GPU CUDA

High-Performance Computing

Summer 2021 at GIST

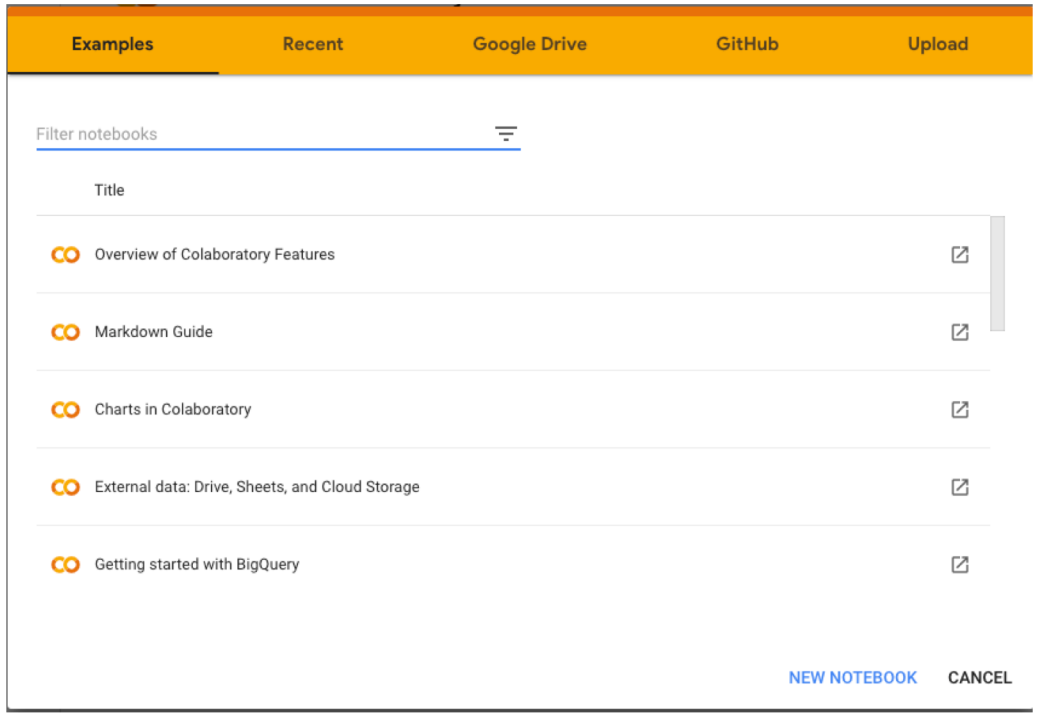
Tae-Hyuk (Ted) Ahn
Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University

SAINT LOUIS
UNIVERSITY
EST. 1818

Google Colab

Colaboratory, or "Colab" for short, allows you to write and execute Python in your browser, with

- Zero configuration required
- Free access to GPUs
- Easy sharing
- Built on top of Jupyter Notebook
- Google Colab notebooks are stored on the drive



Google Colab Let's Begin!

Create a Colab Notebook

- 1. Open Google Colab.
- 2. Click on 'New Notebook'.

OR

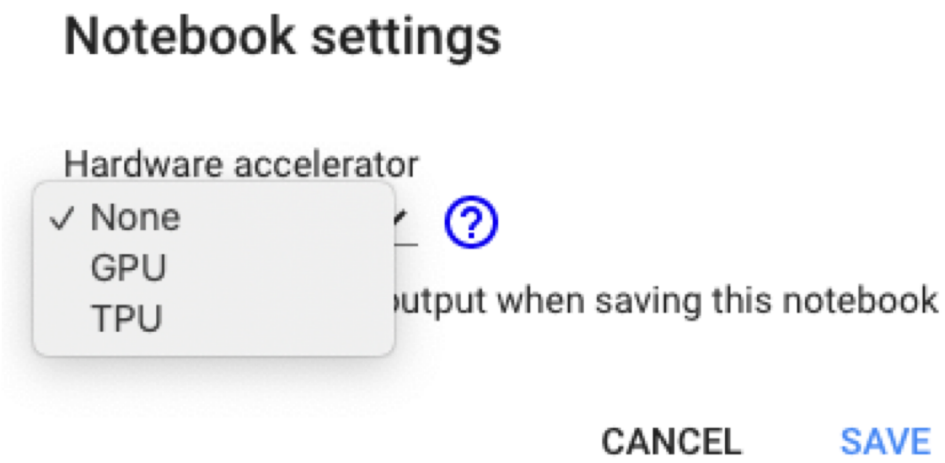
1. Open Google Drive.
2. Create a new folder for the project.
3. Click on 'New' > 'More' > 'Colaboratory'.

I have below default Colab Notebooks directory. Check it on your Google Drive and create HPC_Lab

Setting GPU Accelerator

The default hardware of Google Colab is CPU or it can be GPU.

