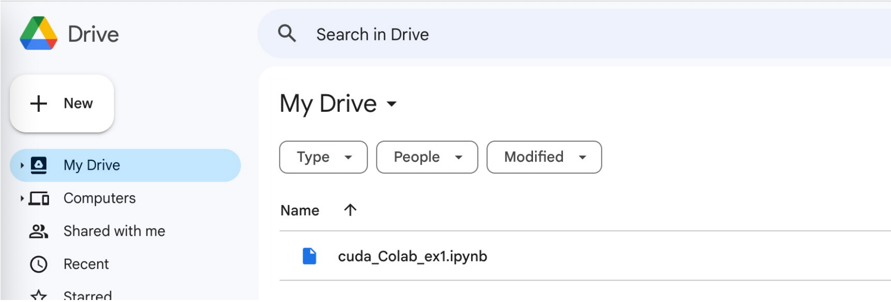
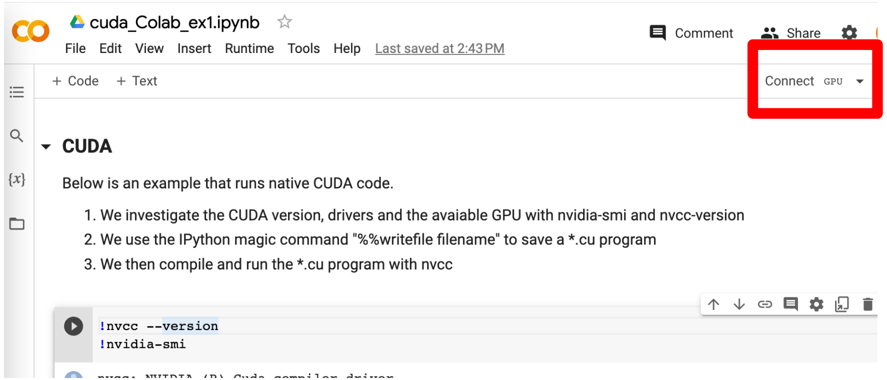
Hands-on Session – Part I

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)

