GPU Architectures and Basic CUDA Programming

Tong Shu

Day 3 — Part 1
DL-GPU Workshop 2023

07/26/2023

Outline

- Heterogeneous data-parallel computing in CUDA programming
- Multi-dimensional grids and data in CUDA programming
- Architectures of state-of-the-art GPUs
- Memory and data locality in CUDA programming

Heterogeneous Data-Parallel Computing in CUDA Programming

Process vs Thread

- A process is the instance of a computer program that is being executed by one or many threads.
- A thread of execution is the smallest sequence of programmed instructions that can be managed independently by the operating system's scheduler.

Why and How to Optimize Execution Throughput?

- Why?
  - Reducing latency is much more expensive than increasing throughput in terms of power and chip area.

- How?
  - GPU's design saves chip area and power by allowing pipelined memory channels and arithmetic operations to have long-latency.
  - The reduced area and power allows more arithmetic logic units to be on a chip and thus increase the total execution throughput.

Fundamentally Different Processor Design Philosophies

- Latency-oriented design
  - Minimize the execution latency of a single thread
  - Eg., CPU design

- Throughput-oriented design
  - Maximize the total execution throughput of a large number of threads while allowing individual threads to take a potentially much longer time to execute
  - Eg., GPU design

Heterogeneous Parallel Computing

- It is expected that applications use both CPUs and GPUs, executing the sequential parts on the CPU and numerically intensive parts on the GPUs.

Hello World in CUDA C

- Edit the program
  - hello_world.cu

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

__global__ void print_from_gpu(void) {
        printf("Hello World! from thread [%d,%d] \
                From device\n", blockIdx.x, threadIdx.x);
}

int main(void) {
        printf("Hello World from host!\n");
        print_from_gpu<<<2, 3>>>();
        cudaDeviceSynchronize();
        return 0;
}
```

The keyword __global__ tells the compiler that this is a kernel function (running on GPU) and is called by a host function (running on CPU).

The return type of the kernel function is always "void".

(blockIdx.x, threadIdx.x) is a unique ID given to each thread.

The keyword <<<,>>> tells the compiler that this is a call to the kernel function. The parameters inside angle brackets dictates the number of blocks and threads to launch in the kernel.

API cudaDeviceSynchronize() waits until all of the previous calls to the device (GPU) have finished.
All the kernel calls are asynchronous in nature. The host (CPU) becomes free after calling the kernel and starts executing the next instruction.

Hello World in CUDA C (Cont.)

- Compile the program
  - Command

```
$ nvcc -o hello_world hello_world.cu
```

  - Makefile (exact file name: case-sensitive)

```
hello_world: hello_world.cu
<TAB>nvcc -o hello_world hello_world.cu

clean:
<TAB>rm -f hello_world
```

```
$ make
```

```
$ make clean
```

- Run the executable

```
$ ./hello_world
Hello World from host!
Hello World! from thread [0,0]          From device
Hello World! from thread [0,1]          From device
Hello World! from thread [0,2]          From device
Hello World! from thread [1,0]          From device
Hello World! from thread [1,1]          From device
Hello World! from thread [1,2]          From device
```

CUDA Software is Mapped onto GPU Hardware

Software		Hardware
CUDA thread	Executes on	Streaming processor/Single Instruction Multiple Data (SIMD) core
CUDA block		Streaming multiprocessor
Grid/kernel		GPU device

Lab Environments

- NVIDIA GPU driver and CUDA Toolkit installed
  - The NVIDIA System Management Interface is a command line utility, based on top of the NVIDIA Management Library, intended to aid in the management and monitoring of NVIDIA GPU devices.

