

COMP 4007: Parallel Processing and Computer Architecture

Tutorial 2: Programming with CUDA on GPUs

TA: Wenxiang Lin (869259303@qq.com)

Contents

- Part 1: Programming environment setup
- Part 2: Programming with CUDA
 - Demo / Compile / Debug / Run
- Appendix: matrix multiplication

Part 1:

Programming environment setup

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 find `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
~
```

CUDA Environment on CS Lab2

- Check available GPUs:

```
$ nvidia-smi
```

- Detailed description of the installed GPUs

```
$ mkdir ~/cuda-samples
```

```
$ cp -r /usr/local/cuda-8.0/samples/1_Uutilities ~/cuda-samples/
```

```
$ cp -r /usr/local/cuda-8.0/samples/common ~/cuda-samples/
```

```
$ cd ~/cuda-samples/1_Uutilities/deviceQuery
```

```
$ make
```

```
$ ./deviceQuery
```

CUDA Environment on CS Lab2

NVIDIA-SMI 460.27.04				Driver Version: 460.27.04		CUDA Version: 11.2	
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile Uncorr. ECC		
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.	MIG M.
=====							
0	GeForce GTX 960	Off	00000000:01:00.0	Off		N/A	
39%	27C	P0	26W / 130W	0MiB / 2001MiB	0%	Default	N/A

Processes:							
GPU	GI	CI	PID	Type	Process name	GPU Memory	
	ID	ID				Usage	
=====							
No running processes found							

```
./deviceQuery Starting...

