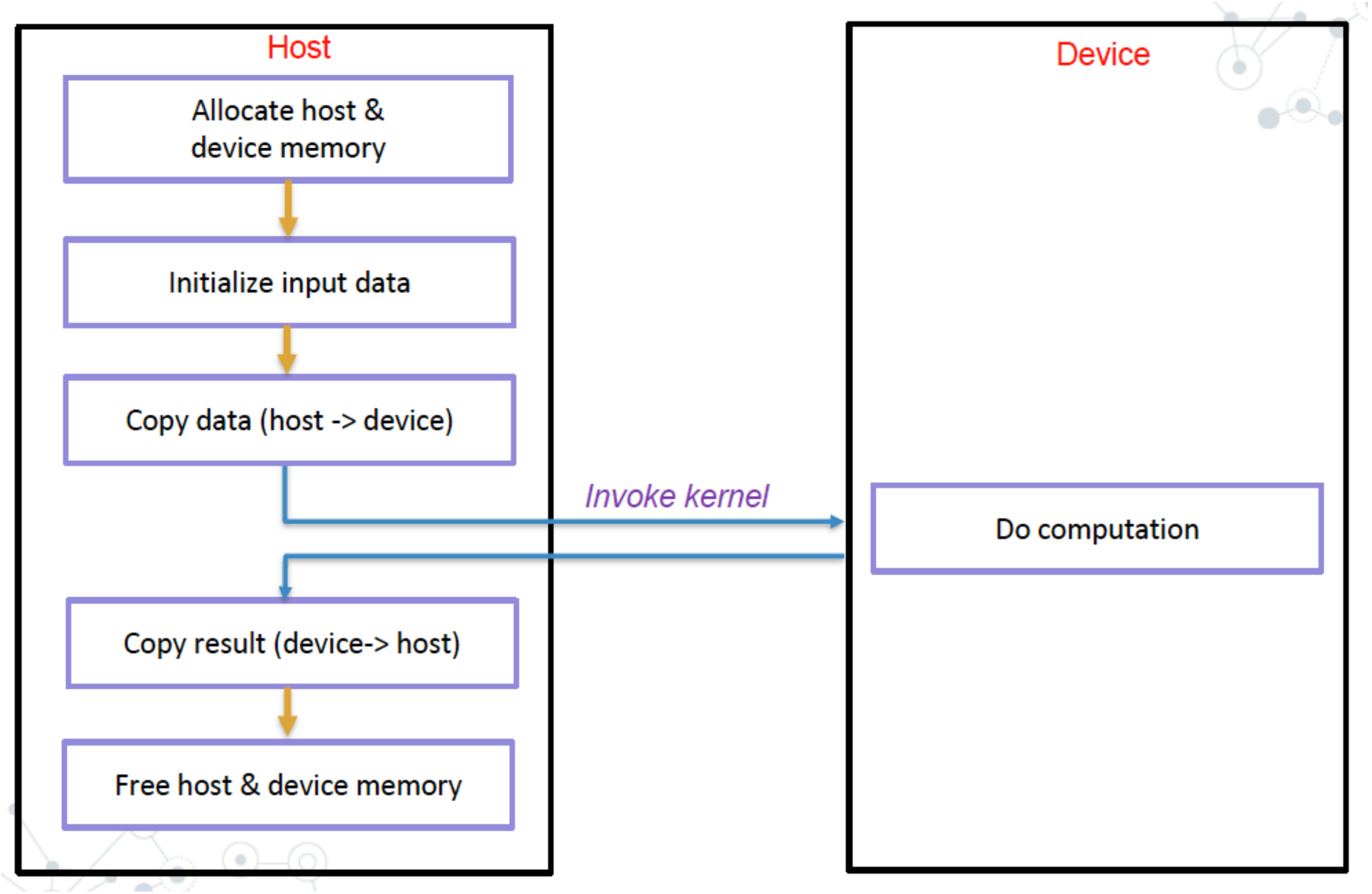
- CUDA version: 8.0
  - path:/usr/local/cuda-8.0/
- Check your CUDA environment first in the terminal:
  - Use `nvcc --version`
- If you cannot found `nvcc` command(or it is not CUDA 8.0), please add the CUDA toolkit installation path to the end of your `~/.cshrc_user` file

```
csl2wk01:ywanghz:156> 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:ywanghz:157>
```

```
###################################################
#       File : .cshrc_user
#       Generic Version 1.1. CS. HKUST
###################################################

setenv MANPATH /usr/local/share/man:${MANPATH}
setenv PATH "${PATH}:/usr/local/software/openmpi/bin"
setenv PATH "${PATH}:/usr/local/cuda-8.0/bin/"
#export CUDA_DEBUGGER_SOFTWARE_PREEMPTION=1
set cuda software_preemption on
~
```