```
$ nvidia-smi
Fri Jan 15 18:03:02 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 450.80.02    Driver Version: 450.80.02    CUDA Version: 11.0     |
|-------------------------------+----------------------+----------------------+
| 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 745     Off  | 00000000:01:00.0 Off |                  N/A |
| 20%   26C    P8    N/A /  N/A |    146MiB /  4040MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      2465      G   /usr/libexec/Xorg                  33MiB |
|    0   N/A  N/A      2637      G   /usr/bin/gnome-shell              109MiB |
+-----------------------------------------------------------------------------+
```

  - CUDA compiler driver

```
$ which nvcc
/usr/local/cuda-11.0/bin/nvcc
```

```
$ cuda-install-samples-11.0.sh ~
$ cd ~/NVIDIA_CUDA-11.0_Samples/1_Utilities
$ make
$ ./deviceQuery
./deviceQuery Starting...
 CUDA Device Query (Runtime API) version (CUDART static linking)
Detected 1 CUDA Capable device(s)
Device 0: "GeForce GTX 745"
  CUDA Driver Version / Runtime Version          11.0 / 11.0
  CUDA Capability Major/Minor version number:    5.0
  Total amount of global memory:                 4041 MBytes (4236902400 bytes)
  ( 3) Multiprocessors, (128) CUDA Cores/MP:     384 CUDA Cores
  GPU Max Clock rate:                            1032 MHz (1.03 GHz)
  Memory Clock rate:                             900 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 2097152 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 1 copy engine(s)
  Run time limit on kernels:                     Yes
  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 supports Managed Memory:                Yes
  Device supports Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  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 = 11.0, CUDA Runtime Version = 11.0, NumDevs = 1
Result = PASS
```

Data Parallelism vs. Task Parallelism

- Data parallelism
  - A form of parallel computing for multiple processors using a technique for distributing the data across different parallel processor nodes
  - The computation around the data is reorganized so that the resulting independent computations can be executed in parallel to complete the overall job faster.
  - The main source of scalability for parallel programs
  - E.g., color-to-greyscale conversion

- Task parallelism
  - A form of parallel computing for multiple processors using a technique for distributing execution of processes and threads across different parallel processor nodes
  - Typically exposed through task decomposition of applications
  - Exists if the two tasks can be done independently
  - E.g., streams

Example of Data Parallelism

- Convert a color image to greyscale
  - For each pixel, the luminance value $L = 0.21r + 0.72g + 0.07b$
  - The input: an image organized as an array I of RGB values
  - The output: a corresponding array O of luminance values
  - None of these per-pixel computations depends on each other

Compilation Process of a CUDA C Program

- NVIDIA C Compiler (NVCC) processes a CUDA C program, using the CUDA keywords to separate the host code and device code.
  - The host code is ANSI C code, which is compiled with the host's standard C/ C++ compilers and runs as a CPU process.
  - The device code is marked with CUDA keywords for data parallel functions (called kernels) and their associated helper functions and data structures, and is compiled by a run-time component of NVCC and executed on a GPU.

Execution of a CUDA Program

- Scheduling efficiency
  - GPU threads take very few clock cycles to generate and schedule due to efficient hardware support.
  - CPU threads typically take thousands of clock cycles to generate and schedule.

A Vector Addition C Code on CPU

vector_addition_cpu.c:

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

// Compute vector sum h_C = h_A + h_B with the host function
void vecAdd(float* h_A, float* h_B, float* h_C, int n) {
        for (int i = 0; i < n; i++)
                h_C[i] = h_A[i] + h_B[i];
}

int main(void) {
        float *h_A, *h_B, *h_C;
        int n = 512;
        int size = n * sizeof(float);

        // Memory allocation for h_A, h_B, and h_C
        h_A = (float *) malloc(size);
        h_B = (float *) malloc(size);
        h_C = (float *) malloc(size);

        // Setup input values into each of n elements of h_A and h_B
        for(int i = 0; i < n; i++)    h_A[i] = (float) i;
        for(int i = 0; i < n; i++)    h_B[i] = (float) i;

        vecAdd(h_A, h_B, h_C, n);    // Call the host function for vector addition

        for(int i = 0; i < n; i++) printf("%.1f + %.1f = %.1f\n", h_A[i] , h_B[i], h_C[i]);    // Output the results

        free(h_A); free(h_B); free(h_C);    // Free host memory for A, B, C
        return 0;
}
```

h_A is passed by reference to function vecAdd(). As as result, h_A[i] in the function body of vecAdd() can be used to access h_A[i] in function main().

