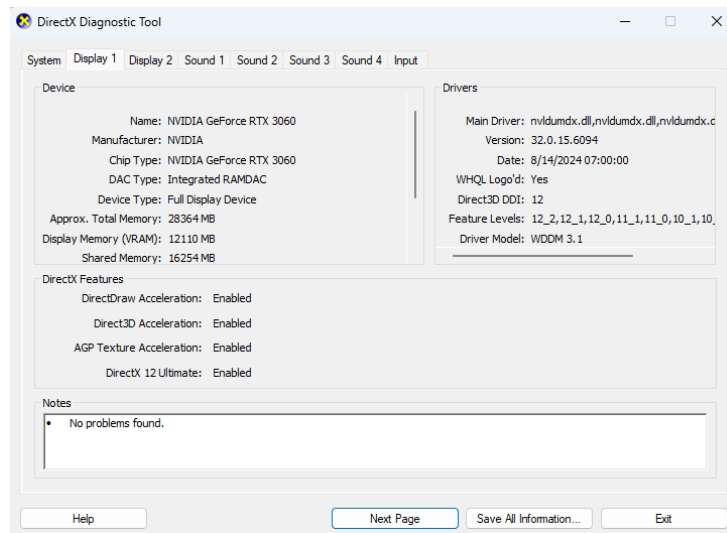


## Phần I. Cài đặt CUDA

### 1.1 Cài đặt trên máy có GPU NVIDIA

#### Bước 1: Kiểm tra phần cứng

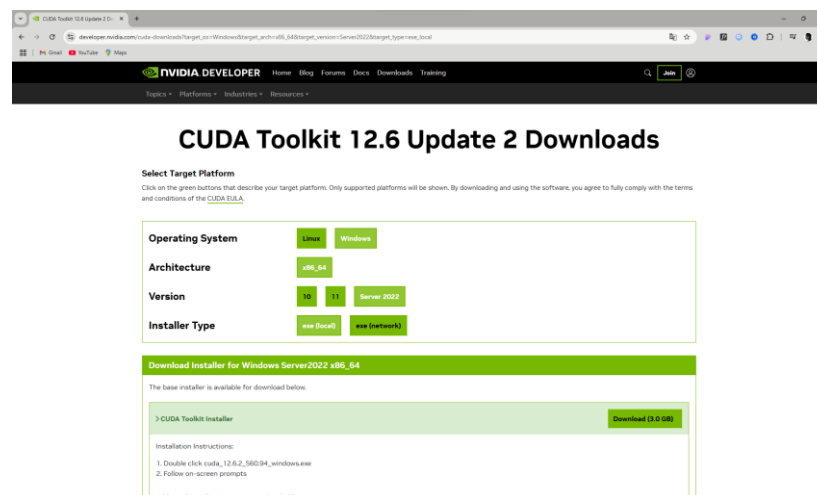
- Nhấn **Start** => gõ vào ô **Run** (hoặc nhấn tổ hợp phím **Windows + R**). Gõ vào đó chữ **“dxdiag”** => nhấn **Enter**.
- Chọn **Display** => toàn bộ thông số chi tiết của card sẽ hiển thị. Trên các thông tin hiển thị GPU



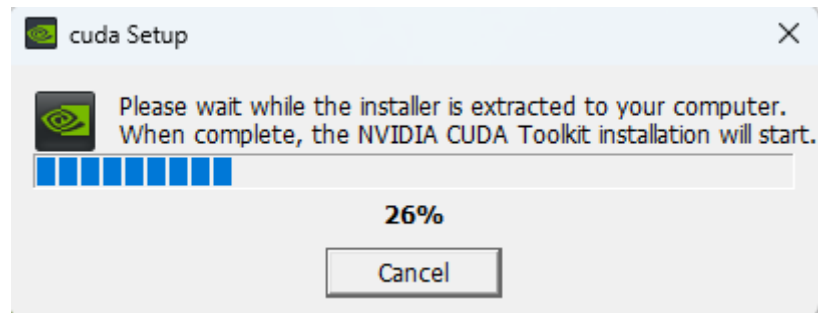
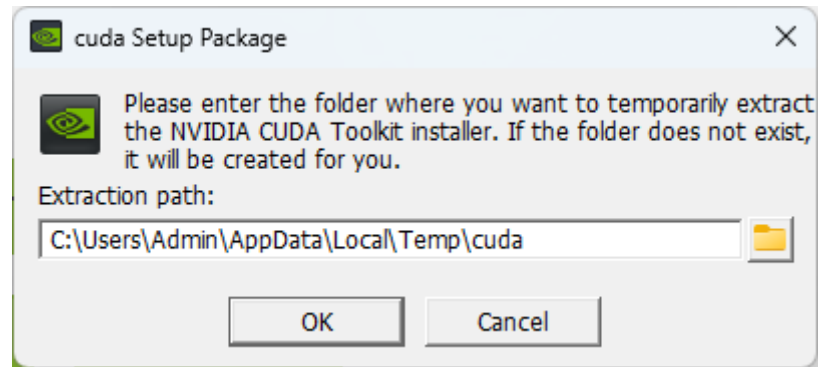
#### Bước 2:

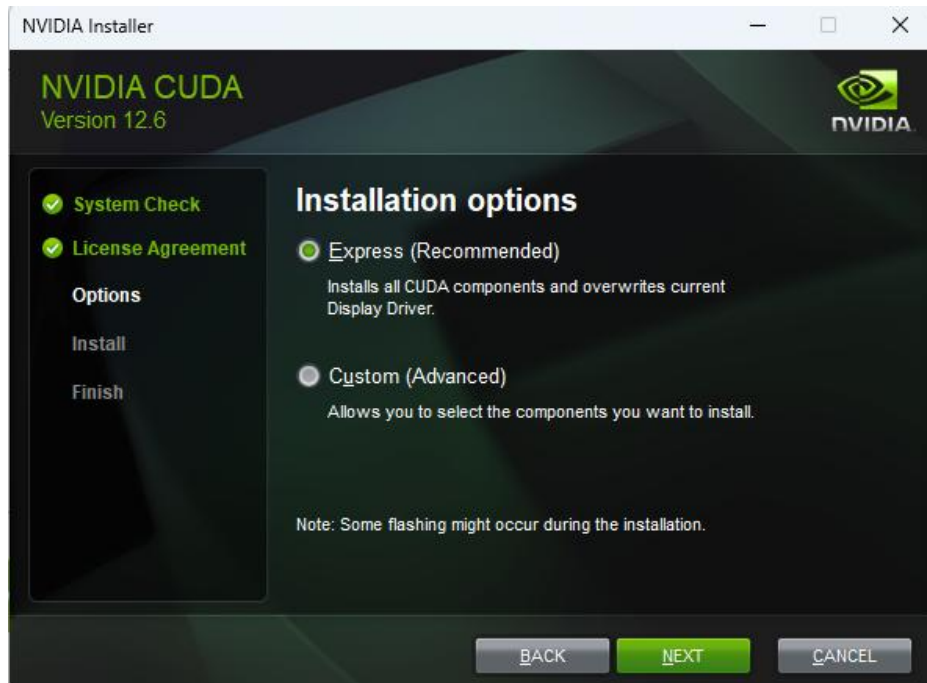
Tải và cài đặt phiên bản CUDA Toolkit phù hợp từ trang chủ NVIDIA.

