



# Hands-on Session – Part I



# Using CUDA in Google Colab

- Colab is a free Jupyter notebook environment on the Google cloud.
- We will leverage the GPU resource provided on Google cloud service.
- To start, you need a Google account.
- Upload *summer-school-gpu/Nivida/cuda\_Colab\_ex1.ipynb* to your google drive and double click to launch
  - Ensure runtime connected to GPU (see the next slide)

The image shows two parts of the Google ecosystem. The top part is a Google Drive interface with a sidebar on the left containing 'New', 'My Drive', 'Computers', 'Shared with me', 'Recent', and 'Starred'. The main area shows 'My Drive' with a file named 'cuda\_Colab\_ex1.ipynb'. The bottom part is a Google Colab notebook titled 'cuda\_Colab\_ex1.ipynb'. The top bar includes 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', 'Help', and 'Last saved at 2:43 PM'. On the right, there are buttons for 'Comment', 'Share', and a 'Connect GPU' button which is highlighted with a red rectangle. The notebook content shows a section titled 'CUDA' with instructions: 'Below is an example that runs native CUDA code.' followed by a numbered list: 1. We investigate the CUDA version, drivers and the available GPU with nvidia-smi and nvcc-version; 2. We use the IPython magic command %%writefile filename to save a \*.cu program; 3. We then compile and run the \*.cu program with nvcc. At the bottom, a code cell is visible with the commands 'nvcc --version' and 'nvidia-smi'.



# Use CUDA in Google Colab

## Setting up the environment

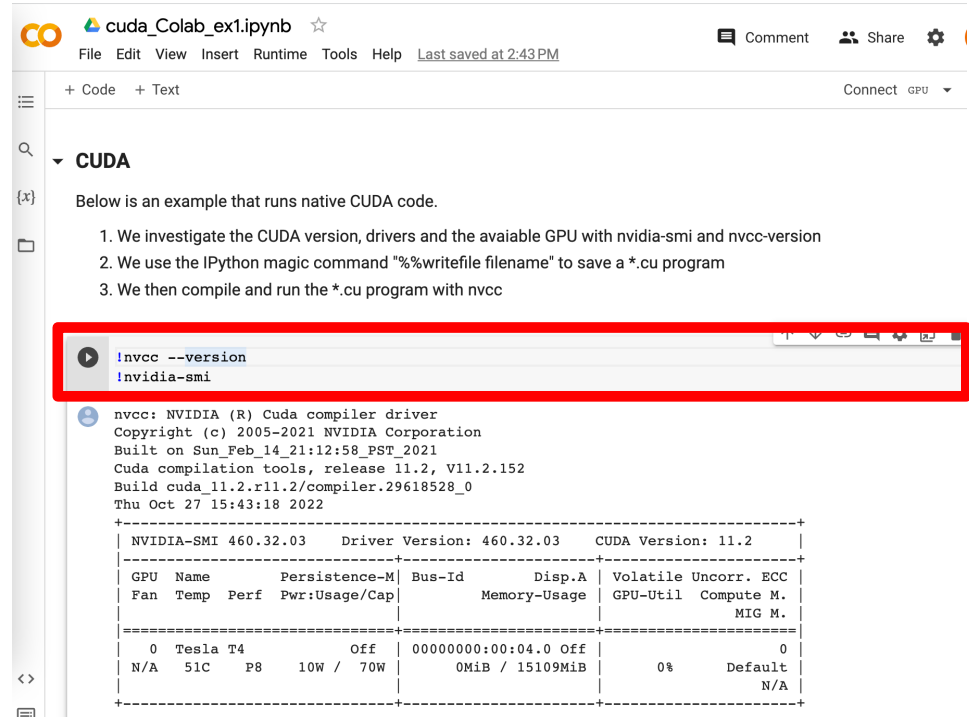
**Important note:** you need to change your runtime to use

The screenshot shows the Google Colab interface for a notebook titled 'CUDA Colab bandwidthtest.ipynb'. The 'Runtime' menu is open, displaying various options. The option 'Change runtime type' is highlighted. The notebook content shows a terminal window with the command `!ls /usr/local/cuda` and its output, which includes paths like `/usr/local/cuda/bin`, `/usr/local/cuda/compat`, `/usr/local/cuda/compute-sanitizer`, and `/usr/local/cuda/DOCS`.

The screenshot shows the 'Notebook settings' dialog box. The 'GPU' option is selected under the 'Accelerator' section. Below this, there is a checkbox for 'Want access to premium GPUs?' with a link to 'Purchase additional compute units here.' and another checkbox for 'Omit code cell output when saving this notebook'.

# Query Nvidia GPU environment

Question:  
Which GPU  
architecture and  
CUDA version are  
you running on?