CUDA Device Query (Runtime API) version (CUDA static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 960"
  CUDA Driver Version / Runtime Version      11.2 / 8.0
  CUDA Capability Major/Minor version number: 5.2
  Total amount of global memory:             2002 MBytes (2099052544 bytes)
  ( 8) Multiprocessors, (128) CUDA Cores/MP: 1024 CUDA Cores
  GPU Max Clock rate:                       1240 MHz (1.24 GHz)
  Memory Clock rate:                         3505 Mhz
  Memory Bus Width:                          128-bit
  L2 Cache Size:                             1048576 bytes
  Maximum Texture Dimension Size (x,y,z)     1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers 1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers 2D=(16384, 16384), 2048 layers
  Total amount of constant memory:            65536 bytes
  Total amount of shared memory per block:    49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                  32
  Maximum number of threads per multiprocessor: 2048
  Maximum number of threads per block:        1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                       2147483647 bytes
  Texture alignment:                           512 bytes
  Concurrent copy and kernel execution:       Yes with 2 copy engine(s)
  Run time limit on kernels:                   No
  Integrated GPU sharing Host Memory:         No
  Support host page-locked memory mapping:    Yes
  Alignment requirement for Surfaces:         Yes
  Device has ECC support:                     Disabled
  Device supports Unified Addressing (UVA):    Yes
  Device PCI Domain ID / Bus ID / location ID: 0 / 1 / 0
  Compute Mode:
    < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.2, CUDA Runtime Version = 8.0, NumDevs = 1, Device0 = GeForce GTX 960
Result = PASS
```

Part 2:

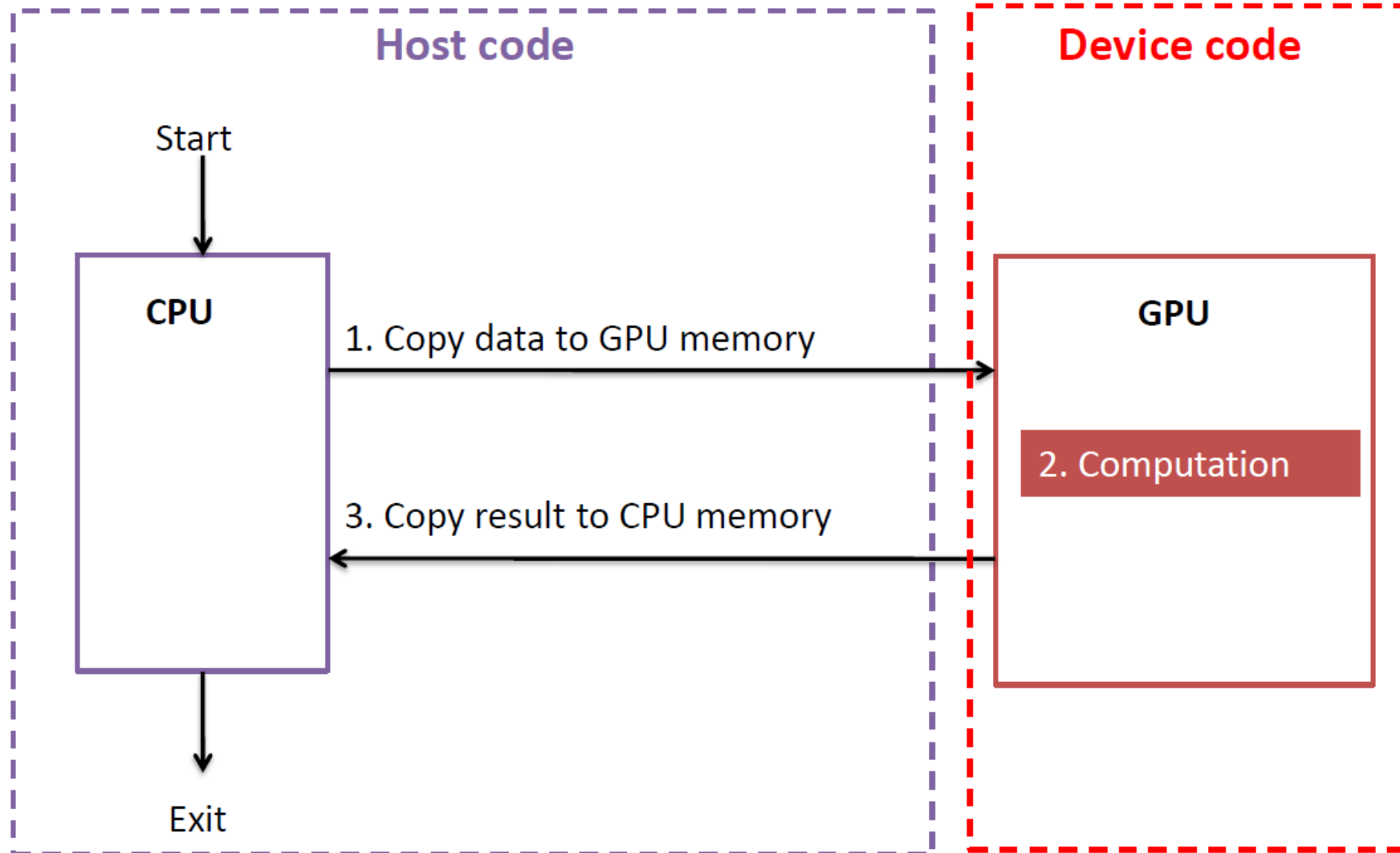
Programming with CUDA

Demo / Compile / Run / Debug

Host and Device code

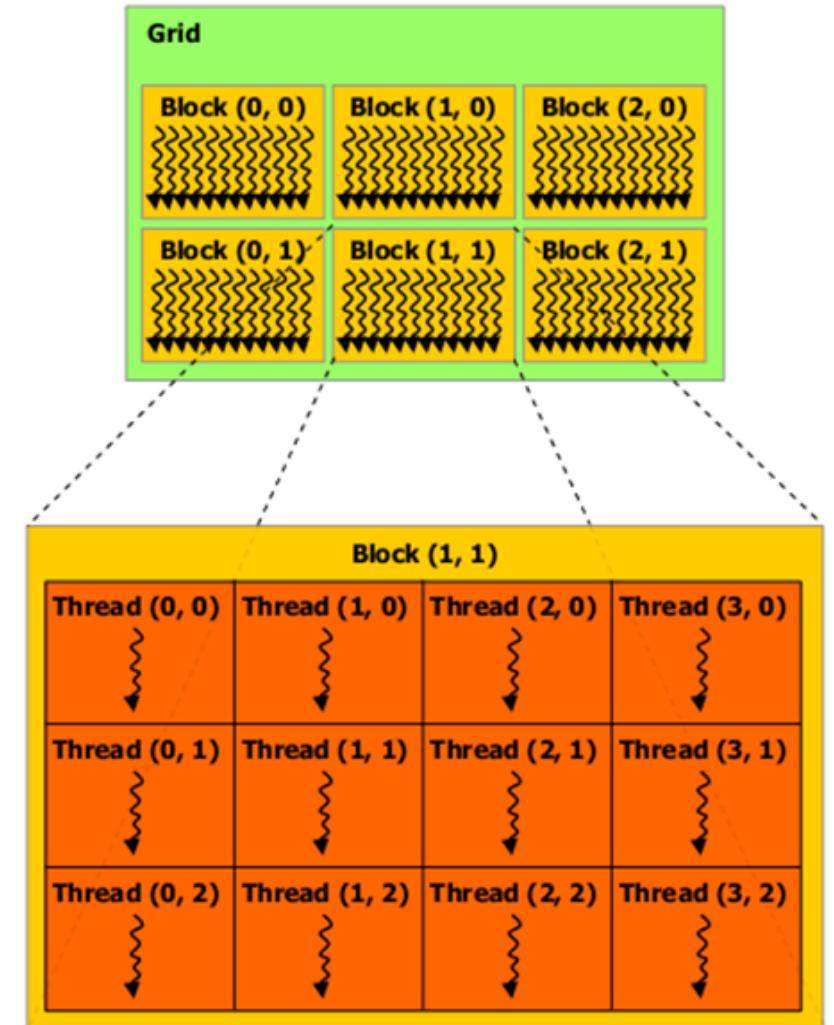
- A CUDA program consists of two parts: *host* and *device* (or *kernel*) code.
- Host code: executed on the CPU
 - Memory copy between the GPU and the CPU
 - Computation on the CPU and call GPU kernel
- Device code: executed on the GPU
 - GPU-based computation
- A CUDA program always starts from the host code, and then invokes the GPU kernels.

Processing Flow of a CUDA Program



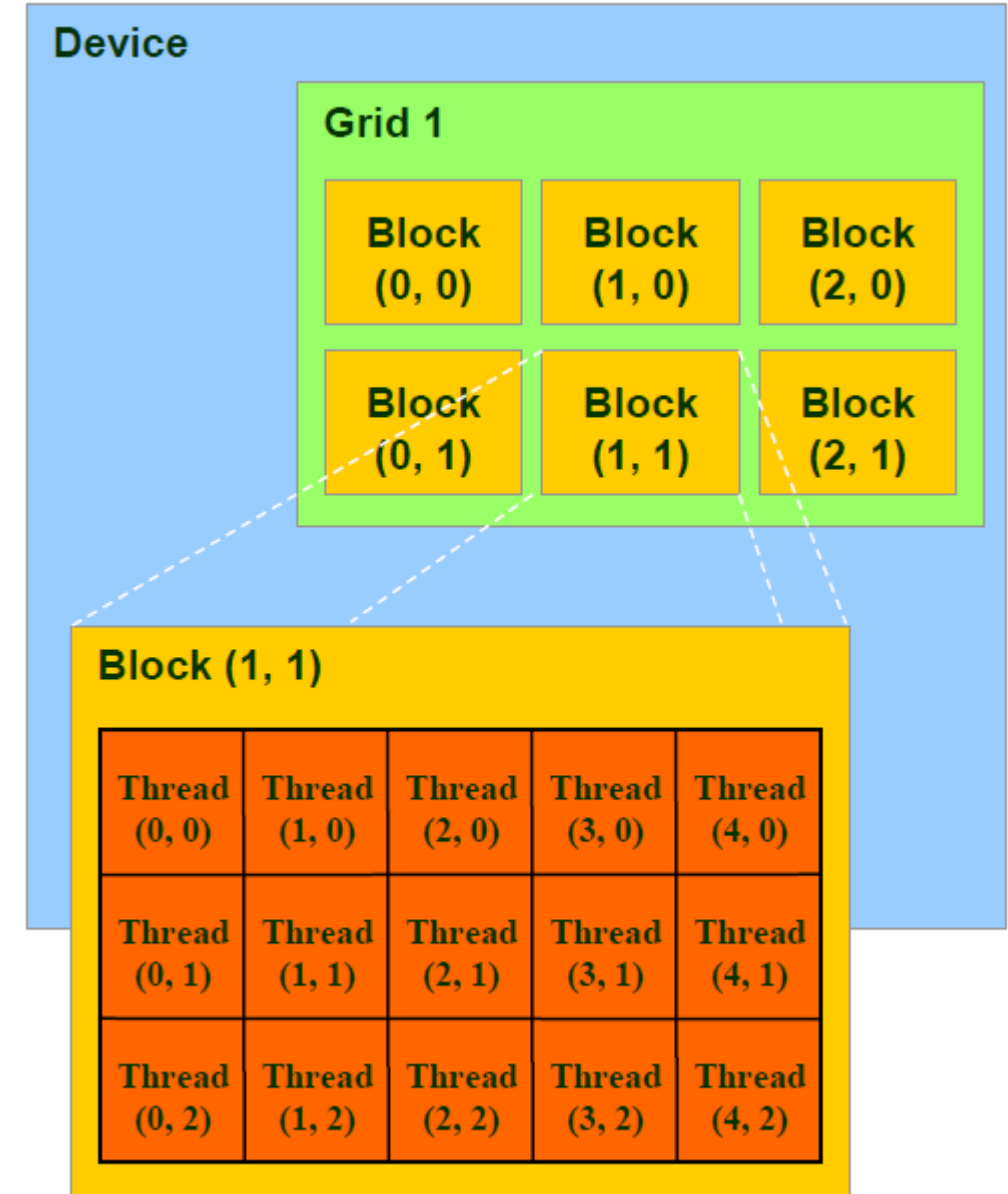
Grid, blocks and threads

- Each kernel corresponds to a **grid** of threads.
- Each grid consists of multiple thread **blocks**.
- Each thread block contains multiple **threads**.
- A grid consists of i-dimension (i=1,2,3) **blocks**
 - gridDim.x, gridDim.y, gridDim.z
- A thread block contains **threads** organized in 1-3 dimensions
 - blockDim.x, blockDim.y, blockDim.z
- Any unspecified dimension is set to size 1.



Block and Thread IDs