1. Click on ‘Edit’ > ‘Notebook Settings’ > ‘Hardware Accelerator’ > ‘GPU’.

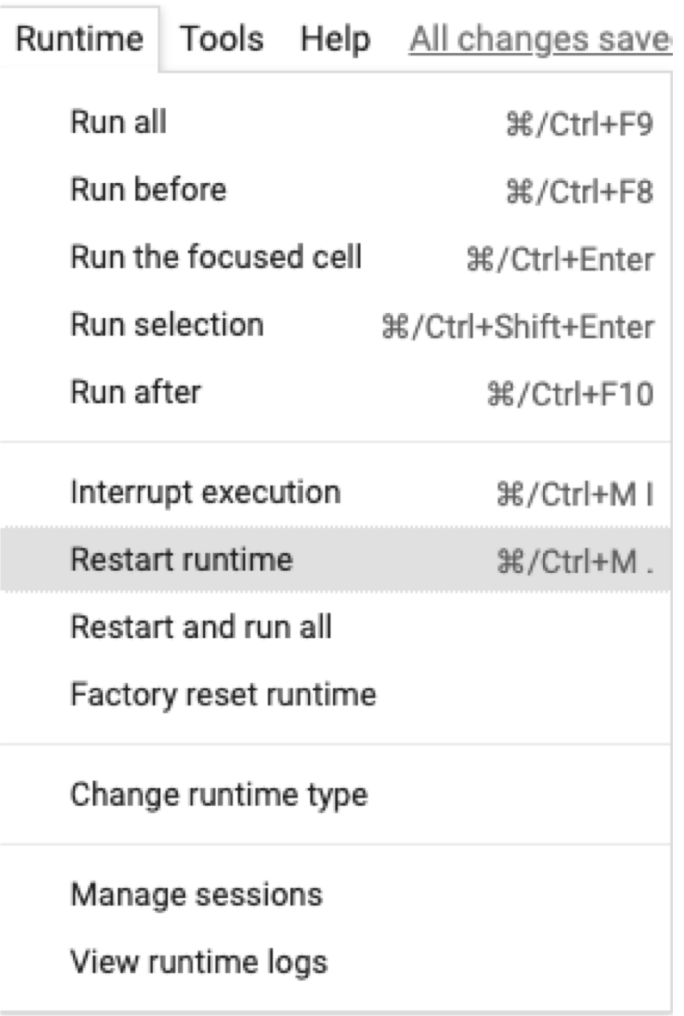


Running a Cell

- Make sure the runtime is connected. The notebook shows a green check and 'Connected' on the top right corner.
- There are various runtime options in 'Runtime'.

OR

- To run the current cell, press SHIFT + ENTER.



Lab1: Running a Cell

- Bash commands can be run by prefixing the command with '!'.

```
[1]  !ls

     sample_data
```

```
     !ls sample_data/

     anscombe.json                    mnist_test.csv
     california_housing_test.csv      mnist_train_small.csv
     california_housing_train.csv     README.md
```

Lab1: Connect to your Google Drive

```
1  from google.colab import drive
2  drive.mount('/content/drive/')
```

- When you run the code above, you should see a result like this:

- **Click** the link, **copy** verification code and **paste** it to text box. Then, it will be

```
[5]  from google.colab import drive
     drive.mount('/content/drive/')

     Mounted at /content/drive/
```

Lab1: Connect to your Google Drive

```
[6]  !ls

     drive  sample_data
```

```
[8]  !ls drive/MyDrive/

      22.160P_A_Tilt_FPS26_Live5
      23.160M_S_Automotive_Live5
     'BCB-5250 Spring 2021 Survey.gform'
      Career
     'Colab Notebooks'
      ColabTest
      Commitment
     'CSCI-2100 5002 Final Exam Schedule Voting.gform'
     'CSCI-4850 5850 HPC Spring 2021 Survey.gform'
      EBSCO
      Kirmizis_2007_Sir2_wildtype_histone_H3.merge.bw
      PersonalDocuments
      Research
      Software
      Teaching
     '~$ummary.docx'
     'Worksheet (1).gform'
      Worksheet.gform
```

Lab1: Check CPU and RAM specifications

```
!cat /proc/cpuinfo
!cat /proc/meminfo
```

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 63
model name      : Intel(R) Xeon(R) CPU @ 2.30GHz
stepping        : 0
microcode       : 0x1
cpu MHz         : 2299.998
cache size      : 46080 KB
physical id     : 0
siblings        : 2
core id         : 0
cpu cores       : 1
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8
bugs            : cpu_meltdown spectre_v1 spectre_v2
bogomips        : 4599.99
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:

processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 63
```

```
MemTotal:       13333564 kB
MemFree:        10529740 kB
MemAvailable:   12432924 kB
Buffers:           85436 kB
Cached:          1958172 kB
SwapCached:            0 kB
Active:          1025112 kB
Inactive:        1462468 kB
Active(anon):     409852 kB
Inactive(anon):      360 kB
Active(file):     615260 kB
Inactive(file):  1462108 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:             0 kB
SwapFree:              0 kB
Dirty:               372 kB
Writeback:             0 kB
AnonPages:        443960 kB
Mapped:           245996 kB
Shmem:              1012 kB
```

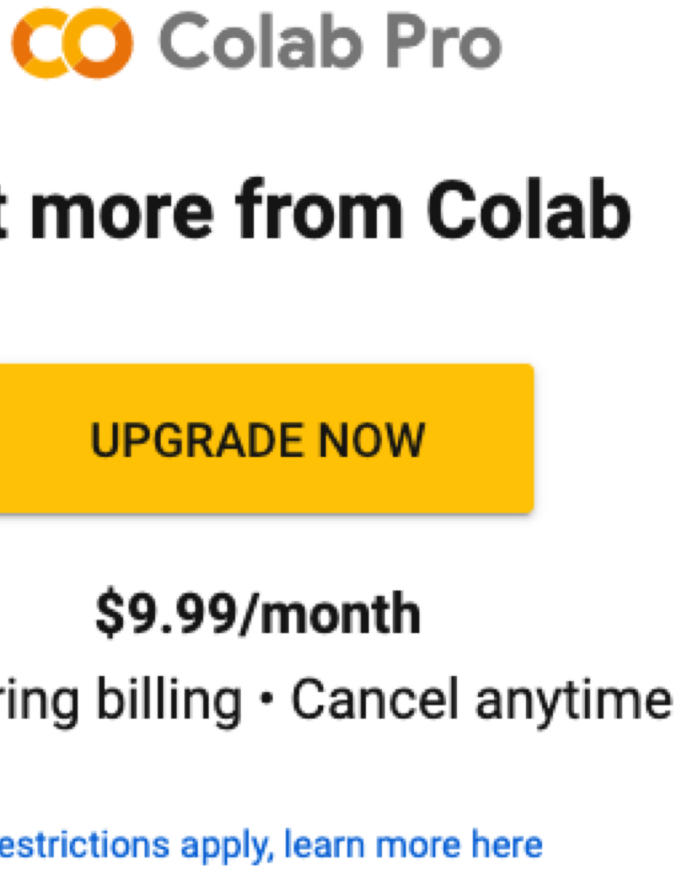
Lab1: Check GPU

```
!nvidia-smi
```

```
Thu Apr 29 13:36:22 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 465.19.01    Driver Version: 460.32.03    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  Tesla P4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   39C    P0    23W /  75W |    199MiB /  7611MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
+-----------------------------------------------------------------------------+
```

- https://developer.nvidia.com/blog/nvidia-tesla-p4-gpus-available-now-on-the-google-cloud-platform/

Need more powerful GPUs on Google Colab?



Faster GPUs

Priority access to faster GPUs and TPUs means you spend less time waiting while code is running. Learn more

Longer runtimes

Longer running notebooks and fewer idle timeouts mean you disconnect less often. Learn more

More memory

More RAM and more disk means more room for your data. Learn more

What kinds of GPUs are available in Colab Pro?

With Colab Pro you get priority access to our fastest GPUs. For example, you may get access to T4 and P100 GPUs at times when non-subscribers get K80s. You also get priority access to TPUs. There are still usage limits in Colab Pro, though, and the types of GPUs and TPUs available in Colab Pro may vary over time.

In the free version of Colab there is very limited access to faster GPUs and to TPUs, and usage limits are much lower than they are in Colab Pro.

Lab2: Running CUDA in Google Colab

- **CUDA** comes already pre-installed in **Google Colab**.

```
!nvidia-smi
```

```
Thu Apr 29 14:38:42 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 465.19.01    Driver Version: 460.32.03    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  Tesla K80           Off  | 00000000:00:04.0 Off |                    0 |
| N/A   73C    P8    32W / 149W |      0MiB / 11441MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

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

Lab2: Running CUDA in Google Colab

- **CUDA** comes already pre-installed in **Google Colab**.

```
[2]  !nvcc --version

     nvcc: NVIDIA (R) Cuda compiler driver
     Copyright (c) 2005-2020 NVIDIA Corporation
     Built on Wed_Jul_22_19:09:09_PDT_2020
     Cuda compilation tools, release 11.0, V11.0.221
     Build cuda_11.0_bu.TC445_37.28845127_0
```

- As it can be seen, the installation regards CUDA 11.0.

Lab2: Running CUDA in Google Colab

- To enable CUDA **programming** and **execution** directly under Google Colab, you need to install the nvcc4jupyter plugin as

  !pip install git+git://github.com/andreinechaev/nvcc4jupyter.git

```
!pip install git+git://github.com/andreinechaev/nvcc4jupyter.git

Collecting git+git://github.com/andreinechaev/nvcc4jupyter.git
  Cloning git://github.com/andreinechaev/nvcc4jupyter.git to /tmp/pip-req-build-ugiskyks
  Running command git clone -q git://github.com/andreinechaev/nvcc4jupyter.git /tmp/pip-req-
Building wheels for collected packages: NVCCPlugin
  Building wheel for NVCCPlugin (setup.py) ... done
  Created wheel for NVCCPlugin: filename=NVCCPlugin-0.0.2-cp37-none-any.whl size=4307 sha256
  Stored in directory: /tmp/pip-ephem-wheel-cache-lrani5l9/wheels/10/c2/05/ca241da37bff77d60
Successfully built NVCCPlugin
Installing collected packages: NVCCPlugin
Successfully installed NVCCPlugin-0.0.2
```

Lab2: Running CUDA in Google Colab

- After that, you should load the plugin as

  %load_ext nvcc_plugin

```
%load_ext nvcc_plugin

created output directory at /content/src
Out bin /content/result.out
```

Now it is ready to run CUDA code!!

