ECE 408 Exam 1, Spring 2025
Tuesday, March 4, 7:00 – 10:00 PM

Name: Heng An Cheng

NetID: hacheng2

UIN: 667692370

- **Be sure your exam booklet has exactly FIFTEEN pages.**
- **Write your name at the top of each page.**
- **Do not tear the exam apart.**
- **This is a closed-book exam; NO handwritten notes are permitted.**
- **You may not use any electronic devices except for a simple calculator.**
- **Absolutely no interaction between students is allowed.**
- **Show all work, and clearly indicate any assumptions that you make.**
- **Assume all omitted code is implemented correctly.**
- **Illegible answers will likely be graded as incorrect.**
- **Don't panic, and good luck!**

Problem 1	30	points	_____
Problem 2	10	points	_____
Problem 3	20	points	_____
Problem 4	40	points	_____
Total	100	points	_____

Name: Heng An Cheng

Problem 1 (30 points): The Ideas of Massively Parallel Programming

In both **Part A** and **Part B**, select either **True** or **False** for each statement, and provide a **one-sentence** justification for your response. Each statement will be graded as follows:

- **+2 points:** Correct answer with adequate justification
- **+1 point:** Correct answer with insufficient justification
- **0 points:** Blank answer or correct answer without justification
- **-1 point:** Incorrect answer and/or justification

A net negative score will reduce your total exam score.

Part A (20 points): **(A.1)** Host code can transfer data from host to device, and device code can transfer data from device to host. ○ True ◉ False

only kernel code can transfer data between host and device

(A.2) Starting around 2004, computer architects began incorporating sophisticated branch prediction logic and increasing clock frequencies more rapidly with each process generation—as transistor sizes shrank—in order to reduce chip power consumption. ○ True ◉ False

the increase of clock frequencies will increase chip power consumption

(A.3) The sign function ($sign(x) = 1$ for $x > 0$, $sign(0) = 0$, and $sign(x) = -1$ for $x < 0$) could not be used as a good DNN activation function. ◉ True ○ False

it's not a smooth function.

(A.4) Consecutive threads within the same block are assigned to the same or contiguous warps, and consecutive blocks within the same grid are allocated to the same or neighboring SMs. ○ True ◉ False

Consecutive blocks should only be in same SM in same grid.

(A.5) Charles asked the ECE408 chatbot the following question:

```
helppp my code no worky <pastes Charles's lab9.cu>
```

His inquiry adhered to the course chatbot policy. ○ True ◉ False

your cannot directly paste your code to the chatbot

(A.6) Control divergence can lead to inefficiency because if threads from the same warp take different paths, all threads must wait until the longest path is completed, potentially leading to idle cycles for some threads. ○ True ◉ False

all threads will run the all different paths sequentially from the same warp

Problem 1, continued:

(A.7) A constant memory read is not always faster than a global memory read. ○ True ✓ False

constant memory read is always faster

(A.8) The CUDA compiler generates an intermediate representation called PTX, which is not specific to any particular GPU. At runtime, the GPU driver just-in-time compiles the PTX into GPU-specific assembly. ○ True ✓ False

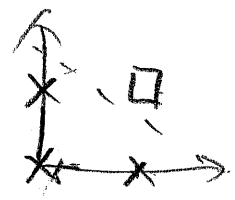
At runtime, GPU driver will compile first then running assembly.

(A.9) Each individual thread has its own copy of the local variables stored in local memory. ○ True ○ False

(A.A) We can learn NAND with a perceptron. ✓ True ○ False

0 0 1
0 1 1
1 0 1
1 1 0

one line (a perceptron) is enough to seperate the plane of NAND



Part B (10 points): `fun` is supposed to compute the prefix sum over `in`, storing results in `out` and returning the summation of `out` to `sum`. Yue came up with the following CPU version of code to do it:

```
1  int fun(int in[], int out[], int in_len, int out_len) {
2      int sum = out[0] = in[0];
3      for (int i = 1; i < in_len && i < out_len; ++i) {
4          out[i] = in[i] + out[i - 1];
5          sum += out[i];
6      }
7      for (int i = in_len; i < out_len; ++i) {
8          out[i] = out[in_len - 1];
9          sum += out[i];
10     }
11     return sum;
12 }
```