- Threads and blocks have built-in IDs
 - Block ID: (blockIdx.x, blockIdx.y, blockIdx.z)
 - Thread ID: 1D, 2D, or 3D **within a block** (threadIdx.x, threadIdx.y, threadIdx.z)



Device Code (Kernel)

- The device code is the same for each thread.
- A kernel function has the prefix `__global__`, and has a *void* return type.

`__global__ void kernel1(param1, ...)`

- Note: device code has no direct access to main memory.

Kernel Invocation in Host Code

kernelName<<<#block, #thread, shared_size, s>>> (param1, ...)

#block: number of thread blocks in the grid

#thread: number of threads per block

shared_size: optional; size of shared memory per block, default 0.

s: optional; the associated stream, default 0.

Memory Management

- In CUDA, host (i.e., CPU) and device (i.e., GPU) have separate memory spaces
- To execute a kernel on GPU, we need to
 1. allocate memory on the device
 2. transfer data from host memory to allocated device memory
- After device execution, we need to transfer the result data from device memory back to host

GPU memory management functions

- GPU memory allocation:
`cudaMalloc(devPtr, size)`
`cudaFree(devPtr)`
- Memory copy:
`cudaMemcpy(dst, src, size, direction)`
direction: cudaMemcpyHostToDevice,
cudaMemcpyDeviceToHost

Demo 1

- ▶ Allocate CPU memory for n integers
- ▶ Allocate GPU memory for n integers
- ▶ Initialize GPU memory to 0s
- ▶ Copy from GPU to CPU
- ▶ Print the values

Demo 1