Dynamically allocated 1-D arrays

Implement vecAdd() with a Kernel Function

- Step 1
  - Allocate the device (GPU) memory space to hold copies of the vectors A, B, and C
  - Copy vectors A and B from the host memory to the device memory
- Step 2
  - Launch kernel code to have the device to perform the actual vector addition in parallel execution
- Step 3
  - Copy the sum vector C from the device memory back to the host memory
  - Free all the space of vectors A, B, and C in device memory

CUDA Runtime API: Allocate Memory on the Device

- __host__ __device__ cudaError_t cudaMalloc (void** devPtr, size_t size)
  - Parameters
    - devPtr: the address of a pointer to allocated device memory
    - size: requested allocation size in bytes
  - Returns (cudaError_t: CUDA error types)
    - cudaSuccess = 0: The API call returned with no errors.
    - cudaErrorInvalidValue = 1: One or more of the parameters passed to the API call is not within an acceptable range of values.
    - cudaErrorMemoryAllocation = 2: The API call failed because it was unable to allocate enough memory to perform the requested operation.
  - Description
    - Allocate size bytes of linear memory on the device and returns in *devPtr a pointer to the allocated memory.
    - The allocated memory is suitably aligned for any kind of variable.

CUDA Runtime API: Copy Data between Host and Device

- __host__ cudaError_t cudaMemcpy (void* dst, const void* src, size_t count, cudaMemcpyKind kind)
  - Parameters
    - dst: destination memory address
    - src: source memory address
    - count: size in bytes to copy
    - kind: type of transfer
  - Returns
    - cudaSuccess = 0
    - cudaErrorInvalidValue = 1
    - cudaErrorInvalidMemcpyDirection = 21: The direction of the memcpy passed to the API call is not one of the types specified by cudaMemcpyKind.

  - Description
    - Copy count bytes from the memory area pointed to by src to the memory area pointed to by dst, where kind specifies the direction of the copy, and must be one of cudaMemcpyHostToHost=0, cudaMemcpyHostToDevice=1, cudaMemcpyDeviceToHost=2, cudaMemcpyDeviceToDevice=3, or cudaMemcpyDefault=4.

CUDA Runtime API: Frees Memory on the Device

- __host__ __device__ cudaError_t cudaFree (void* devPtr)
  - Parameters
    - devPtr: device pointer to memory to free
  - Returns
    - cudaSuccess = 0
    - cudaErrorInvalidValue = 1
  - Description
    - Free the memory space pointed to by devPtr, which must have been returned by a previous call to cudaMalloc().
    - If cudaFree(devPtr) has already been called before, an error is returned.
    - If devPtr is 0, no operation is performed.

CUDA C Keywords for Function Declaration

	Executed on the	Only callable from the
__device__ float DeviceFunc()	device	device
__global__ void KernelFunc()	device	host
__host__ float HostFunc()	host	host

- __global__: A kernel function executes on the device and can only be called from the host function.
- __ device__: A device function executes on a device and can only be called from a kernel function or another device function.
- __host__: A host function executes on host and can only be called from another host function.
  - By default, all functions are host functions.
- Both __host__ and __device__: Two versions of object files for the same function are generated
  - One is executed on the host and can only be called from a host function.
  - The other is executed on the device and can only be called from a device or kernel function.

A Stub Function for Launching a Kernel

- Transparent outsourcing

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n) {
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;    // pointers to device copies of A, B, C

    // Allocate device memory space for device copies of A, B, C
    cudaMalloc((void **) &d_A, size);
    cudaMalloc((void **) &d_B, size);
    cudaMalloc((void **) &d_C, size);

    // Copy vectors A and B from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Launch the kernel function to have the device to perform the actual vector addition
    int threads_per_block = 256;
    int no_of_blocks = ceil(n / (float) threads_per_block);
    vecAddKernel<<<no_of_blocks, threads_per_block>>>(d_A, d_B, d_C, n);