The following CUDA kernel is written by Xiyue to finish the same task on a CUDA-capable GPU:

```
1  __global__ void fun_kernel(int in[], int out[], int in_len, int out_len, int* sum) {
2      if (threadIdx.x == 0) {
3          out[0] = in[0];
4      }
5      for (int i = 1; i < in_len && i < threadIdx.x; ++i) {
6          out[i] = in[i] + out[i - 1];
7      }
8      __syncthreads();
9      if (threadIdx.x >= in_len && threadIdx.x < out_len) {
10         out[threadIdx.x] = out[in_len - 1];
11     }
12     *sum += out[threadIdx.x];
13 }
14
15 // d_in already holds the input data in the device memory
16 // d_out is allocated in the device memory
17 // d_sum is initialized to 0 in the device memory
18 // The kernel is launched from the host as follows:
19 fun_kernel<<<1, 128>>>(d_in, d_out, 80, 128, &d_sum);
```

Name: Heng An Cheng

Problem 1, continued:

The CUDA kernel is duplicated for your convenience:

```
1  __global__ void fun_kernel(int in[], int out[], int in_len, int out_len, int* sum) {
2      if (threadIdx.x == 0) {
3          out[0] = in[0];
4      }
5      for (int i = 1; i < in_len && i < threadIdx.x; ++i) {
6          out[i] = in[i] + out[i - 1];
7      }
8      __syncthreads();
9      if (threadIdx.x >= in_len && threadIdx.x < out_len) {
10         out[threadIdx.x] = out[in_len - 1];
11     }
12     *sum += out[threadIdx.x];
13 }
14
15 // d_in already holds the input data in the device memory
16 // d_out is allocated in the device memory
17 // d_sum is initialized to 0 in the device memory
18 // The kernel is launched from the host as follows:
19 fun_kernel<<<1, 128>>>(d_in, d_out, 80, 128, &d_sum);
```

0~31 ✓
32~63 ✓
64~95 ✓
96~128

(B.1) The `fun_kernel` produces the same `out` array as `fun`. ○ True ✓ False

For line #6, it might have race condition for out[i], leading to wrong out array.

(B.2) The `fun_kernel` produces the same `sum` as `fun`. ○ True ✓ False

the out array might be wrong, so the sum might be also wrong.

(B.3) During this kernel launch, only two warps experience control divergence. ○ True ✓ False

3 warps will have control divergence because of line #5~#7 and #9

(B.4) The global memory write to `out` on line 10 is coalesced. ✓ True ○ False

memory write with adjacent threads to consecutive memory

(B.5) Yifei added a `__syncthreads` call between lines 6 and 7 would not change the overall functionality. The block synchronization would slow down execution. ○ True ✓ False

That will prevent the possible race conditions, and produces correct out array as fun.

Problem 2 (10 points): GPU Benchmarking

Given a heterogeneous CPU-GPU system, Hrishi and Colin are asked to deploy a benchmark program, which is developed by Vijay, to calibrate the A40 GPU performance.

The system has the following characteristics:

- Contains a single-core CPU and an A40 GPU.
- The CPU is running under the clock at 3.80 GHz, and the GPU is running under the boost mode.
- Each multiply-add operation requires 3 CPU cycles or 460 GPU cycles per CUDA thread to complete on this system.

The benchmark program has the following characteristics:

- $\frac{\text{Parallel workload}}{\text{Sequential workload}} = 2.00$
- On a CPU-only system, both parallel and sequential workloads are done by the CPU; In this CPU-GPU system, the sequential workloads are only done by the CPU, and the parallel workloads are carried out only by the GPU.
- In this benchmark, all parallel and sequential workloads are multiply-add operations.
- The overall count of multiply-add operations executed on both the CPU-only and CPU-GPU systems is identical.
- GPU kernel is launched with `blockDim (7, 4, 4)` and minimum `GridDim` for max SM occupancy.
- CPU and GPU execution do not overlap, i.e., GPU execution starts only after CPU execution completes and vice versa.
- The benchmark program takes 33.12 seconds to finish on the CPU-only system, and you observe an overall speedup of 2.7824 on the CPU-GPU system.

Part A (4 points): With the information above, how many CUDA threads are assigned in each SM?