```
#include <cuda.h>
#include <stdio.h>
int main() {
    int dimx= 16;
    int num_bytes= dimx* sizeof(int);
    int*d_a= 0, *h_a= 0; // device and host pointers
    h_a= (int*)malloc(num_bytes);
    cudaMalloc((void**)&d_a, num_bytes);
    if (0 == h_a|| 0 == d_a) {
        printf("couldn't allocate memory\n");
        return 1;
    }
    cudaMemset(d_a, 0, num_bytes);
    cudaMemcpy(h_a, d_a, num_bytes, cudaMemcpyDeviceToHost);
    for (int i= 0; i< dimx; i++)
        printf("%d\n", h_a[i]);
    free(h_a);
    cudaFree(d_a);
    return 0;
}
```


Demo 1

- Compile
 - `nvcc source_file -o output_file`
- Run
 - `./output_file`

Demo 2

- ▶ Use GPU to initialize an array by thread IDs
- ▶ Copy the array to CPU
- ▶ Print the values

A kernel function:

```
__global__ void mykernel(int* a)
{
    int idx= blockIdx.x* blockDim.x+ threadIdx.x;
    a[idx] = 7;
}
```

```

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

__global__ void mykernel(int* a) {
    int idx= blockIdx.x* blockDim.x+ threadIdx.x; // locate the data item handled by this thread
    a[idx] = threadIdx.x;
}

int main() {
    int dimx= 16, num_bytes= dimx*sizeof(int);
    int*d_a= 0, *h_a= 0; // device and host pointers
    h_a= (int*)malloc(num_bytes);
    cudaMalloc((void**)&d_a, num_bytes);
    cudaMemset(d_a, 0, num_bytes);
    dim3 grid, block;
    block.x= 4; // each block has 4 threads
    grid.x= dimx / block.x; // # of blocks is calculated
    mykernel<<<grid, block>>>(d_a);
    cudaMemcpy(h_a, d_a, num_bytes, cudaMemcpyDeviceToHost);
    for(int i= 0; i< dimx; i++)
        printf("%d\n", h_a[i]);
    free(h_a);
    cudaFree(d_a);
    return 0;
}

```

The output will be:

```
0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
```

Debugging using CUDA-GDB

- What does CUDA-GDB do?
 - Control the execution of the application
 - Breakpoints
 - Single-step
 - CTRL-C
 - Inspect the current state of the application
 - Kernels, blocks, threads
 - Devices, SMs, warps
 - Inspect and Modify
 - Code memory (disassemble)
 - Global, shared, and local memory
 - Hardware registers
 - Textures (read-only)

Debugging using CUDA-GDB

- Compile:
 - `nvcc -g -G XXX.cu -o XXX`
- Enter debug mode:
 - `cuda-gdb`

Debugging using CUDA-GDB

- Execution control
 - launch the application
 - (cuda-gdb) run [arguments]
 - resume the application (all host and dev threads)
 - (cuda-gdb) continue
 - kill the application
 - (cuda-gdb) kill
 - interrupt the application
 - CTRL-C

Debugging using CUDA-GDB

- Breakpoints
 - Symbolic breakpoints
 - (cuda-gdb) break my_kernel
 - Line number breakpoints
 - (cuda-gdb) break my_app.cu:380
 - Address breakpoints
 - (cuda-gdb) break *0x3e840a8
 - Kernel entry breakpoints
 - (cuda-gdb) set cuda break_on_launch application
 - List of breakpoints
 - (cuda-gdb) info breakpoints

Debugging using CUDA-GDB

- Focus Query
 - Query commands
 - `cuda <list of coordinates>`
 - `thread`
 - If focus set to device thread
 - **`(cuda-gdb) cuda kernel block thread`**
 - kernel 1, block (0, 0, 0), thread (0, 0, 0)
 - **`(cuda-gdb) cuda device kernel block warp thread`**
 - kernel 1, block (0, 0, 0), thread (0, 0, 0), device 0, warp 0
 - If focus set to host thread
 - **`(cuda-gdb) thread`**
 - [Current thread is 1 ...]
 - **`(cuda-gdb) cuda thread`**
 - Focus not set on any active CUDA kernel

Debugging using CUDA-GDB

- Focus Switch
 - Switch command
 - cuda <list of coordinate-value pairs>
 - thread <host thread id>
 - Only switch the specified coordinates
 - current coordinates are assumed in case of non-specified coordinates
 - if no current coordinates, best effort to match request
 - error if cannot match request

Debugging using CUDA-GDB

- Focus Switch
 - **(cuda-gdb) cuda kernel 1 block 1 thread 2,0**
 - [Switching focus to CUDA kernel 1, grid 2, block (1, 0, 0), thread (2, 0, 0), device 0, sm 5, warp 0, lane 2]
 - **(cuda-gdb) cuda kernel 1**
 - [Switching focus to CUDA kernel 1, grid 2, block (0, 0, 0), thread (0, 0, 0), device 0, sm 1, warp 0, lane 0]
 - **(cuda-gdb) cuda thread 256**
 - Request cannot be satisfied. CUDA focus unchanged.

Debugging using CUDA-GDB

Step 1: compile using `-g -G` augments

```
nvcc -g -G demo2.cu -o demo2
```

Step 2: enter cuda-gdb

```
cuda-gdb -q demo2
```

Step 3: display some codes

```
list
```

Step 4: Set break point

```
break mykernel
```

```
(cuda-gdb) list
3
4      // Device code
5      __global__ void mykernel(int* a) {
6          int idx= blockIdx.x* blockDim.x+
threadIdx.x; // locate the data item handled by
this thread
7          a[idx] = threadIdx.x;
8      }
9
10     // Host code
11     int main() {
12         int dimx= 16, num_bytes=
dimx*sizeof(int);

(cuda-gdb) break mykernel
Breakpoint 1 at 0x402c00: file demo2.cu, line 5.

(cuda-gdb) break mykernel
Breakpoint 1 at 0x402c00: file demo2.cu, line 5.
```

Debugging using CUDA-GDB

Step 5: Run the program

run

Step 6: run next line

next

Step 7: check cuda threads

info cuda threads

```
(cuda-gdb) run
Starting program: /homes/yxingag/hpc/lab02/demo2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
[New Thread 0x7ffff54ef700 (LWP 27647)]
[New Thread 0x7ffff4cee700 (LWP 27648)]
[New Thread 0x7ffff44ed700 (LWP 27649)]
[Switching focus to CUDA kernel 0, grid 1, block (0,0,0),
thread (0,0,0), device 0, sm 0, warp 0, lane 0]

Breakpoint 1, mykernel<<<(4,1,1),(4,1,1)>>> (a=0x4026e0000)
at demo2.cu:6
6          int idx= blockIdx.x* blockDim.x+ threadIdx.x;
// locate the data item handled by this thread

(cuda-gdb) next
7  a[idx] = threadIdx.x;