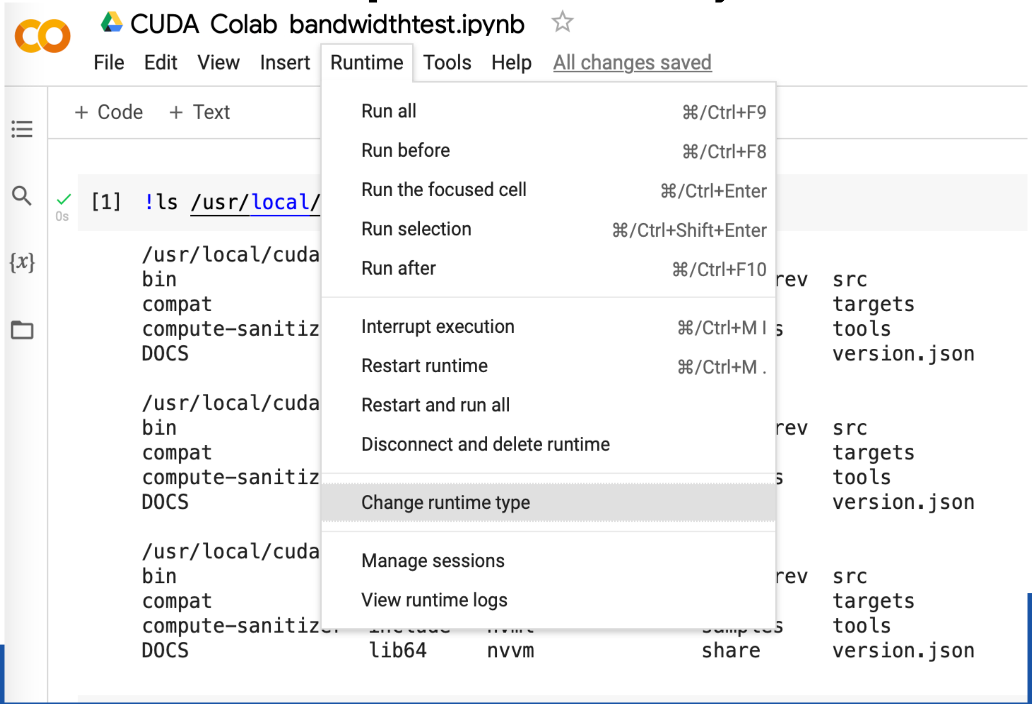


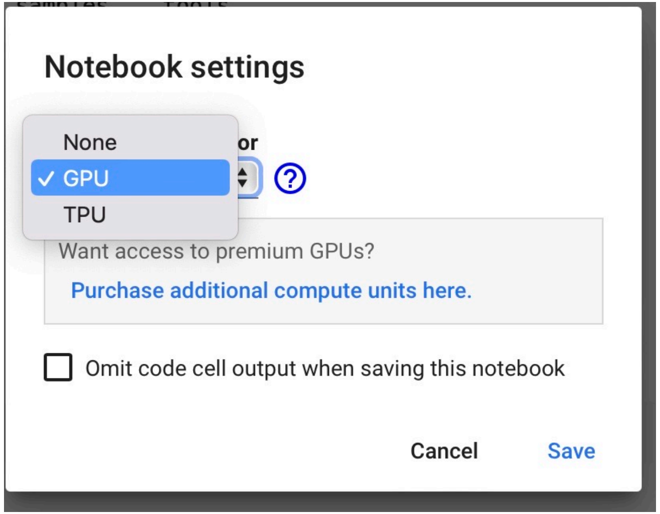


Use CUDA in Google Colab

Setting up the environment

Important note: you need to change your runtime to use

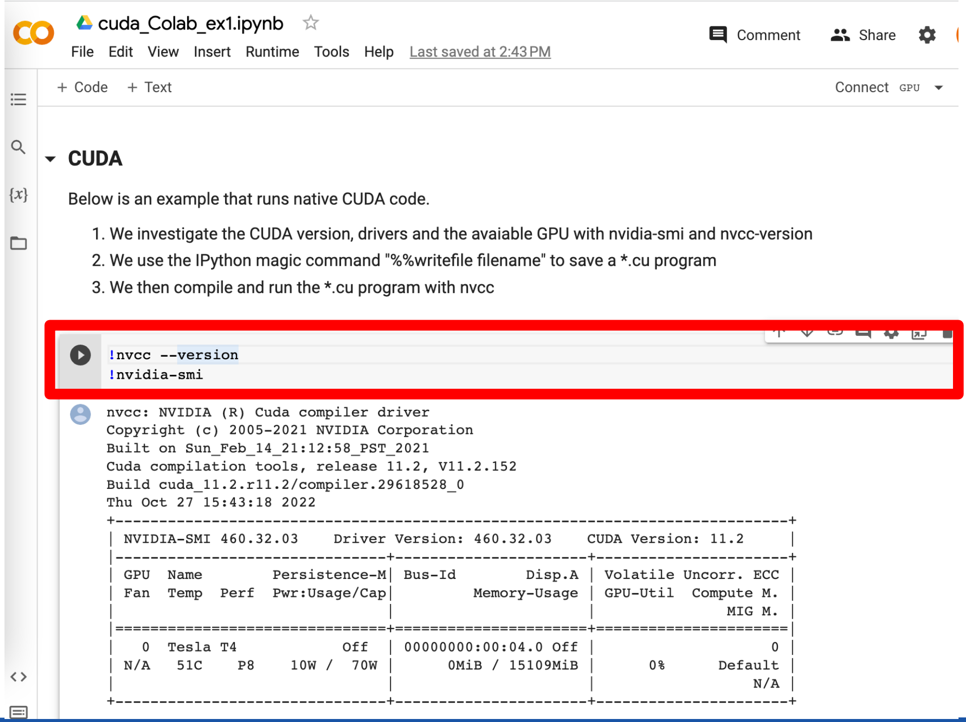




Query Nvidia GPU environment

Question:

Which GPU architecture and CUDA version are you running on?