Hello World Again!

- Standard C code!

```c
#include<stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

```
aigrad1@dgx-v100-n1:/mnt/aigrad1/Lab/CUDA$ gcc hello.c -o hello
aigrad1@dgx-v100-n1:/mnt/aigrad1/Lab/CUDA$ ./hello
Hello World!
```

Hello World using CUDA compiler!

- File extension: .cu
  - hello_cuda.cu

```
#include<stdio.h>

__global__ void mykernel() {
}

int main()
{
    mykernel<<<1,1>>>();
    printf("Hello World CUDA from GPU\n");
    return 0;
}
```

- Compile
  - nvcc

```
aigrad1@dgx-v100-n1:/mnt/aigrad1/Lab/CUDA$ nvcc hello_cuda.cu -o hello_cuda
aigrad1@dgx-v100-n1:/mnt/aigrad1/Lab/CUDA$ ./hello_cuda
Hello World CUDA from GPU
```

Lab2: Running CUDA in Google Colab

- Write and run CUDA code by adding below prefix

  %%cu

- Compilation and execution occurs when pressing the play button to run a code cell.

```
%%cu

#include<stdio.h>

__global__ void mykernel() {
}

int main()
{
    mykernel<<<1,1>>>();
    printf("Hello World CUDA from GPU!\n");
    return 0;
}
```

```
Hello World CUDA from GPU!
```

Hello World! with Device Code

```
__global__ void mykernel() {
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code

- `nvcc` separates source code into host and device components
  - Device functions (e.g. **`mykernel()`**) processed by NVIDIA compiler
  - Host functions (e.g. **`main()`**) processed by standard host compiler
    - **`gcc`**, **`cl.exe`**

CUDA C/C++ Keywords

- Keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host

- Keyword `__host__` indicates a function that:
  - Runs on the host

- Keyword `__device__` indicates a function that:
  - Runs on the device
  - Is called from device

Hello World! with Device Code

```
mykernel<<<1,1>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
  - Also called a "kernel launch"
  - We'll return to the parameters (1,1) in a moment

- That's all that is required to execute a function on the GPU!

Lab3: Vector addition using CUDA

vector_add.c

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

#define N 10000000
#define MAX_ERR 1e-6

void vector_add(float *out, float *a, float *b, int n) {
    for (int i=0; i<n; i++) {
        out[i] = a[i] + b[i];
    }
}

```

```c
int main()
{
    float *a, *b, *out;

    // allocate memory
    a   = (float*)malloc(sizeof(float) * N);
    b   = (float*)malloc(sizeof(float) * N);
    out = (float*)malloc(sizeof(float) * N);

    // initialize array
    for (int i=0; i<N; i++) {
        a[i] = 1.0f;
        b[i] = 2.0f;
    }

    // call function
    vector_add(out, a, b, N);

    // verification
    for (int i=0; i<N; i++) {
        assert(fabs(out[i] - a[i] - b[i]) < MAX_ERR);
    }

    printf("out[0] = %f\n", out[0]);
    printf("PASSED\n");

    // deallocate host memory
    free(a);
    free(b);
    free(out);
}
```

Converting vector addition to CUDA

- Copy vector_add.c to vector_add_cuda.cu
- Convert vector_add() to GPU kernel

```
 8 __global__ void vector_add(float *out, float *a, float *b, int n) {
 9     for (int i=0; i<n; i++) {
10         out[i] = a[i] + b[i];
11     }
12 }
```

- Change vector_add() call in main() to kernel call

```
29     // call function
30     vector_add<<<1,1>>>(out, a, b, N);
```

- Compile and run the program

```
aigrad1@dgx-v100-n1:/mnt/aigrad1/Lab/CUDA$ nvcc vector_add_cuda.cu -o vector_add_cuda
aigrad1@dgx-v100-n1:/mnt/aigrad1/Lab/CUDA$ ./vector_add_cuda
vector_add_cuda: vector_add_cuda.cu:34: int main(): Assertion `fabs(out[i] - a[i] - b[i]) < MA
X_ERR' failed.
Aborted (core dumped)
```

Converting vector addition to CUDA

- You will notice that the program does not work correctly.
- The reason is CPU and GPUs are separate entities.
- Both have their own memory space. CPU cannot directly access GPU memory, and vice versa.
- In CUDA terminology, CPU memory is called host memory and GPU memory is called device memory. Pointers to CPU and GPU memory are called host pointer and device pointer, respectively.