    // Copy result vector C from the device memory to host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);    // Free device memory for A, B, C
}
```

Dynamically allocated 1-D arrays

Coordinating the Host and the Device

- Kernel launches are asynchronous
  - Control returns to the CPU immediately

- CPU needs to synchronize before consuming the results
  - cudaMemcpy()
    - Blocks the CPU until the copy is complete
    - Copy begins when all preceding CUDA calls have completed
  - cudaMemcpyAsync()
    - Asynchronous, does not block the CPU
  - cudaDeviceSynchronize()
    - Blocks the CPU until all preceding CUDA calls have completed

Execution Configuration Parameters

- In host functions
  - Specified inside angle brackets <<< , >>> in a kernel launch statement
    - 1st configuration parameter: the number of blocks in the grid
    - 2nd configuration parameter: the number of threads pre block

- In kernel functions
  - gridDim: the dimensions of the grid in the number of blocks
  - blockDim: the dimensions of each block in the number of threads

Loop Parallelism

- Iterations of the original sequential code are executed by threads in parallel.
- Kernel function for vector addition

```
// Compute vector sum C = A + B
// Each thread performs one pair-wise addition
__global__ void vecAddKernel(float* d_A, float* d_B, float* d_C, int n) {
        int i = blockDim.x * blockIdx.x + threadIdx.x;
        if (i < n)
                d_C[i] = d_A[i] + d_B[i];
}
```

  - A version of i will be generated for every thread.

- Compared to the host function in traditional C for vector addition

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n) {
        for (int i = 0; i < n; i++)
                h_C[i] = h_A[i] + h_B[i];
}
```

Kernel Functions and Threading

- When a kernel is launched, the CUDA run-time system generates a grid of threads in a two-level hierarchy.
  - Each grid is organized as an array of thread blocks (referred to as blocks).
  - All blocks of a grid are of the same size (up to 1024 threads per block).

- All the thread blocks can be executed in any arbitrary order.

- Single-Program Multiple-Data (SPMD)

Built-in Variables

- The values of built-in variables
  - Often preinitialized by the run-time system
  - Typically read-only in the program

- blockDim: the number of threads in a block
  - Struct type with three unsigned integer fields x, y, and z to help organize the threads into a 1D, 2D, or 3D array.
  - The number of threads in each dimension of blocks should be multiples of 32 due to hardware efficiency.

- (blockIdx, threadIdx)
  - threadIdx gives each thread a unique coordinate within a block.
  - blockIdx gives all threads in a block a common block coordinate.

Error Checking and Handling

- __host__ __device__ const char* cudaGetErrorString (cudaError_t error)
  - Parameters
    - error: error code to convert to string
  - Returns
    - char* pointer to a NULL-terminated string
  - Description
    - Returns the description string for an error code
- Example

```
cudaError_t err = cudaMalloc((void **) &d_A, size);
if (err != cudaSuccess) {
      printf("%s in %s at line %d\n", cudaGetErrorString(err), __FILE__, __LINE__);
      exit(EXIT_FAILURE);
}
```

  - Macros
    - __FILE__ is a preprocessor macro that expands to full path to the current file.
    - __LINE__ is a preprocessor macro that expands to current line number in the source file, as an integer.
    - EXIT_FAILURE = 1

Hands-on Exercise 1

- Implement the dot product of a vector in CUDA C

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

// Implement the kernel function
__global__ void vectorElementMultiplKernel(float* d_C,
            float* d_A, float* d_B, int n) {
      …
}

// The host stub function
void vectorDotProduct(float* h_C, float* h_A, float* h_B, int n) {
      // Implement Phase 1, in which the kernel function is launched.
      …
      // Phase 2: sum reduction
      for (int i = 1; i < n; i++)
            h_C[0] += h_C[i];
}
```