<https://developer.nvidia.com/cuda-downloads>



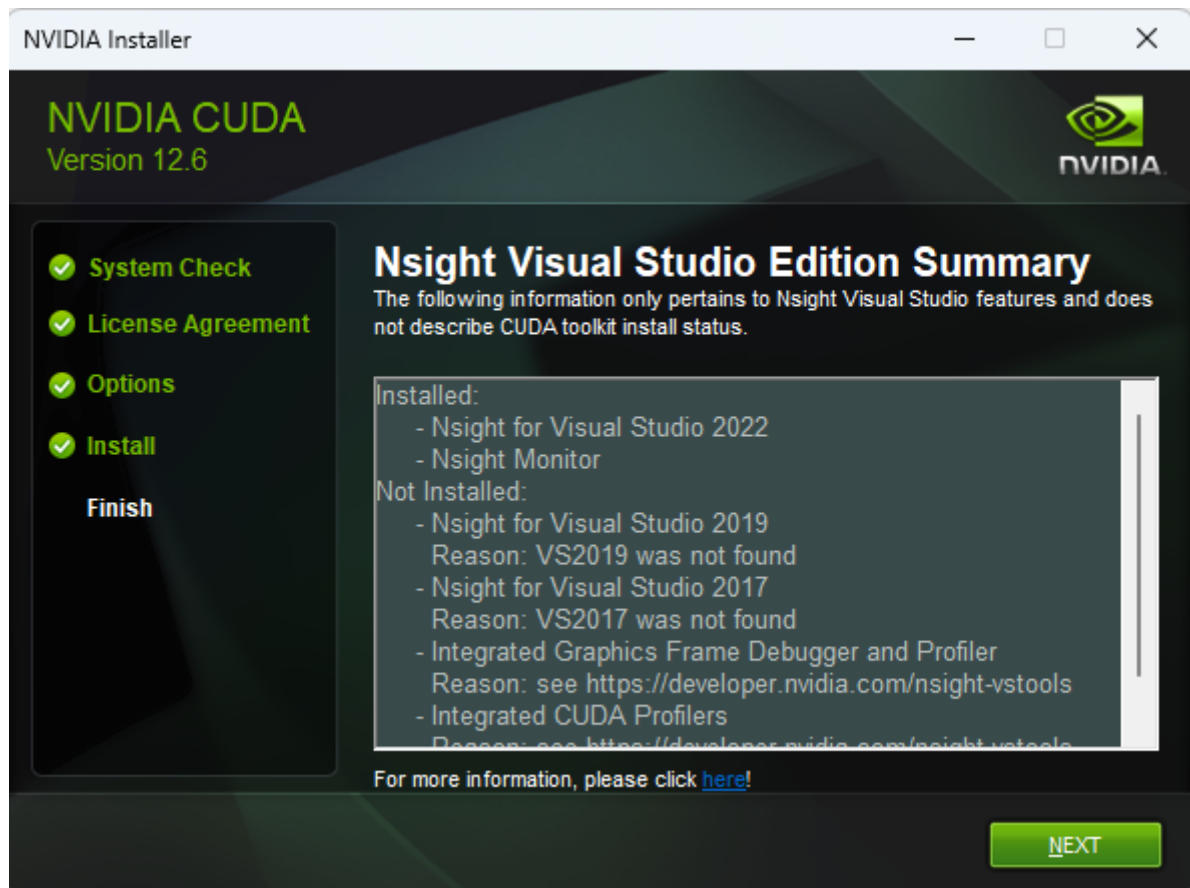
- Chọn đường dẫn

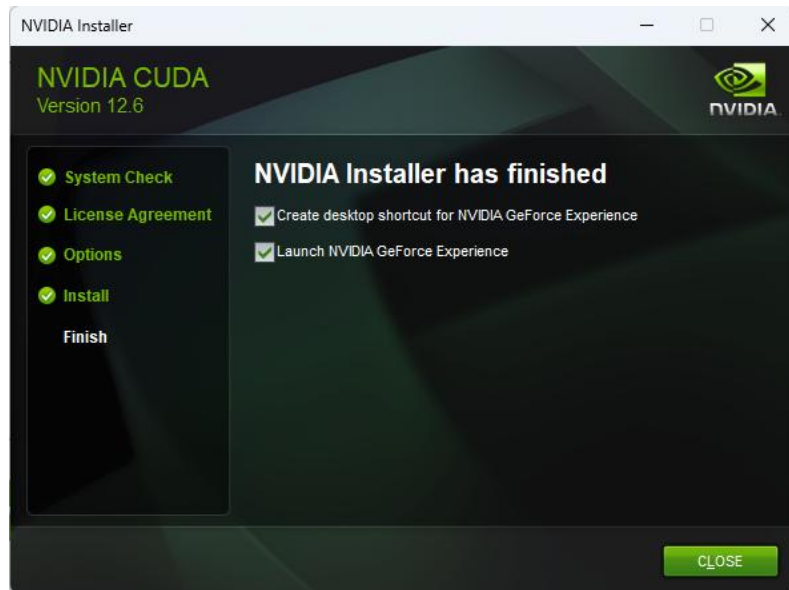




-Express: tải tất cả các thành phần Cuda và ghi đè lên Display driver hiện tại.

-Custom: chỉ tải những thành phần muốn tải





Kiểm tra cài đặt: Chạy lệnh `nvidia-smi` để kiểm tra phiên bản CUDA.

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.22631.4317]
(c) Microsoft Corporation. All rights reserved.