According to the datasheet, maximum number of resident threads per block is 1536

$\Rightarrow \frac{1536}{7\times4\times4} \approx 13.71 < 16$ (maximum number of blocks)

$\Rightarrow$ total threads $= 7\times4\times4\times13 = 1456$ #

Part B (6 points): Calculate the combined duration of the kernel launch overhead and memory transfer inside the CPU-GPU system.

$\frac{33.12}{3} = 11.04 \Rightarrow$ sequential work takes 11.04 sec

Overall speedup 2.7824 $\Rightarrow \frac{33.12}{2.7824} = 11.90$ sec (CPU-GPU)

speed up of GPU $\Rightarrow \frac{460}{3} = 153.33 \Rightarrow \frac{22.08}{153.33} = 0.144$ sec

(parallel workload in GPU)

$\Rightarrow$ duration $= 11.90 - 11.04 - 0.144 = 0.716$ sec #

Name: Heng An Cheng

Problem 3 (20 points): Help Howie *V*: The Plus Sign Sum

College is hard, and sometimes we all need some help staying positive. Even when everything in life seems to be against you, there are still many small happy moments around you to cherish!

Knowing the importance of finding ways to be positive, you decide to write some cuda code to help Howie implement the Plus Sign Sum, a special type of 2D matrix operation invented by Daksh that looks like a plus sign.

The operation is defined as follows: Given an `N` × `N` input matrix stored in row-major order, the Plus Sign Sum outputs another `N` × `N` matrix where each element at row *i* and column *j* is calculated by summing all element in the *i*-th row and the *j*-th column of the input matrix.

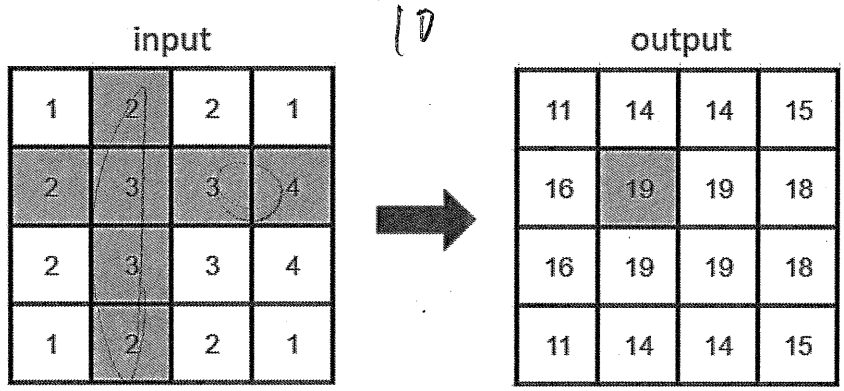


Figure 1: Example of Plus Sign Sum

Howie has left a few notes for you:

- You must employ a **tiling** strategy utilizing shared memory.
- Each thread in your kernel should compute exactly **one** element in the output matrix.
- Though the input matrix is always a square matrix, the width of the square can be any positive value.

The remainder of this page has been intentionally left blank.

Problem 3, continued:

Part HL (0 points): Before you get to coding, Howie asks you for a restaurant recommendation since he would like to try all the good restaurants in Urbana-Champaign area before graduating.

Your favorite restaurant is Subway

Part A (6 points): It is time to write the host code that launches the Plus Sign Sum kernel.

```
#define TILE_WIDTH 16 // your tile width for code in both parts A and B

// The signature of the CUDA kernel is provided for your reference.
__global__ void plus_sign_sum(float* in, float* out, int width);


// N is the width of the square of the 2D input matrix
// h_in points to input matrix in the host memory
// h_out points to the properly allocated output matrix in the host memory
void plus_sign_sum_host(float* h_in, float* h_out, int N) {
    float *d_in, *d_out;
    int size = N*N*sizeof(float);
    cudaMalloc((void **)&d_in, size);
    cudaMalloc((void **)&d_out, size);
    cudaMemcpy(d_in, h_in, size, cudaMemcpyHostToDevice);

    dim3 dimGrid(ceil(N*1.0/16), ceil(N*1.0/16), 1);
    dim3 dimBlock(16,16,1);

    plus_sign_sum<<< dimGrid, dimBlock>>>(d_in,d_out, N);

    cudaMemcpy(h_out, d_out, size, cudaMemcpyDeviceToHost);
    cudaFree(d_in);
    cudaFree(d_out);
}
```