Converting vector addition to CUDA

For data to be accessible by GPU, it must be presented in the device memory. CUDA provides APIs for allocating device memory and data transfer between host and device memory. Following is the common workflow of CUDA programs.

1. Allocate host memory and initialized host data
2. Allocate device memory
3. Transfer input data from host to device memory
4. Execute kernels
5. Transfer output from device memory to host

So far, we have done step 1 and 4. We will add step 2, 3, and 5 to our vector addition program and finish this exercise.

Device memory management

- CUDA provides several functions for allocating device memory. The most common ones are `cudaMalloc()` and `cudaFree()`. The syntax for both functions are as follow

```
cudaMalloc(void **devPtr, size_t count);
cadaFree(void *devPtr);
```

- `cudaMalloc()` allocates memory of size count in the device memory and updates the device pointer devPtr to the allocated memory.
- `cudaFree()` deallocates a region of the device memory where the device pointer devPtr points to.
- They are comparable to `malloc()` and `free()` in C, respectively

Memory transfer

- Transfering date between host and device memory can be done through `cudaMemcpy` function, which is similar to memcpy in C. The syntax of cudaMemcpy is as follow

```
cudaMemcpy(void *dst, void *src, size_t count, cudaMemcpyKind kind)
```

- The function copy a memory of size count from src to dst.
- kind indicates the direction. For typical usage, the value of kind is either `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost`.

Completing vector addition

- Copy vector_add_cuda.cu to vector_add_correct_cuda.cu
- Allocate device memory for array a, b, and out.

```
16      float *a, *b, *out;
17      float *d_a, *d_b, *d_out;
```

```
19      // allocate memory
20      a   = (float*)malloc(sizeof(float) * N);
21      b   = (float*)malloc(sizeof(float) * N);
22      out = (float*)malloc(sizeof(float) * N);
23
24      // allocate device memory
25      cudaMalloc((void**)&d_a, sizeof(float) * N);
26      cudaMalloc((void**)&d_b, sizeof(float) * N);
27      cudaMalloc((void**)&d_out, sizeof(float) * N);
```

Completing vector addition

- Transfer data from host to device memory.
- Be careful to send d_out, d_a, d_b to the kernel function.

```
35      // transfer data from host to device memory
36      cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);
37      cudaMemcpy(d_b, b, sizeof(float) * N, cudaMemcpyHostToDevice);
38
39      // call function
40      vector_add<<<1,1>>>(d_out, d_a, d_b, N);
41
```

Completing vector addition

- Transfer output data from device to host memory.

```
42      // transfer data from device to host
43      cudaMemcpy(out, d_out, sizeof(float) * N, cudaMemcpyDeviceToHost);
44
```

- Deallocate device memory

```
58      // deallocate device memory
59      cudaFree(d_a);
60      cudaFree(d_b);
61      cudaFree(d_out);
```

Lab3: Completing vector addition on Google Colab

```
%%cu

#include <math.h>
#include <stdio.h>
#include <assert.h>

#define N 10000000
#define MAX_ERR 1e-6

__global__ void vector_add(float *out, float *a, float *b, int n) {
    for (int i=0; i<n; i++) {
        out[i] = a[i] + b[i];
    }
}

int main()
{
    float *a, *b, *out;
    float *d_a, *d_b, *d_out;

    // allocate memory
    a   = (float*)malloc(sizeof(float) * N);
    b   = (float*)malloc(sizeof(float) * N);
    out = (float*)malloc(sizeof(float) * N);

    // allocate device memory
    cudaMalloc((void**)&d_a, sizeof(float) * N);
    cudaMalloc((void**)&d_b, sizeof(float) * N);
    cudaMalloc((void**)&d_out, sizeof(float) * N);

    // initialize array
    for (int i=0; i<N; i++) {
        a[i] = 1.0f;
        b[i] = 2.0f;
    }

    // transfer data from host to device memory
    cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, sizeof(float) * N, cudaMemcpyHostToDevice);

    // call function
    vector_add<<<1,1>>>(d_out, d_a, d_b, N);

    // transfer data from device to host
    cudaMemcpy(out, d_out, sizeof(float) * N, cudaMemcpyDeviceToHost);

    // verification
    for (int i=0; i<N; i++) {
        assert(fabs(out[i] - a[i] - b[i]) < MAX_ERR);
    }

    printf("out[0] = %f\n", out[0]);
    printf("PASSED\n");

    // deallocate host memory
    free(a);
    free(b);
    free(out);

    // deallocate device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_out);
}
```

```
out[0] = 3.000000
PASSED
```