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:`
    - `**ptr: Pointer to allocated device memory.`
    - `size: Requested allocation size in bytes.`
  - `returns:`
    - `cudaSuccess, cudaErrorMemoryAllocation`

Example

```
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`
    - `cudaMalloc(void **ptr);`
  - `Parameters:`
    - `**ptr:Device pointer to memory to free.`
  - `returns:`
    - `cudaSuccess, cudaErrorInvalidDevicePointer, cudaErrorInitializationError.`

Example

```
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)
kind:
 cudaMemcpyHostToHost,
 cudaMemcpyHostToDevice,
 cudaMemcpyDeviceToHost,
 cudaMemcpyDeviceToDevice
```

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

CUDA Kernel Declaration and Invocation

- A kernel function declaration has the prefix `__global__`, return type `void`.

```
__global__ void kernelName(param1, ..);
```

- A kernel function invocation includes launch parameters: #block, #thread..

```
kernelName<<<#block, #thread, size, stream>>>(param1, ..);
```

```
        #block: number of blocks per grid.
        #thread: number of threads per block.
        size and stream can be ignored in our assignment.
```

- E.g:

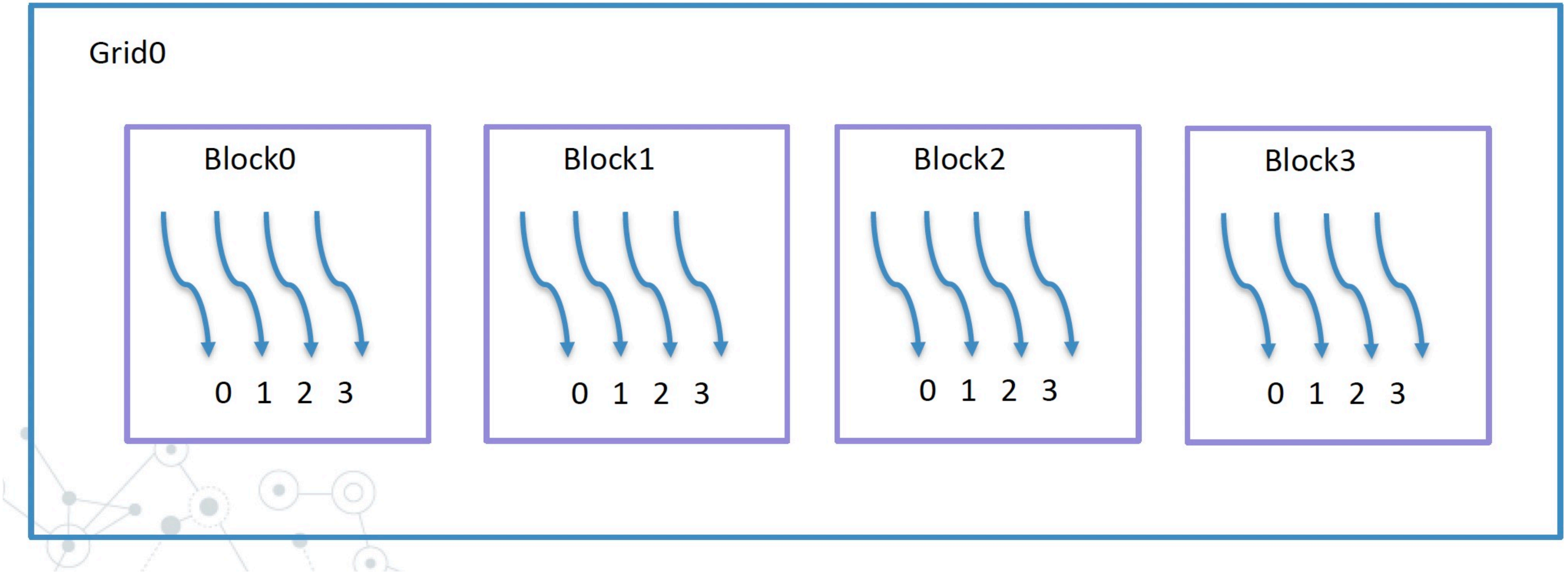
```
AddKernel<<<32, 1024>>>(d_c, d_a, d_b);
```

Build-in variable **`dim3`**

- `dim3` is an integer vector type that can be used in CUDA code.
- Its most common application:
  - pass the grid and block dimensions in a kernel invocation.
- `dim3` has 3 elements: x, y, z
  - C code initialization: `dim3 grid = {512, 512, 1}`
  - C++ code initialization: `dim3 gird(512, 512, 1);`
  - Not all three elements need to be provided.
    - Any element not provided during initialization is initialized to 1, not 0!
- E.g:

```
dim3 block(32); // 32 * 1 * 1
dim3 thread(1024) // 1024 * 1 * 1
AddKernel<<< block, thread>>>(d_c, d_a, d_b);
```

Dim3 example




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

Thread index calculation

- 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

- 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;
```

- 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;
```

Assignment3:Compiling on CS lab2

```
Compile: nvcc -std=c++11 -arch=compute_52 -code=sm_52
             main.cu
             cuda_smith_waterman_skeleton.cu
             -o cuda_smith_waterman
```

```
Run: ./cuda_smith_waterman <input file> <num of blocks per grid> <number of thread
per block>
```

```
Or just use run_cuda.sh bash script.
```

Assignment3: 2D score -> 1D score

Linear representation of a 2D array is convenient to achieve **Coalesced Memory Access.**

In assignment3, the score matrix is represented as 1D array.

To be fair, the serial code in assignment3 reacts to the changes.

Index transform function.

```
#pragma once

using namespace std;

const int MATCH = 3, MIS = -3, GAP = 2;

int smith_waterman(char *a, char *b, int a_len, int b_len);

// return score of substitution matrix
inline int sub_mat(char x, char y) {
    return x == y ? MATCH : MIS;
}

inline int idx(int x, int y, int n){
    return x * n + y;
}
```

score matrix allocation changes.

```
int smith_waterman(char *a, char *b, int a_len, int b_len) {
    // init score matrix
//    int **score = new int*[a_len + 1];
//    for (int i = 0; i <= a_len; i++) {
//        score[i] = new int[b_len + 1];
//        for (int j = 0; j <= b_len; j++) {
//            score[i][j] = 0;
//        }
//    }

    int *score = (int *)malloc(sizeof(int) * (a_len + 1) * (b_len + 1));
    for(int i = 0; i <= a_len; i++){
        for(int j = 0; j <= b_len; j++){
            score[idx(i, j, b_len + 1)] = 0;
        }
    }
```

score matrix addressing changes.

```
// main loop
int max_score = 0;

for (int i = 1; i <= a_len; i++) {
      for (int j = 1; j <= b_len; j++) {
          score[i][j] = max(0,
                            max(score[i - 1][j - 1] + sub_mat(a[i - 1], b[j - 1]),
                            max(score[i - 1][j] - GAP,
                                score[i][j - 1] - GAP)));
          max_score = max(max_score, score[i][j]);

    for (int j = 1; j <= b_len; j++) {
        score[idx(i, j, b_len + 1)] = max(0,
                                 max(score[idx(i - 1, j - 1, b_len + 1)] + sub_mat(a[i - 1], b[j - 1]),
                                         max(score[idx(i - 1, j, b_len + 1)] - GAP,
                                         score[idx(i, j - 1, b_len + 1)] - GAP)));
        max_score = max(max_score, score[idx(i, j, b_len + 1)]);
    }
}
```

Assignment3: 2D score -> 1D score

A device version of index transform is provided in the cuda_smith_waterman.h

```
#pragma once

const int MATCH = 3, MIS = -3, GAP = 2;


int smith_waterman(int blocks_per_grid, int threads_per_block, char *a, char *b, int a_len, int b_len);

inline __device__ int sub_mat(char x, char y) {
    return x == y ? MATCH : MIS;
}

#define GPUErrChk(ans) { utils::GPUAssert((ans), __FILE__, __LINE__); }

namespace utils {

    inline void GPUAssert(cudaError_t code, const char *file, int line, bool abort = true) {
        if (code != cudaSuccess) {
            fprintf(stderr, "GPU assert: %s %s %d\n", cudaGetErrorString(code), file, line);
            if (abort)
                exit(code);
        }
    }

    inline __device__ int dev_idx(int x, int y, int n) {
        return x * n + y;
    }
}
```

Assignment3: Error check helper function