Name: Heng An Cheng

Problem 3, continued:

Part B (14 points): Now you will help Howie write the Plus Sign Sum kernel:

```
__global__ void plus_sign_sum(float* in, float* out, int width) {
  int bx = blockIdx.x; int tx = threadIdx.x;
  int by = blockIdx.y; int ty = threadIdx.y;
  __shared__ tile[16][16];
  int row = by * blockDim.y + ty;
  int col = bx * blockDim.x + tx;
  float sum = 0;
  for(int k=0; k < width/16; k++){
      tile[ty][tx] = in[row*width + k*16 + tx];
      __syncthreads();
      for(int m=0; m<16; m++){
          sum += tile[ty][m];
          sum += tile[m][tx];
      }
      sum -= tile[ty][tx]; // calculate center twice
      __syncthreads();
  }
  out[row*width + col] = sum;
}
```

Problem 4 (40 points): Convoluted Convolutions

In this problem, we will perform 2D Tiled Convolution. We will only compute the output elements that allow the mask to fully overlap with the input (also known as "valid" padding). For example, if the input is 20×15, and the mask is 5×5, then the output will be 16×11, as illustrated below.

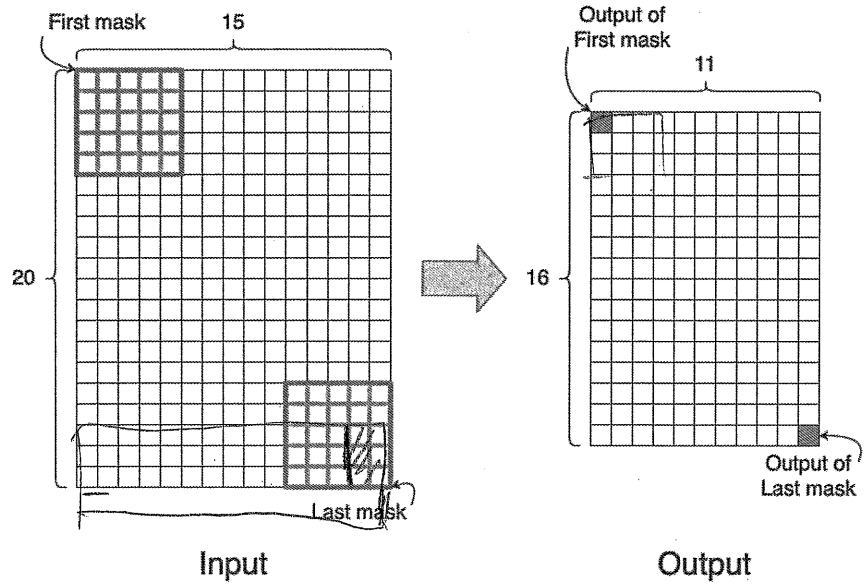


Figure 2: Example of "Valid" Convolution

In this case, we can simplify some of the calculations by always indexing from the top left corner of the mask, rather than centered in the mask. For example, with a mask width of 5, the output at element row `i` column `j` will be calculated using the input between rows `i` to `i+4` and columns `j` to `j+4` (inclusive), and using mask row and column indices between 0 to 4 (inclusive).

Our kernel will use shared tiling with a variation of Strategy 1 discussed in lecture in order to load from global memory. Additionally, each thread will not only be responsible for a single output element, but a rectangular tile of output elements with size `THREAD_TILE_HEIGHT` × `THREAD_TILE_WIDTH`.

In order to load the elements into shared memory, each thread will essentially disregard the existing structure of the block dimensions by converting its thread index into a 1-D linearized index within its block. Then, each thread will iterate through the shared memory, loading elements in a strided pattern. Each iteration, all of the threads within a block will in total load `threads_per_block` elements from the input into the shared memory. It will continue loading the next `threads_per_block` elements until all of the shared memory has been filled.

In **Part A** and **Part B**, you will be asked to complete the host and the kernel functions. When using macros, always reference the macro names or their allowed abbreviations instead of hardcoding their values. Here are the macros and their allowed abbreviations:

```
#define THREAD_TILE_HEIGHT 2   /* number of rows of output elements each
                                  thread is responsible for */
#define THREAD_TILE_WIDTH 3    /* number of columns of output elements each
                                  thread is responsible for */
#define BLOCK_HEIGHT 16        /* blockDim.y */
#define BLOCK_WIDTH 16         /* blockDim.x */
#define MASK_WIDTH 5           /* mask size (mask is a square) */
```

The allowed abbreviations can be found on the next page.