```
int main(void) {
      float *h_A, *h_B, *h_C;
      int n = 64, size = n * sizeof(float);
      h_A = (float *) malloc(size);
      h_B = (float *) malloc(size);
      h_C = (float *) malloc(size);
      for (int i = 0; i < n; i++) {
            h_A[i] = (float) i;
            h_B[i] = (float) i;
      }

      printf("Vector A = \n");
      for(int i = 0; i < n; i++)
            printf("%.1f\t", h_A[i]);
      printf("\nVector B = \n");
      for(int i = 0; i < n; i++)
            printf("%.1f\t", h_B[i]);

      vectorDotProduct(h_A, h_B, h_C, n);

      printf("\nDot product = %.1f\n", h_C[0]);
      free(h_A);
      free(h_B);
      free(h_c);
      return 0;
}
```

Multi-Dimensional Grids and Data in CUDA Programming

CUDA Thread Organization

- Hierarchical thread organization
  - A grid is a three-dimensional array of blocks.
  - Each block is a three dimensional array of threads.

- Configuration parameter type

```
struct dim3 {
        unsigned int  x;
        unsigned int  y;
        unsigned int  z;
};
```

  - E.g.,

```
dim3 dimGrid(ceil(n / 256.0), 1, 1);
dim3 dimBlock(256, 1, 1);
vecAddKernel<<<dimGrid, dimBlock>>>(…);
```

  - Shortcut for 1D configurations `vecAddKernel<<<ceil(n / 256.0), 256>>>(…);`

- The total size of a block is up to 1024 threads.
  - blockDim(512, 1, 1), blockDim(8, 16, 4), and blockDim(32, 16, 2) **OK!**
  - blockDim(32, 32, 2) **NO!**

A Multidimensional Example of Grid Organization

- Block and thread labeling notation
  - The highest dimension comes first
  - In the reversed order of that used in the C statements for setting configuration parameters where the lowest dimension comes first
  - Block(blockIdx.y, blockIdx.x)
    - E.g., Block(1,0) has blockIdx.y=1 and blockIdx.x=0
  - Thread(threadIdx.z, threadIdx.y, threadIdx.x)
    - E.g., Thread(1,0,2) has threadIdx.z=1, threadIdx.y=0, and threadIdx.x=2

Multidimensional Arrays

- A multidimensional array is linearized into an equivalent 1D array for storage
  - Row-major layout, e.g., C

location(M[i, j]) = address of M[row_lb, col_lb] + (((i - row_lb) * n) + (j - col_lb)) * element_size

  - Column-major layout, e.g., FORTRAN

Static vs. Dynamic Memory Allocation

- Static memory allocation
  - Memory is allocated at compile time before the associated program is executed
  - Programmers can use higher-dimensional indexing syntax such as A[i][j] to access their elements.
  - The compiler linearizes them into an equivalent 1-D array and translates the multidimensional indexing syntax into a 1-D offset.
  - E.g., C

```
float v;
float A[10];
float B[10][20];
```

- Dynamic memory allocation
  - Memory in heap is allocated as required at run time.
  - The compiler cannot translate the multidimensional indexing syntax into a 1-D offset because of the lack of dimensional information at compile time
  - E.g., C

```
float *p = (float *) malloc(sizeof(float));
float *P = (float *) malloc(3 * sizeof(float));
float *P = (float *) malloc(2 * 3 * sizeof(float));
```

  - E.g., CUDA C