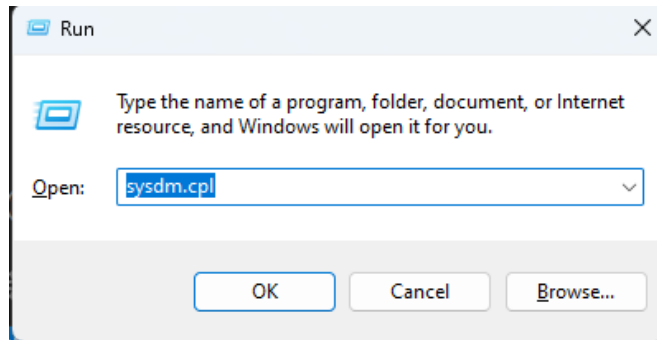
C:\Users\Admin>nvidia-smi
Sun Nov 17 14:29:26 2024
```

NVIDIA-SMI 560.94				Driver Version: 560.94				CUDA Version: 12.6			
GPU	Name	Fan	Temp	Perf	Driver-Model	Bus-Id	Memory-Usage	Disp.A	Volatile GPU-Util	Uncorr. Compute M.	ECC MIG M.
0	NVIDIA GeForce RTX 3060	0%	49C	P8	WDDM 19W / 170W	00000000:01:00:00	1425MiB / 12288MiB	On	4%	Default	N/A

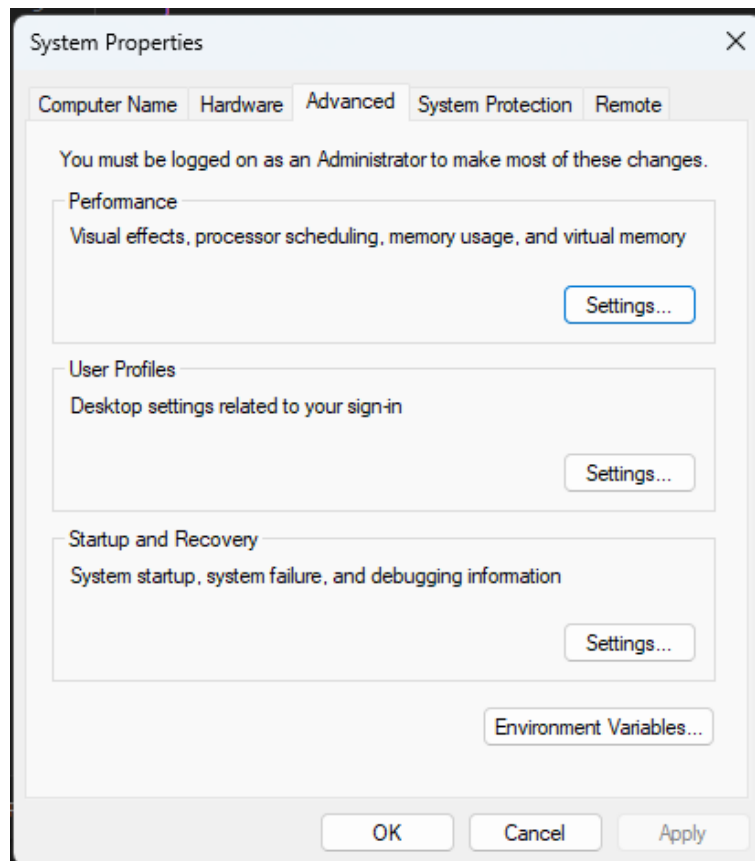
```
Processes:
```

GPU	GI ID	CI ID	PID	Type	Process name	GPU Memory Usage
0	N/A	N/A	2356	C+G	...on\130.0.2849.80\msedgebview2.exe	N/A
0	N/A	N/A	3220	C+G	...15.0_x64_8wekyb3d8bbwe\Cortana.exe	N/A
0	N/A	N/A	4724	C+G	...oogle\Chrome\Application\chrome.exe	N/A
0	N/A	N/A	5840	C+G	...crosoft\Edge\Application\msedge.exe	N/A
0	N/A	N/A	5880	C+G	...cef\cef.win7x64\steamwebhelper.exe	N/A
0	N/A	N/A	6124	C+G	...oogle\Chrome\Application\chrome.exe	N/A
0	N/A	N/A	6796	C+G	C:\Windows\explorer.exe	N/A
0	N/A	N/A	10972	C+G	...Programs\Zalo\Zalo-24.11.1\Zalo.exe	N/A
0	N/A	N/A	12820	C+G	...ft.Office\root\Office16\WINWORD.EXE	N/A
0	N/A	N/A	15040	C+G	...nt.CBS_cw5n1h2txyewy\SearchHost.exe	N/A
0	N/A	N/A	15644	C+G	...pdnekdrrrea0\XboxGameBar\Spotify.exe	N/A