Problem 4, continued:

```
#define TTH THREAD_TILE_HEIGHT  /* number of rows of output elements each
                                   thread is responsible for */
#define TTW THREAD_TILE_WIDTH   /* number of columns of output elements each
                                   thread is responsible for */
#define BH  BLOCK_HEIGHT        /* blockDim.y */
#define BW  BLOCK_WIDTH         /* blockDim.x */
#define MW  MASK_WIDTH          /* mask size (mask is a square) */
```

Part A (4 points): Write the host code to properly declare and copy the host mask to `c_mask` in constant memory and decide `gridDim`.

```
// input_height and input_width are the height and width of the input image
// d_input points to pre-allocated device memory that has input image
// d_output points to the properly allocated output image in the device memory
// h_mask points to the mask in the host memory
void convolve_host(float* d_input, float* d_output, float* h_mask,
                   const int input_height, const int input_width) {
  __constant__ float c_mask[MW][MW];
  cudaMemcpyToSymbol(c_mask, h_mask, MW*MW*sizeof(float));
  // grid x dimension covers blocks across the width of the output
  // grid y dimension will be across the height.
  dim3 gridDim(ceil(input_width*1.0/(BW*TTW)),
               ceil(input_height*1.0/(BH*TTH)));
  dim3 blockDim(BLOCK_WIDTH, BLOCK_HEIGHT);
  convolve<<<gridDim, blockDim>>>(d_input, d_output, input_height, input_width);
}
```

Part B (18 points): Write the kernel to perform the required convolution. (The `/* (C.#) */` comments will be used in **Part C**.)

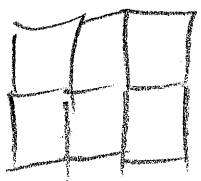
```
__global__ void convolve(float* input, float* output,
                         const int input_height, const int input_width) {
  // total output dimensions
  int output_height = input_height - MW + 1;
  int output_width = input_width - MW + 1;
  // coordinates of the output tile for this block
  int block_output_y = blockIdx.y * BH + threadIdx.y * TTH;
  int block_output_x = blockIdx.x * BW + threadIdx.x * TTW;
  // `constexpr` is specified here to declare compile-time constant
  // size of the shared tile
  constexpr int shared_tile_height = TTH * BH;
  constexpr int shared_tile_width = TTW * BW;
  __shared__ float shared_tile[shared_tile_height][shared_tile_width];
```

The rest of the kernel continues on the next page.

Name: HengAn Cheng

Problem 4, continued:

```
// load the required elements
int linearized_index = block_output_y * input_width + block_output_x;
int total_loads = linearized_index + BW * BH * TTW * TTH;
int threads_per_block = BW * BH;
for (int load_index = linearized_index;
    load_index < total_loads;                                   /* (C.1) */
    load_index += threads_per_block) {
  // the shared tile coordinates
  int relative_load_y = load_index / shared_tile_width;
  int relative_load_x = load_index % shared_tile_width;
  // the absolute input coordinates
  int load_y = load_index / input_width;
  int load_x = load_index % input_width;
  if (load_y < output_height && load_x < output_width)          /* (C.2) */
    shared_tile[relative_load_y][relative_load_x] =
      input[load_index];
  }
}
__syncthreads();
for (int element_y = 0; element_y < TTH; element_y++) {         /* (C.3) */
  for (int element_x = 0; element_x < TTW; element_x++) {       /* (C.4) */
    int output_y = block_output_y + BH * element_y;
    int output_x = block_output_x + BW * element_x;
    if (output_y < output_height && output_x < output_width) {  /* (C.5) */
      float sum = 0.0f;
      for (int mask_y = 0; mask_y < MW; mask_y++) {             /* (C.6) */
        for (int mask_x = 0; mask_x < MW; mask_x++) {           /* (C.7) */
          // coordinates of the input element in the shared tile
          int tile_y = mask_y + element_y * TTH;
          int tile_x = mask_x + element_x * TTW;
          sum += shared_tile[tile_y][tile_x] * c_mask[mask_y][mask_x];
        }
      }
      output[output_y * input_width + output_x] = sum;
    }
  }
}
```



Name: HengAn Cheng

Problem 4, continued:

Part C (8 points): Shengjie would like to know, which lines of code with comments `/* (C.#) */` may cause control divergence for certain values of `input_height`, `input_width`, `BLOCK_HEIGHT`, `BLOCK_WIDTH`, `THREAD_TILE_HEIGHT`, `THREAD_TILE_WIDTH`, and `MASK_WIDTH`? Briefly explain how control divergence would occur in each case.