The screenshot shows a Jupyter Notebook titled 'cuda\_Colab\_ex1.ipynb'. The interface includes a menu bar (File, Edit, View, Insert, Runtime, Tools, Help) and a toolbar with icons for code, text, and execution. The notebook content is under a 'CUDA' section, containing a list of three steps: 1. Investigate CUDA version, drivers, and available GPU with 'nvidia-smi' and 'nvcc-version'. 2. Use the IPython magic command '%writefile filename' to save a \*.cu program. 3. Compile and run the \*.cu program with 'nvcc'. Below the text, a code cell is highlighted with a red box, containing the commands: `!nvcc --version` and `!nvidia-smi`. The output of these commands is displayed below the code cell. The 'nvcc' output shows the NVIDIA (R) Cuda compiler driver version 11.2. The 'nvidia-smi' output is a table showing GPU information for a Tesla T4.

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2021 NVIDIA Corporation
Built on Sun_Feb_14_21:12:58_PST_2021
Cuda compilation tools, release 11.2, V11.2.152
Build cuda_11.2.r11.2/compiler.29618528_0
Thu Oct 27 15:43:18 2022
```

NVIDIA-SMI 460.32.03 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 T4	Off	00000000:00:04:0	Off	0	0			
N/A	51C	P8	10W / 70W	0MiB / 15109MiB	0%	Default			
					N/A	N/A			

## Write, Compile, Run a simple CUDA code

- The second section in *cuda\_Colab\_ex1.ipynb* writes a native CUDA code called 'vectorAdd.cu'.
- Use the IPython magic command "%writefile filename" to save a \*.cu program.
- Running the following code block should output 'writing vectorAdd.cu'

Next, we write a native CUDA code and save it as 'vectorAdd.cu'

```

❶ //writefile vectorAdd.cu
#include <stdio.h>

#include <stdlib.h>

__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}

int main() {
    int a, b, c;
    // host copies of variables a, b & c
    int *d_a, *d_b, *d_c;
    // device copies of variables a, b & c
    int size = sizeof(int);
    // Allocate space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);
    // Setup input values
    c = 0;
    a = 3;
    b = 5;
    // Copy inputs to device
    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);
    // Launch add() kernel on GPU
    add<<<1,1>>>>(d_a, d_b, d_c);
    // Copy result back to host
    cudaError err = cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
    if(err!=cudaSuccess) {
        printf("CUDA error copying to Host: %s\n", cudaGetErrorString(err));
    }
    printf("result is %d\n",c);
    // Cleanup
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    return 0;
}

```



# Write, Compile, Run a simple CUDA code

We compile the saved cuda code using nvcc compiler

```
!nvcc vectorAdd.cu -o vectorAdd
!ls
```

---

```
a.out sample_data vectorAdd vectorAdd.cu
```

Finally, we execute the binary of the compiled code

```
!./vectorAdd
```

---

```
result is 8
```

## Question:

Do we need to add  
`cudaDeviceSynchronize()`  
code? Why or Why not?



# Timing with CPU timer

A CPU timer can be created by using *gettimeofday()* system call to get the system's wall-clock time, which returns the number of seconds since the epoch. You need to include the *sys/time.h* header file.

```
double cpuSecond() {  
    struct timeval tp;  
    gettimeofday(&tp, NULL);  
    return ((double)tp.tv_sec + (double)tp.tv_usec*1.e-6);  
}
```

```
double iStart = cpuSecond();  
kernel_name<<>>(argument list);  
cudaDeviceSynchronize();  
double iElaps = cpuSecond() - iStart;
```

Because a kernel call is asynchronous with respect to the host, you need to use *cudaDeviceSynchronize()* to wait for all GPU threads to complete.

# Timing with Nvprof

**nvprof** is a command-line profiling tool. It collects timeline information from your application's CPU and GPU activities, including kernel execution, memory transfers, and CUDA API calls.