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'

```
%%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;
}
```

```
Writing vectorAdd.cu
```

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.

```
!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

cuda_Colab_ex1.ipynb

File Edit View Insert Runtime Tools Help All changes saved

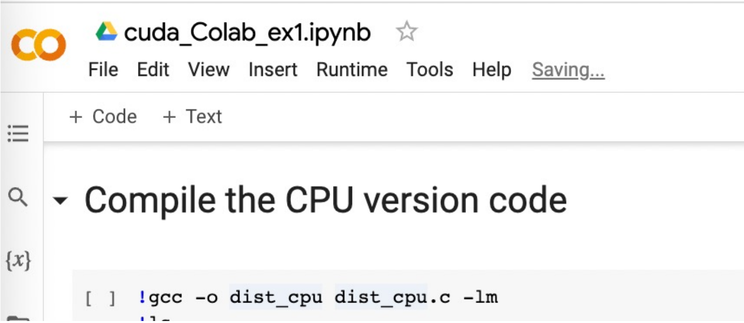
+ Code + Text

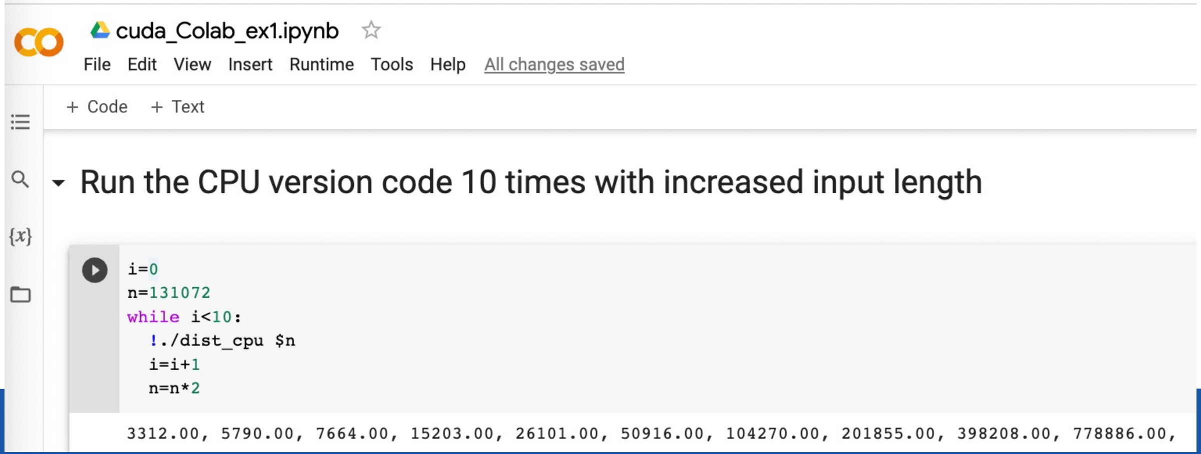
Run the CPU version code 10 times with increased input length

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

```
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





Quick plots in Jupyter Notebook

cuda_Colab_ex1.ipynb ☆

File Edit View Insert Runtime Tools Help All changes saved

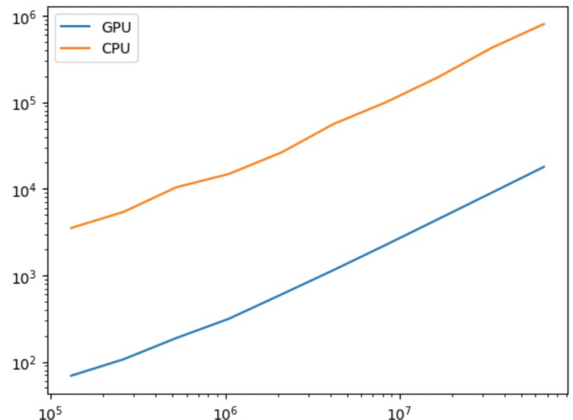
+ Code + Text

Next, plot the timing results from your CPU and GPU versions of VectorAdd

```python
import matplotlib.pyplot as plt

x = [131072, 262144, 524288, 1048576, 2097152, 4194304, 8388608, 16777216, 33554432, 67108864]
y_gpu = [69.00, 107.00, 188.00, 314.00, 599.00, 1152.00, 2262.00, 4500.00, 8967.00, 17928.00]
y_cpu = [3518.00, 5412.00, 10398.00, 14829.00, 26232.00, 56267.00, 100461.00, 197027.00, 424173.00, 80

plt.plot(x, y_gpu, label='GPU');
plt.plot(x, y_cpu, label='CPU');
plt.xscale("log")
plt.yscale("log")
plt.legend(loc="upper left")
plt.show()
```



Q & A