Going Parallel

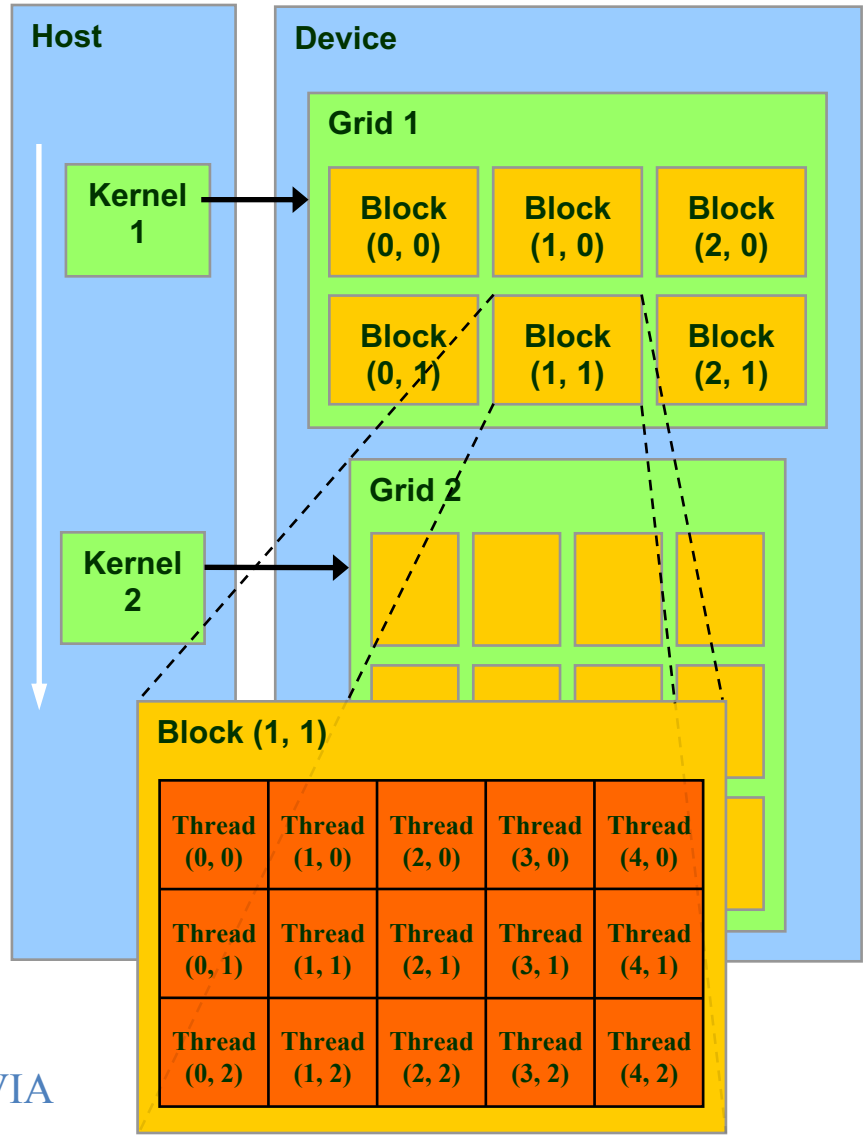
- CUDA use a kernel execution configuration <<<...>>> to tell CUDA runtime how many threads to launch on GPU. CUDA organizes threads into a group called "thread block". Kernel can launch multiple thread blocks, organized into a "grid" structure.

```
<<< M , T >>>
```

- Which indicate that a kernel launches with a grid of *M* thread blocks.
- Each thread block has *T* parallel threads.

Thread Batching: Grids and Blocks

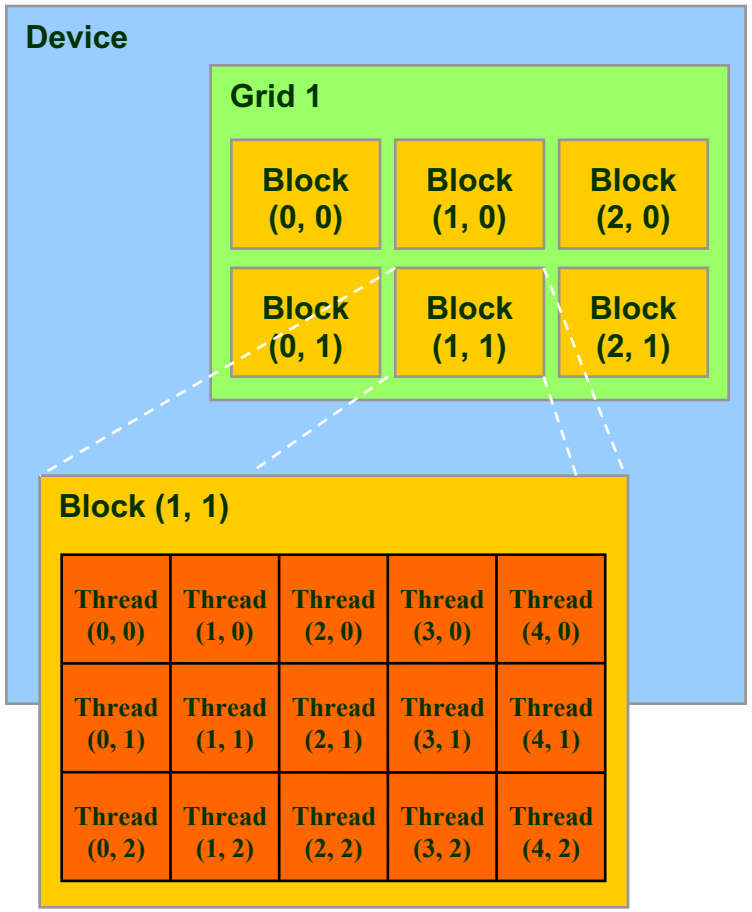
- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate



Courtesy: NDVIA

Block and Thread IDs

- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D, 2D, or 3D (`blockIdx.{x,y,z}`)
  - Thread ID: 1D, 2D, or 3D (`threadIdx.{x,y,z}`)
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Courtesy: NDVIA

Built-in Variables for Block and Thread

Built-in variables:

- `threadIdx.{x,y,z}` – thread ID within a block
- `blockIdx.{x,y,z}` – block ID within a grid
- `blockDim.{x,y,z}` – number of threads within a block
- `gridDim.{x,y,z}` – number of blocks within a grid

`kernel<<<nBlocks,nThreads>>>(args)`

Invokes a parallel kernel function on a grid of `nBlocks` where each block instantiates `nThreads` concurrent threads

Parallelizing vector addition using multithread

```
vector_add <<< 1 , 256 >>> (d_out, d_a, d_b, N);
```

- threadIdx.x contains the index of the thread within the block
- blockDim.x contains the size of thread block (number of threads in the thread block).
- For the vector_add() configuration, the value of threadIdx.x ranges from 0 to 255 and the value of blockDim.x is 256.
- Then, what is the maximum number of threads per block???

deviceQuery

- Find and run

  deviceQuery

  under ~/NVIDIA_CUDA-11.X_Samples.

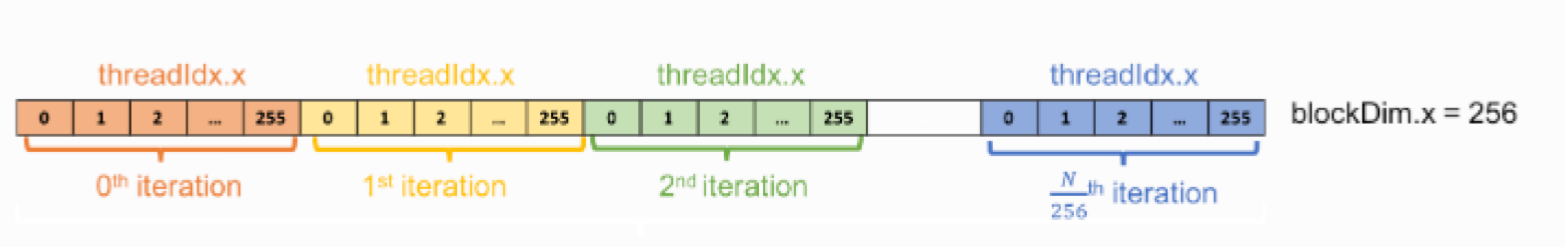
- Max number of threads per block is 1024!

```
aigrad1@dgx-v100-n1:/mnt/aigrad1/Lab/CUDA$ /usr/local/cuda/samples/bin/x86_64/linux/release/deviceQuery
/usr/local/cuda/samples/bin/x86_64/linux/release/deviceQuery Starting...

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

Detected 1 CUDA Capable device(s)

Device 0: "Tesla V100-SXM2-32GB"
  CUDA Driver Version / Runtime Version          10.1 / 10.1
  CUDA Capability Major/Minor version number:    7.0
  Total amount of global memory:                 32480 MBytes (34058272768 bytes)
  (80) Multiprocessors, ( 64) CUDA Cores/MP:     5120 CUDA Cores
  GPU Max Clock rate:                            1530 MHz (1.53 GHz)
  Memory Clock rate:                             877 Mhz
  Memory Bus Width:                              4096-bit
  L2 Cache Size:                                 6291456 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 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 6 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:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 133 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 10.1, CUDA Runtime Version = 10.1, NumDevs = 1
Result = PASS
```

Parallelizing vector addition using multithread

```
vector_add <<< 1 , 256 >>> (d_out, d_a, d_b, N);
```

Parallelizing vector addition using multithread



- For the k-th thread, the loop starts from k-th element and iterates through the array with a loop stride of 256.
- For example, in the 0-th iteration, the k-th thread computes the addition of k-th element.
- In the next iteration, the k-th thread computes the addition of (k+256)-th element, and so on.