```
float *p; cudaMalloc((void **) &p, sizeof(float));
float *P; cudaMalloc((void **) &P, sizeof(3 * float));
float *Q; cudaMalloc((void **) &Q, sizeof(2 * 3 * float));
```

Convert a Color Image to Greyscale

- Input
  - An image organized as an array I of RGB values (r, g, b)
- Output
  - A corresponding array O of luminance values
- Process
  - For each pixel, the luminance value $L = 0.21r + 0.72g + 0.07b$

Color Image to Greyscale: C Code on CPU

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define WIDTH 15
#define HEIGHT 10

struct Pixel {
	unsigned char r;
	unsigned char g;
	unsigned char b;
};
```

```c
void colorToGreyscale(unsigned char *h_out, struct Pixel *h_in, int w, int h) {
	for(int i = 0; i < h; i++) {
		for(int j = 0; j < w; j++) {
			h_out[i * w + j] = round(0.21 * h_in[i * w + j].r
						+ 0.71 * h_in[i * w + j].g + 0.07 * h_in[i * w + j].b);
		}
	}
}
int main(void) {
	int colorImageSize = HEIGHT * WIDTH * sizeof(struct Pixel);
	struct Pixel *h_in = (struct Pixel *) malloc(colorImageSize);
	int greyImageSize = HEIGHT * WIDTH * sizeof(unsigned char);
	unsigned char *h_out = (unsigned char *) malloc(greyImageSize);
	for(int i = 0; i < HEIGHT; i++) {
		for(int j = 0; j < WIDTH; j++) {
			// Randomly generate a color image
			h_in[i * WIDTH + j].r = rand() % 256;
			h_in[i * WIDTH + j].g = rand() % 256;
			h_in[i * WIDTH + j].b = rand() % 256;
		}
	}
	colorToGreyscale(h_out, h_in, WIDTH, HEIGHT);
	for(int i = 0; i < HEIGHT; i++) {
		for(int j = 0; j < WIDTH; j++) {
			printf("Output pixel (%d, %d) = %d\n",
						i, j, h_out[i * WIDTH + j]);
		}
	}
	free(h_in); free(h_out);
	return 0;
}
```

Using a 2D grid to Process a Picture

- A $m \times n$ picture P
  - m pixels in the horizontal or x direction
  - n pixels in the vertical or y direction
  - $P_{i,j}$: pixel in the ith row and the jth column
- The pixel processed by Thread(threadIdx.y, threadIdx.x) of Block(blockIdx.y, blockIdx.x) is

$$P_{\text{blockIdx.y}\times\text{blockDim.y}+\text{threadIdx.y},\text{blockIdx.x}\times\text{blockDim.x}+\text{threadIdx.x}}$$

row = threadIdx.y + blockIdx.y * blockDim.y
from 0 to blockDim.y * gridDim.y - 1

$P_{\text{blockIdx.y}\times\text{blockDim.y}+\text{threadIdx.y},\text{blockIdx.x}\times\text{blockDim.x}+\text{threadIdx.x}}$
$= P_{1\times16+0,0\times16+0} = P_{16,0}$
$\rightarrow P[16 \times 76 \times 3 + 0] = P[3648]$

col = blockIdx.x*blockDim.x+threadIdx.x from 0 to blockDim.x * gridDim.x - 1

A Stub Function for Launching a Kernel

```
void colorToGreyscale(unsigned char *h_out, struct Pixel *h_in, int w, int h) {
    struct Pixel *d_in;
    unsigned char *d_out;

    int colorImageSize = h * w * sizeof(struct Pixel);
    int greyImageSize = h * w * sizeof(unsigned char);

    cudaMalloc((void **) &d_in, colorImageSize);
    cudaMalloc((void **) &d_out, greyImageSize);

    cudaMemcpy(d_in, h_in, h * w * sizeof(struct Pixel), cudaMemcpyHostToDevice);

    dim3 dimGrid(ceil(w / (float) 8), ceil(h / (float) 8), 1);
    dim3 dimBlock(8, 8, 1);
    colorToGreyscaleKernel<<<dimGrid,dimBlock>>>(d_in, d_out, w, h);

    cudaMemcpy(h_out, d_out, h * w * sizeof(unsigned char), cudaMemcpyDeviceToHost);

    cudaFree(d_in);
    cudaFree(d_out);
}
```

Kernel Function for Color Image to Greyscale

```
__global__
void colorToGreyscaleKernel(unsigned char *d_out, struct Pixel *d_in, int w, int h) {
    int col = threadIdx.x + blockIdx.x * blockDim.x;
    int row = threadIdx.y + blockIdx.y * blockDim.y;

    if (col < w && row < h) {
        int offset = row * w + col;
        d_out[offset] = round(0.21 * d_in[offset].r
                + 0.71 * d_in[offset].g + 0.07 * d_in[offset].b);
    }
}
```

The Number of threads Processing Pixels

16×16 block

3D Arrays

- col = blockIdx.x*blockDim.x+threadIdx.x
- row = blockIdx.y * blockDim.y + threadIdx.y
- plane = blockIdx.z * blockDim.z + threadIdx.z

- Linearized access to a 3-D array P is of the form P[plane*m*n+row*m+col]

Image Blur

- In computer image processing algorithms
  - Reduce the impact of noise and granular rendering effects in an image by correcting problematic pixel values with the clean surrounding pixel values.
- In computer vision
  - Allow edge detection and object recognition algorithms to focus on thematic objects rather than being impeded by a massive quantity of fine-grained objects.