① if input_width / (BLOCK_WIDTH × THREAD_TILE_WIDTH) is not integer

⇒ (C.1), (C.2), (C.4) may have control divergence.

② if input_height / (BH × TTH) is not integer:

⇒ (C.1), (C.2), (C.3) may have control divergence

③ if (input_height − MW + 1) / (BH × TTH) or (input_width − MW + 1) / (BW × TTW) is not integer.

⇒ (C.5) may have control divergence.

Part D (10 points): Answer the following questions about the convolution kernel in **Part B**. Assume N = `input_height` = `input_width`.

(D.1) For an internal block (not near any boundary), determine the amount of reads and writes to global memory **in bytes**. Provide a separate answer for reads and writes, and answer in terms of `TTH`, `TTW`, `BH`, `BW`, `MW`, and N.

Reads: (BH × TTH) × (BW × TTW) × 4

Writes: (BH × TTH) × (BW × TTW) × 4

(D.2) For an internal block, determine the number of floating point operations performed by the block. Answer in terms of `TTH`, `TTW`, `BH`, `BW`, `MW`, and N.

2 × (MW × MW) × (BH × TTH) × (BW × TTW) ※

(D.3) As we increase `THREAD_TILE_HEIGHT` × `THREAD_TILE_WIDTH`, we should expect that the performance of the kernel improves. What is one reason possible reason for this? Assume that our input size is quite large.

more elements will be reused, reducing the time to access global memory.

(D.4) Suppose that we set `MASK_WIDTH` = 3, `BLOCK_WIDTH` = 4, and `BLOCK_HEIGHT` = 48. After profiling the code, we realize that if we keep `THREAD_TILE_HEIGHT` constant and increase `THREAD_TILE_WIDTH`, we get better performance compared to keeping `THREAD_TILE_WIDTH` constant and increasing `THREAD_TILE_HEIGHT`. What is one possible reason for this?

because increasing THREAD_TILE_WIDTH may utilize the benefit of coalesceing, which makes global memory access faster.

Reference Sheet

Execution Space Specifier

`__global__` declares a function as being a kernel. A `__global__` function must have `void` return type, and cannot be a member of a class. A call to a `__global__` function is asynchronous, meaning it returns before the device has completed its execution.

`__host__` declares a function that is (a) executed on the host and (b) callable from the host only.

`__device__` declares a function that is (a) executed on the device and (b) callable from the device only.

Variable Memory Space Specifier

`__constant__` declares a variable that resides in constant memory space.

`__shared__` declares a variable that resides in the shared memory space of a thread block.

Synchronization

`void __syncthreads ( )`
Wait until all threads in the thread block have reached this point and all global and shared memory accesses made by these threads prior to `__syncthreads` are visible to all threads in the block.

`cudaError_t cudaDeviceSynchronize ( void )`
Wait for compute device to finish.

Built-in Variables

`gridDim` is of type `dim3` and contains the dimensions of the grid. Access the number of blocks in the grid in each dimension using `gridDim.x`, `gridDim.y`, and `gridDim.z`.

`blockDim` is of type `dim3` and contains the dimensions of the block. Access the number of threads in the block in each dimension using `blockDim.x`, `blockDim.y`, and `blockDim.z`.

`threadIdx` is of type `uint3` and contains the thread index within the block. Access the thread index in each dimension using `threadIdx.x`, `threadIdx.y`, and `threadIdx.z`.

Kernel Configuration and Kernel launch

```
dim3 dimGrid(x_size, y_size, z_size);
dim3 dimBlock(x_size, y_size, z_size);
```

When defining a variable of type `dim3`, any component left unspecified is initialized to 1.

```
kernelName<<<dimGrid, dimBlock>>> (input parameters)
```

The number of threads per block and the number of blocks per grid specified in the `<<<...>>>` syntax can be of type `int` or `dim3`.