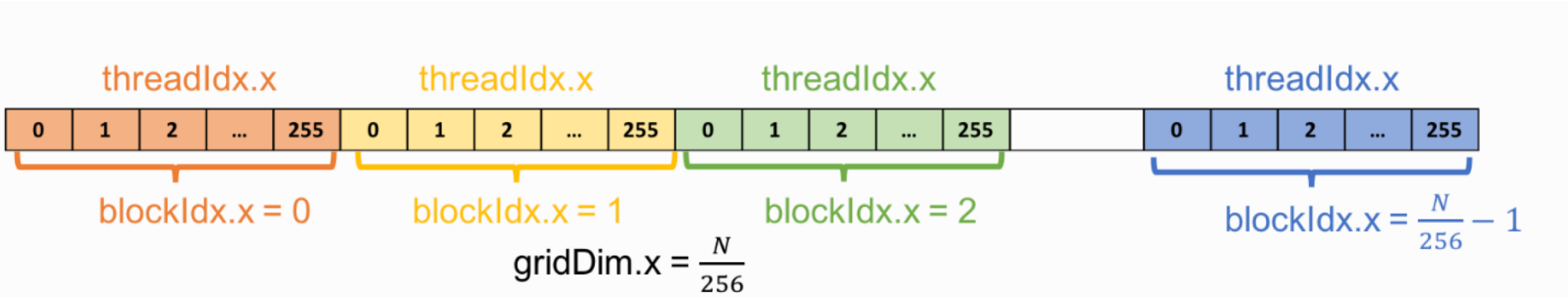
vector_add_thread.cu: You complete this using Colab

```
 8 __global__ void vector_add(float *out, float *a, float *b, int n) {
 9     int index = threadIdx.x;
10     int stride = blockDim.x;
11
12     for (int i=index; i<n; i+=stride) {
13         out[i] = a[i] + b[i];
14     }
15 }
```

```
42     // call function
43     vector_add<<<1,256>>>(d_out, d_a, d_b, N);
44
```

```
aigrad1@dgx-v100-n1:/mnt/aigrad1/Lab/CUDA$ ./vector_add_thread
out[0] = 3.000000
PASSED
```

Adding more threads blocks

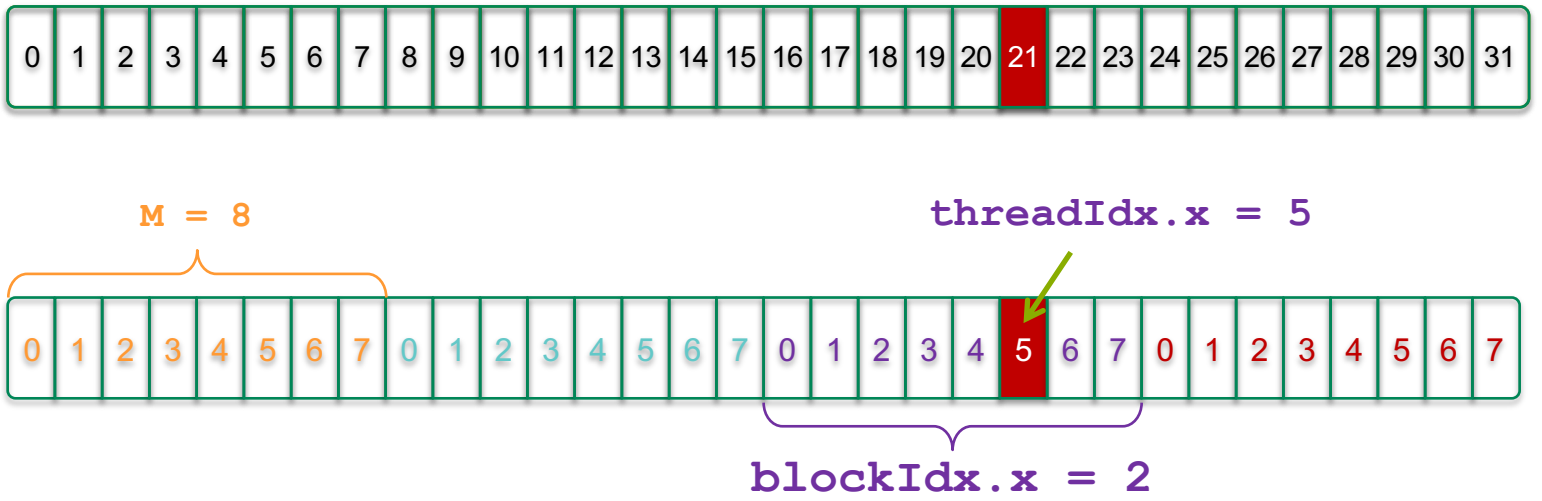


- We will use two of them: `blockIdx.x` and `gridDim.x`.
- `blockIdx.x` contains the index of the block with in the grid
- `gridDim.x` contains the size of the grid
- With 256 threads per thread block, we need at least N/256 thread blocks to have a total of Nthreads. To assign a thread to a specific element, we need to know a unique index for each thread. Such index can be computed as follow

```
int tid = blockIdx.x * M + threadIdx.x;
```

Indexing Arrays: Example

- Which thread will operate on the red element?




```
int index = threadIdx.x + blockIdx.x * M;
          =      5      +     2       * 8;
          = 21;
```

Vector Addition with Blocks and Threads

```
#define N 2560
#define MAX_ERR 1e-6

__global__ void vector_add(float *out, float *a, float *b, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    out[index] = a[index] + b[index];
}
```

```
// call function
vector_add<<<N/256, 256>>>(d_out, d_a, d_b, N);
```