Patch in Image Blur

- Each output pixel is the average of a $N \times N$ patch of pixels surrounding, and including, the target pixel in the input image

- When calculating an output pixel value at the (row, col) position, a patch is centered at the input pixel located at the (row, col) position.

- E.g., a 3×3 patch spans 3 rows (row-1, row, row+1) and 3 columns (col-1, col, col+1).
  - #define BLUR_SIZE 1

An Image Blur Kernel

```
__global__ void imageBlurKernel(struct Pixel * d_out, struct Pixel * d_in, int w, int h) {
    int col = threadIdx.x + blockIdx.x * blockDim.x;
    int row = threadIdx.y + blockIdx.y * blockDim.y;

    if (col < w && row < h) {
        int sumPixRed = 0, sumPixGreen = 0, sumPixBlue = 0, numPix = 0;

        // Get the average of the surrounding BLUR_SIZE x BLUR_SIZE box
        for (int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE + 1; ++blurRow) {
            for (int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE + 1; ++blurCol) {
                int curRow = row + blurRow;
                int curCol = col + blurCol;

                // Verify we have a valid image pixel
                if (curRow >= 0 && curRow <= h - 1 && curCol >= 0 && curCol <= w - 1) {
                    // Keep track of number of pixels in the average
                    sumPixRed += d_in[curRow * w + curCol].r;
                    sumPixGreen += d_in[curRow * w + curCol].g;
                    sumPixBlue += d_in[curRow * w + curCol].b;
                    numPix++;
                }
            }
        }

        d_out[row * w + col].r = (unsigned char) round(sumPixRed / (float) numPix);
        d_out[row * w + col].g = (unsigned char) round(sumPixGreen / (float) numPix);
        d_out[row * w + col].b = (unsigned char) round(sumPixBlue / (float) numPix);
    }
}
```

Hands-on Exercise 2

- Implement the matrix-vector multiplication in CUDA C

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

#define NUM_COL 2
#define NUM_ROW 3

struct Dim2 {
        unsigned char nc;    // the number of columns
        unsigned char nr;    // the number of rows
};

__global__ void matVecMulKernel(float* d_A, float* d_B,
                float* d_C, struct Dim2 dim) {
        // Implement the kernel function
}

void matVecMul(float* h_A, float* h_B,
                float* h_C, struct Dim2 dim) {
        // Implement the host stub function
}
```

```
int main(int argc, char *argv[]) {
        float *h_A = (float *) malloc(NUM_ROW * sizeof(float));
        float *h_B = (float *) malloc(NUM_ROW * NUM_COL
                        * sizeof(float));
        float *h_C = (float *) malloc(NUM_COL * sizeof(float));
        printf("B = \n");
        for(int row = 0; row < NUM_ROW; row++) {
                for(int col = 0; col < NUM_COL; col++) {
                        int offset = row * NUM_COL + col;
                        h_B[offset] = (float) (rand() % 10);
                        printf("%.1f\t", h_B[offset]);
                }
                printf("\n");
        }
        printf("\nC = \n");
        for(int row = 0; row < NUM_COL; row++) {
                h_C[row] = (float) (rand() % 10);
                printf("%.1f\n", h_C[row]);
        }
        struct Dim2 dim;
        dim.nc = NUM_COL;
        dim.nr = NUM_ROW;
        matVecMul(h_A, h_B, h_C, dim);
        printf("\nA = \n");
        for(int row = 0; row < NUM_ROW; row++) {
                printf("%.1f\n", h_A[row]);
        }
        free(h_A); free(h_B); free(h_C);
        return 0;
}
```

Question & Answer