(cuda-gdb) info cuda threads
BlockIdx ThreadIdx To BlockIdx ThreadIdx Count
Virtual PC Filename Line
Kernel 0
* (0,0,0) (0,0,0) (0,0,0) (3,0,0) 4
0x0000000000a85a38 demo2.cu 7
(1,0,0) (0,0,0) (3,0,0) (3,0,0) 12
0x0000000000a859e8 demo2.cu 6
```

Debugging using CUDA-GDB

Step 8: print local variables

`info locals`

Step 9: print variables

`print idx`

Step 10: continue execution

`continue`

```
(cuda-gdb) info locals
idx = 0

(cuda-gdb) print idx
$1 = 0

(cuda-gdb) continue
Continuing.
0
1
2
3
...
1
2
3
[Thread 0x7ffff4cee700 (LWP 27648) exited]
[Thread 0x7ffff54ef700 (LWP 27647) exited]
[Thread 0x7ffff7fc9740 (LWP 27599) exited]
[Inferior 1 (process 27599) exited normally]
```

Demo 3: Vector addition

A kernel function which will be executed on GPU

```
// compute vector sum C = A + B
// each thread performs one pair-wise addition
__global__ void vecAdd( float *A, float *B, float *C, int n)
{
    // locate the memory
    int i = threadIdx.x + blockDim.x * blockIdx.x;

    // perform the addition
    if(i < n) C[i] = A[i] + B[i];
}
```

Demo 3: Vector addition

Host code which will be executed on CPU

```
int main ()
{
    int n = 10000;
    // allocate and initialize host (CPU) memory
    float *H_A = ..., *H_B = ..., *H_C = ...;
    // allocate device (GPU) memory
    float *A_d, *B_d, *C_d;
    cudaMalloc(...); ...
    // copy host memory to device
    cudaMemcpy(...);...
    // run 16 blocks of 256 threads each
    vecAdd<<< ceil(n/256.0), 256 >>>(d_A, d_B, d_C, n);
    // copy result to host
    cudaMemcpy(...);
    cudaFree(A_d);...
}
```

Try to debug by yourself!

Error Handling

- ▶ It is very common to use an error-handling macro to wrap all CUDA API calls
 - ▶ To simplify the error checking process