```
1s !nvprof ./vectorAdd
```

```
==1337== NVPROF is profiling process 1337, command: ./vectorAdd
result is 8
==1337== Profiling application: ./vectorAdd
==1337== Profiling result:
```

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	46.13%	4.7680us	1	4.7680us	4.7680us	4.7680us	add(int*, int*, int*)
	33.13%	3.4240us	2	1.7120us	1.4080us	2.0160us	[CUDA memcpy HtoD]
	20.74%	2.1440us	1	2.1440us	2.1440us	2.1440us	[CUDA memcpy DtoH]
API calls:	98.92%	135.39ms	3	45.130ms	3.5320us	135.38ms	cudaMalloc
	0.82%	1.1240ms	1	1.1240ms	1.1240ms	1.1240ms	cuDeviceGetPCIBusId
	0.09%	120.57us	101	1.1930us	131ns	50.850us	cuDeviceGetAttribute
	0.09%	117.81us	3	39.268us	5.2530us	101.00us	cudaFree
	0.04%	50.475us	3	16.825us	8.3250us	23.081us	cudaMemcpy
	0.02%	31.072us	1	31.072us	31.072us	31.072us	cuDeviceGetName
	0.02%	24.371us	1	24.371us	24.371us	24.371us	cudaLaunchKernel
	0.01%	12.877us	3	4.2920us	154ns	11.799us	cuDeviceGetCount
	0.00%	2.5120us	2	1.2560us	276ns	2.2360us	cuDeviceGet
	0.00%	479ns	1	479ns	479ns	479ns	cuDeviceTotalMem
	0.00%	446ns	1	446ns	446ns	446ns	cuModuleGetLoadingMode
	0.00%	220ns	1	220ns	220ns	220ns	cuDeviceGetUuid





# 1D VecAdd - Exercise

Change the simple example in the previous slide so that

- it takes a user input N as the array length
- set threads per block (TPB) to 128
- add timers to measure the time spent in Add()
- How do you calculate a thread's index in Add() ?
- For N=1024, how many CUDA threads and thread blocks you used?
- For N=131070, does your program still work? If not, what changes did you make? How many CUDA threads and thread blocks you used.



# 1D VecAdd - Exercise

Increase the vector length  $N$  from a small value at doubling rate, plot the time spent in `Add()` and the total program execution time.

- If you don't see the change in time, make  $N$  larger
- Post your plots to Slack Channel
- Change `ThreadBlock` from 128 to 64 and 256. What changes do you observe?



# Run tests and plots in Jupyter Notebook

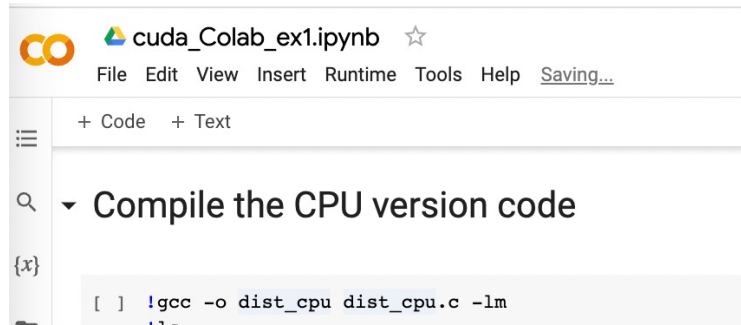
The screenshot shows a Jupyter Notebook interface. At the top, there is a toolbar with icons for 'CO', a Google Cloud logo, and the filename 'cuda\_Colab\_ex1.ipynb'. Below the toolbar is a menu bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', 'Help', and a status message 'All changes saved'. On the left side, there is a sidebar with icons for a menu, search, variables, and files. The main area shows a code cell with a play button icon on the left. The code in the cell is:

```
i=0
n=131072
while i<10:
    !./dist_cpu $n
    i=i+1
    n=n*2
```

Below the code cell, the output is displayed as a single line of text:

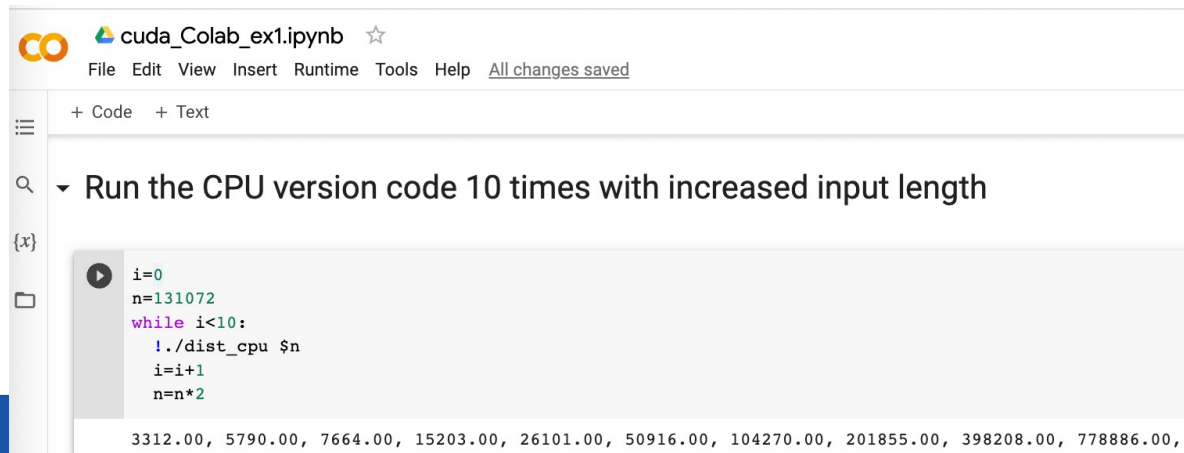
```
3312.00, 5790.00, 7664.00, 15203.00, 26101.00, 50916.00, 104270.00, 201855.00, 398208.00, 778886.00,
```

# Run batch tests in Jupyter Notebook



The screenshot shows the top part of a Jupyter Notebook. The title bar says 'cuda\_Colab\_ex1.ipynb' with a star icon. Below it is a menu bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', 'Help', and 'Saving...'. The left sidebar has icons for a list, search, and a variable '{x}'. The main area has a tab '+ Code' and '+ Text'. Below that is a search bar with a magnifying glass icon and the text 'Compile the CPU version code'. The code cell contains the following text:

```
[ ] !gcc -o dist_cpu dist_cpu.c -lm
```



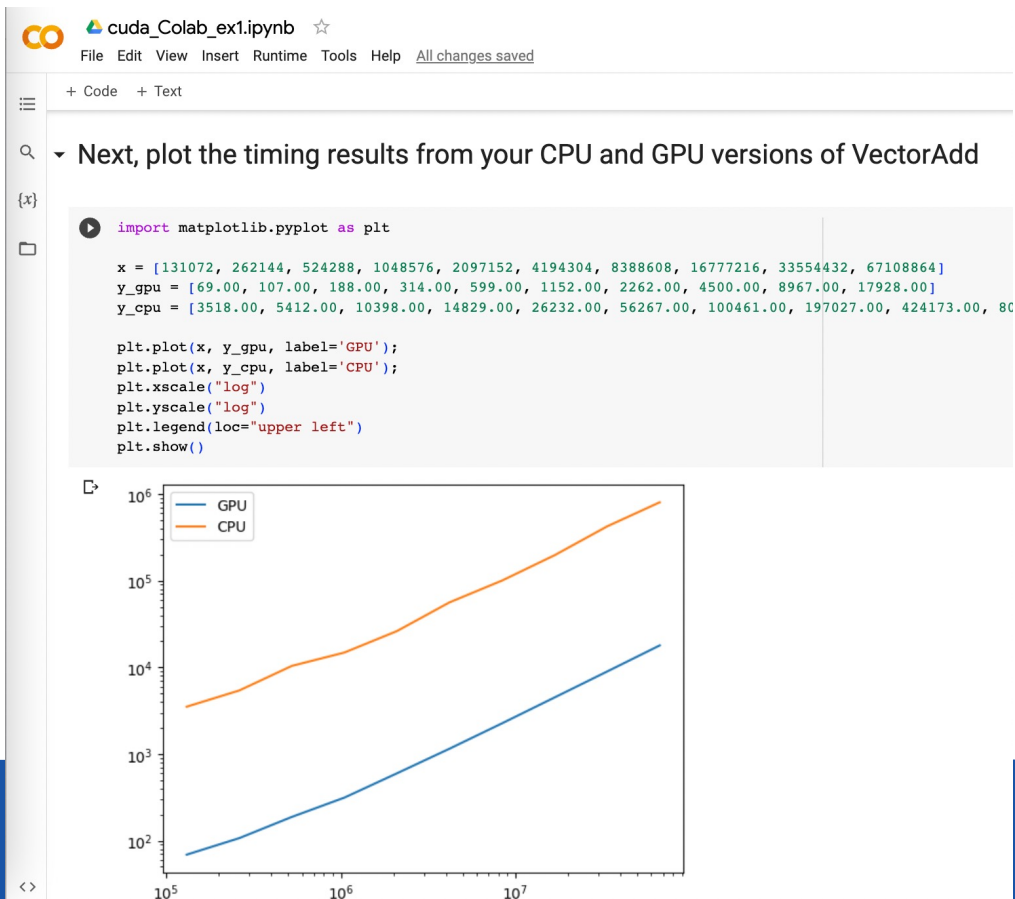
The screenshot shows the bottom part of the Jupyter Notebook. The title bar and menu bar are the same as in the first screenshot. The left sidebar has icons for a list, search, a variable '{x}', and a folder icon. The main area has a tab '+ Code' and '+ Text'. Below that is a search bar with a magnifying glass icon and the text 'Run the CPU version code 10 times with increased input length'. The code cell contains the following text:

```
i=0
n=131072
while i<10:
    !./dist_cpu $n
    i=i+1
    n=n*2
```

Below the code cell, the output is displayed as a single line of text:

```
3312.00, 5790.00, 7664.00, 15203.00, 26101.00, 50916.00, 104270.00, 201855.00, 398208.00, 778886.00,
```

# Quick plots in Jupyter Notebook



# Q & A