```
#pragma once

const int MATCH = 3, MIS = -3, GAP = 2;


int smith_waterman(int blocks_per_grid, int threads_per_block, char *a, char *b, int a_len, int b_len);

inline __device__ int sub_mat(char x, char y) {
    return x == y ? MATCH : MIS;
}

#define GPUErrChk(ans) { utils::GPUAssert((ans), __FILE__, __LINE__); }

namespace utils {

    inline void GPUAssert(cudaError_t code, const char *file, int line, bool abort = true) {
        if (code != cudaSuccess) {
            fprintf(stderr, "GPU assert: %s %s %d\n", cudaGetErrorString(code), file, line);
            if (abort)
                exit(code);
        }
    }

    inline __device__ int dev_idx(int x, int y, int n) {
        return x * n + y;
    }
}
```

```
E.g:
    GPUErrChk(cudaMalloc(&d_A, size));
```

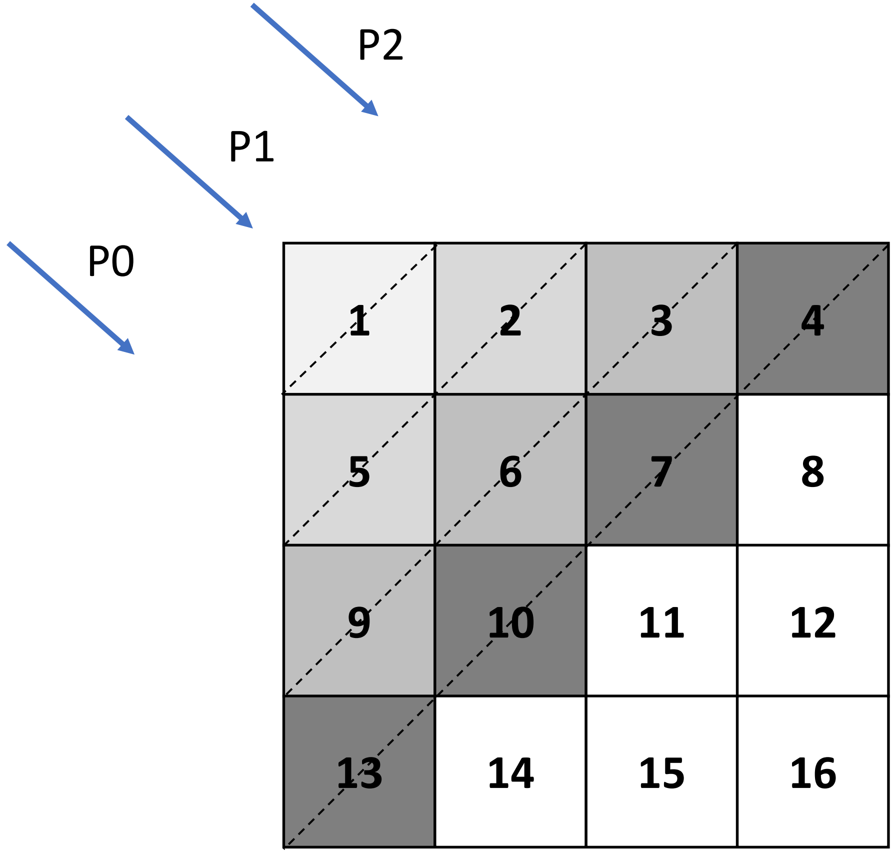
Assignment3: Your task

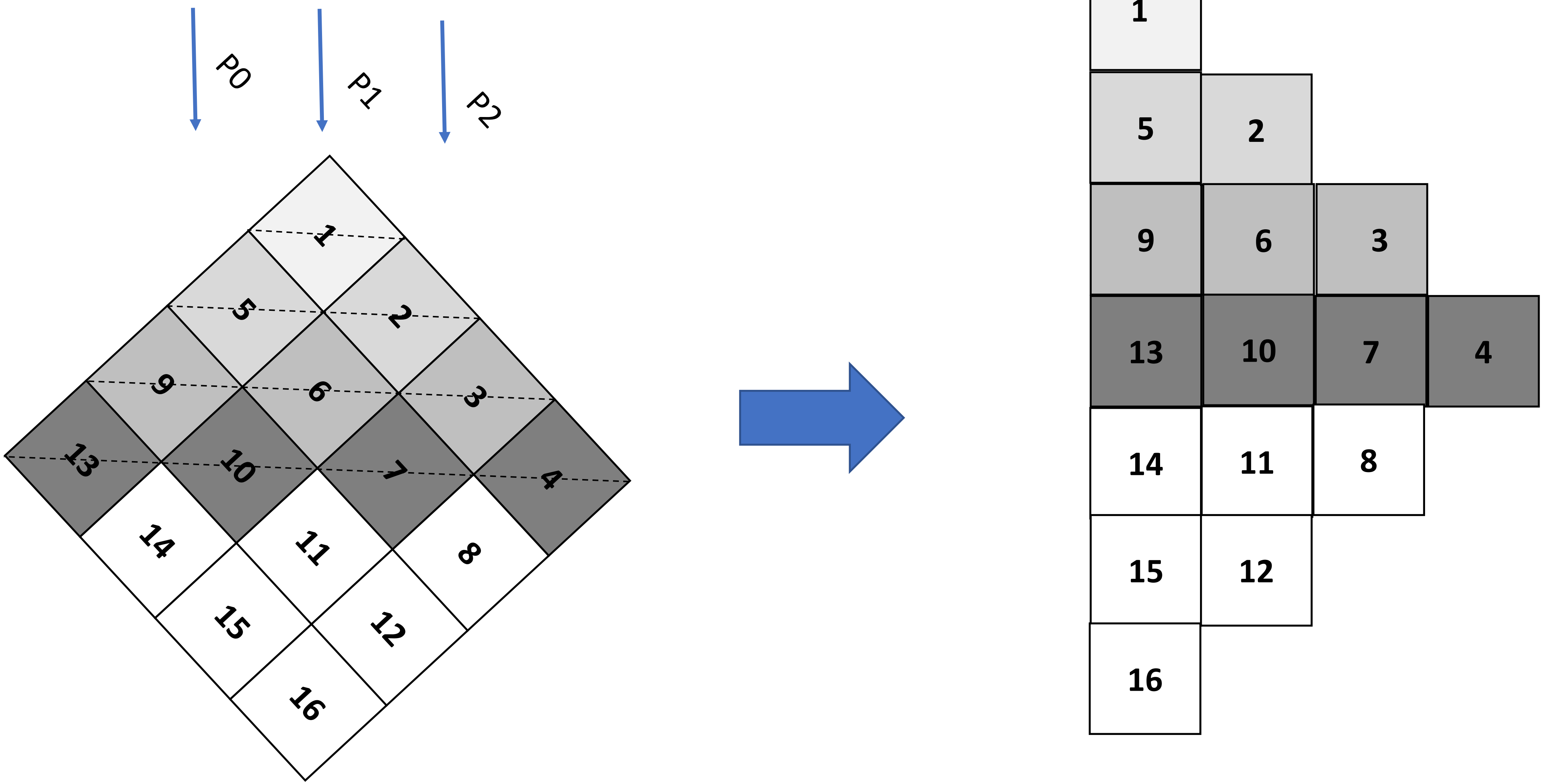
- Handle memory allocation & deallocation by yourself.
- Handle memory copy by yourself.
- Write one or more kernels to do the computation of score matrix.

- Note: Using the global memory is enough.

Assignment3: Hints

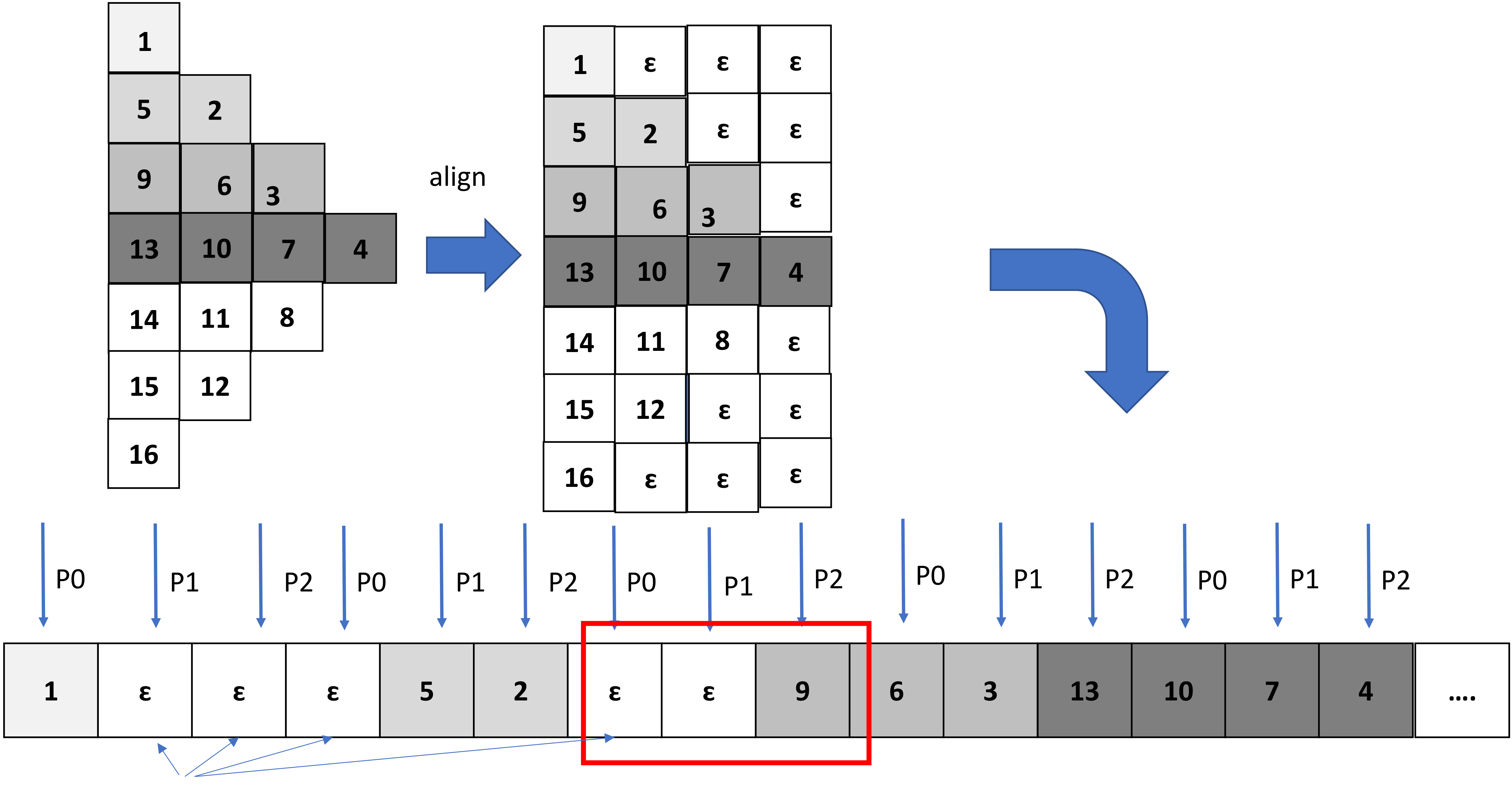
- Coalesced memory access.
  - Improve performance greatly.
  - If the threads in a block are accessing consecutive global memory locations, then all the accesses are combined into a single request(or coalesced) by the hardware.
  - refer to lecture notes:
    - cuda_programming model
    - n-body simulation
- Memory access pattern: Consecutive threads access consecutive memory addresses.





Consecutive threads access consecutive memory addresses.

Transfer anti-diagonals to rows.



The memory layout of threads access.

Consecutive threads access consecutive memory addresses.

Assignment3: References

- Kernel launch parameters setup:
  - #block is set to the number of SM(streaming multiprocessors) of GPU or the multiples of SMs.
    - CS lab2 machines’ GPU has 8 SMs
  - #thread is set to the multiples of 32.
  - E.g: <<<8, 256>>> <<<16, 1024>>> <<<4, 512>>>
- Referential running time:

```
../test/4k.in 8 512
Max score: 12327
Elapsed Time: 0.256098949 s
Driver Time: 0.160557251 s
```

Assignment3: Suggestion

If you have questions about CUDA programming:

- Work hard
- Read your lecture notes
- Read NVIDIA CUDA documents
- Ask Google
- Read previous year’s final exam algorithm code on Maximum Flow problem
- Send emails to TAs