```
#define CHECK(call)
{
    const cudaError_t error = call;
    if (error != cudaSuccess)
    {
        printf("Error: %s:%d, ", __FILE__, __LINE__);
        printf("code:%d, reason: %s\n", error, cudaGetErrorString(error));
        exit(1);
    }
}
```

E.g.:

CHECK(cudaMemcpy(...));

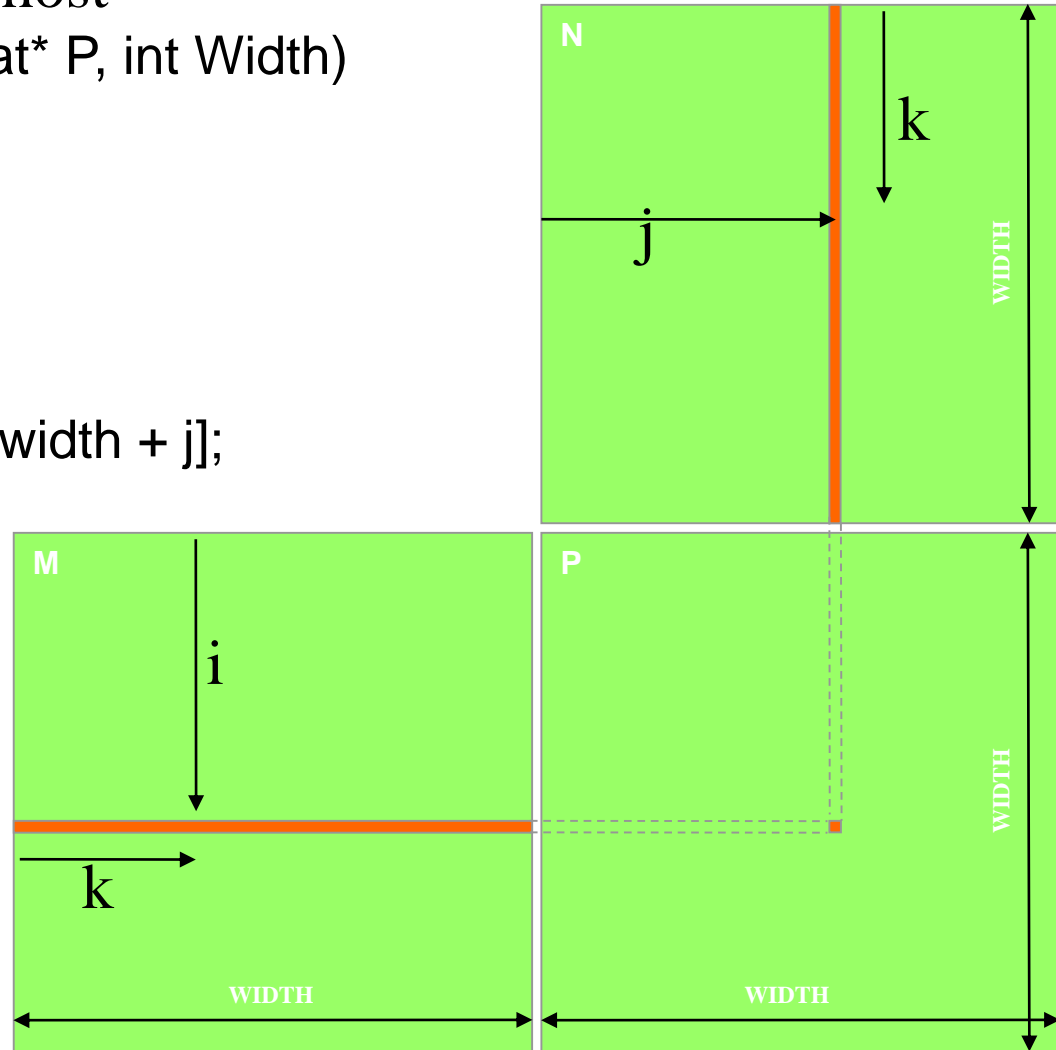
__FILE__ and __LINE__ are standard predefined macros that represent the current input file and input line number, respectively.

Appendix: Matrix multiplication

- ▶ Problem: $P = M \times N$
- ▶ M, N, P are square matrix: $WIDTH \times WIDTH$
- ▶ Parallelization on GPU
 - ▶ The calculation of each item in P is done by a GPU thread
 - ▶ To perform a dot product of two vectors
 - ▶ Question: How to organize the threads into blocks?

Appendix: Serial Solution

```
// Matrix multiplication on the (CPU) host
void MatMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; i++)
        for (int j = 0; j < Width; j++) {
            float sum = 0.0;
            for (int k = 0; k < Width; k++) {
                sum += M[i * width + k] * N[k * width + j];
            }
            P[i * Width + j] = sum;
        }
}
```



Appendix: Matrix multiplication

Please refer to the attached codes of this tutorial!

```
#include <iostream>
#include <cstdio>
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <cmath>
using namespace std;

const int TILE_WIDTH = 16;
__global__ void MatrixMulKernel(int *d_M,int *d_N,int *d_P,int m,int n,int k)
{
    __shared__ int ds_M[TILE_WIDTH][TILE_WIDTH];
    __shared__ int ds_N[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x;
    int by = blockIdx.y;

    int tx = threadIdx.x;
    int ty = threadIdx.y;

    //Identify the row and column of the Pd element to work on
    int row = by * TILE_WIDTH + ty;
    int col = bx * TILE_WIDTH + tx;
    int pValue = 0;
    ....
}
```