Memory Management

`cudaError_t cudaMalloc ( void** devPtr, size_t size )`
Allocates `size` bytes of linear memory on the device and returns in `*devPtr` a pointer to the allocated memory.

```
cudaError_t cudaMemcpy ( void* dst, const void* src,
                         size_t count, cudaMemcpyKind kind )
```
Copies `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`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, or `cudaMemcpyDeviceToDevice`.

```
cudaError_t cudaMemcpyToSymbol ( const void* symbol, const void* src,
                                 size_t count, size_t offset = 0,
                                 cudaMemcpyKind kind = cudaMemcpyHostToDevice )
```
Copies `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `offset` bytes from the start of symbol `symbol`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToDevice`.

`cudaError_t cudaFree ( void* devPtr )`
Frees the memory space pointed to by `devPtr` on the device.

Miscellaneous Functions

`ceil(x)` returns the smallest integer value greater than or equal to `x` (as a floating-point value).

`floor(x)` returns the largest integer value less than or equal to `x` (as a floating-point value).

`sqrtf(x)` returns the square root of the value `x`, where `x` is a non-negative floating-point number. The result is also a floating-point value.

`sizeof(type)` yields the size of `type` in bytes.

NVIDIA A40 GPU Datasheet:

- **GPU Architecture:** NVIDIA Ampere architecture
- **GPU Memory:** 48 GB GDDR6 with ECC (42 GB excluding ECC)
- **Memory Bandwidth:** 696 GB/s
- **Streaming Multiprocessors:** 84
- **RT Cores (2nd Gen):** 84
- **Tensor Cores (3rd Gen):** 336
- **GPU Clocks:**
  - Base: 1305 MHz
  - Boost: 1740 MHz
- **Compute Capability:** 8.6
- **Max Power Consumption:** 300 W
- **Compute APIs:** CUDA, DirectCompute, OpenCL™, OpenACC®
- **Graphics APIs:** DirectX 12.07, Shader Model 5.17, OpenGL 4.68, Vulkan 1.18

Name: HengAn Cheng

NVIDIA Technical Specifications per Compute Capability:

	Compute Capability													
Technical Specifications	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.2	7.5	8.0	8.6	8.7	8.9	9.0
Maximum number of resident grids per device (Concurrent Kernel Execution)	32		16	128	32	16	128	16	128					
Maximum dimensionality of grid of thread blocks	3													
Maximum x -dimension of a grid of thread blocks [thread blocks]	2^{31}-1													
Maximum y- or z-dimension of a grid of thread blocks	65535													
Maximum dimensionality of thread block	3													
Maximum x- or y-dimensionality of a block	1024													
Maximum z-dimension of a block	64													
Maximum number of threads per block	1024													
Warp size	32													
Maximum number of resident blocks per SM	32								16	32	16		24	32
Maximum number of resident warps per SM	64								32	64	48			64
Maximum number of resident threads per SM	2048								1024	2048	1536			2048
Number of 32-bit registers per SM	64 K													
Maximum number of 32-bit registers per thread block	64 K		32 K	64 K		32 K	64 K							
Maximum number of 32-bit registers per thread	255													
Maximum amount of shared memory per SM	64 KB	96 KB	64 KB		96 KB	64 KB	96 KB		64 KB	164 KB	100 KB	164 KB	100 KB	228 KB
Maximum amount of shared memory per thread block [32]	48 KB						96 KB	96 KB	64 KB	163 KB	99 KB	163 KB	99 KB	227 KB
Maximum amount of local memory per thread	512 KB													
Constant memory size	64 KB													
Cache working set per SM for constant memory	8 KB			4 KB	8 KB									
Cache working set per SM for texture memory	Between 12 KB and 48 KB			Between 24 KB and 48 KB			32 ~ 128 KB		32 or 64 KB	28 KB ~ 192 KB	28 KB ~ 128 KB	28 KB ~ 192 KB	28 KB ~ 128 KB	28 KB ~ 256 KB
Maximum width for a 1D texture object using a CUDA array	65536			131072										
Maximum width for a 1D texture object using linear memory	2^{27}			2^{28}	2^{27}		2^{28}	2^{27}	2^{28}					