0	N/A	N/A	16456	C+G	...siveControlPanel\SystemSettings.exe	N/A
0	N/A	N/A	17352	C+G	...air\Consair iCUE5 Software\iCUE.exe	N/A
0	N/A	N/A	17492	C+G	...US\ArmouryDevice\asus_framework.exe	N/A
0	N/A	N/A	18268	C+G	...t.LockApp_cw5n1h2txyewy\LockApp.exe	N/A
0	N/A	N/A	19028	C+G	...2txyewy\StartMenuExperienceHost.exe	N/A
0	N/A	N/A	20624	C+G	C:\Windows\explorer.exe	N/A
0	N/A	N/A	20776	C+G	...min\AppData\Local\Lark\app\Lark.exe	N/A
0	N/A	N/A	21100	C+G	...ejd91yc\AdobeNotificationClient.exe	N/A
0	N/A	N/A	23264	C+G	...CloudFlare WARP\CloudFlare WARP.exe	N/A
0	N/A	N/A	24084	C+G	...4.11.1\plugins\capture\ZaloCall.exe	N/A
0	N/A	N/A	25772	C+G	...5.0_x64_t4vj0pshngkwm\Telegram.exe	N/A
0	N/A	N/A	28288	C+G	...ekyb3d8bbwe\PhoneExperienceHost.exe	N/A
0	N/A	N/A	28528	C+G	...CBS_cw5n1h2txyewy\TextInputHost.exe	N/A
0	N/A	N/A	28704	C+G	...les\Microsoft OneDrive\OneDrive.exe	N/A
0	N/A	N/A	28932	C+G	...on\HEX\Creative Cloud UI Helper.exe	N/A
0	N/A	N/A	29016	C+G	...5n1h2txyewy\ShellExperienceHost.exe	N/A
0	N/A	N/A	29024	C+G	...oaderplus\4kvideodownloaderplus.exe	N/A

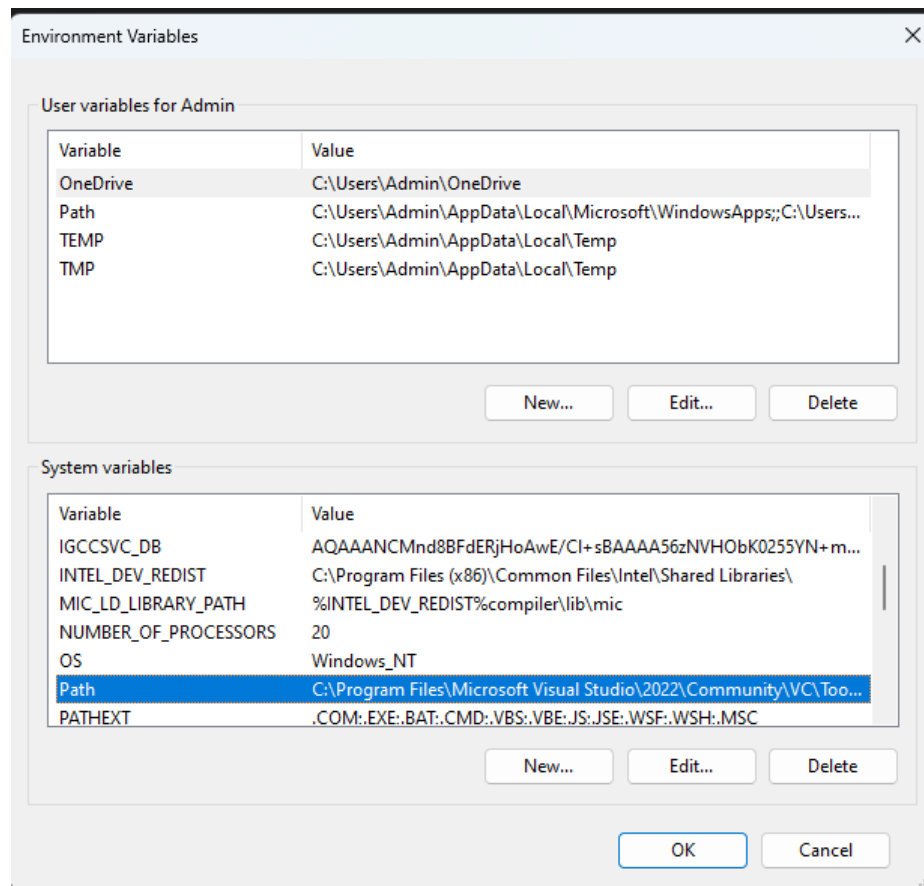
Cài đặt cuda trên visual studio code



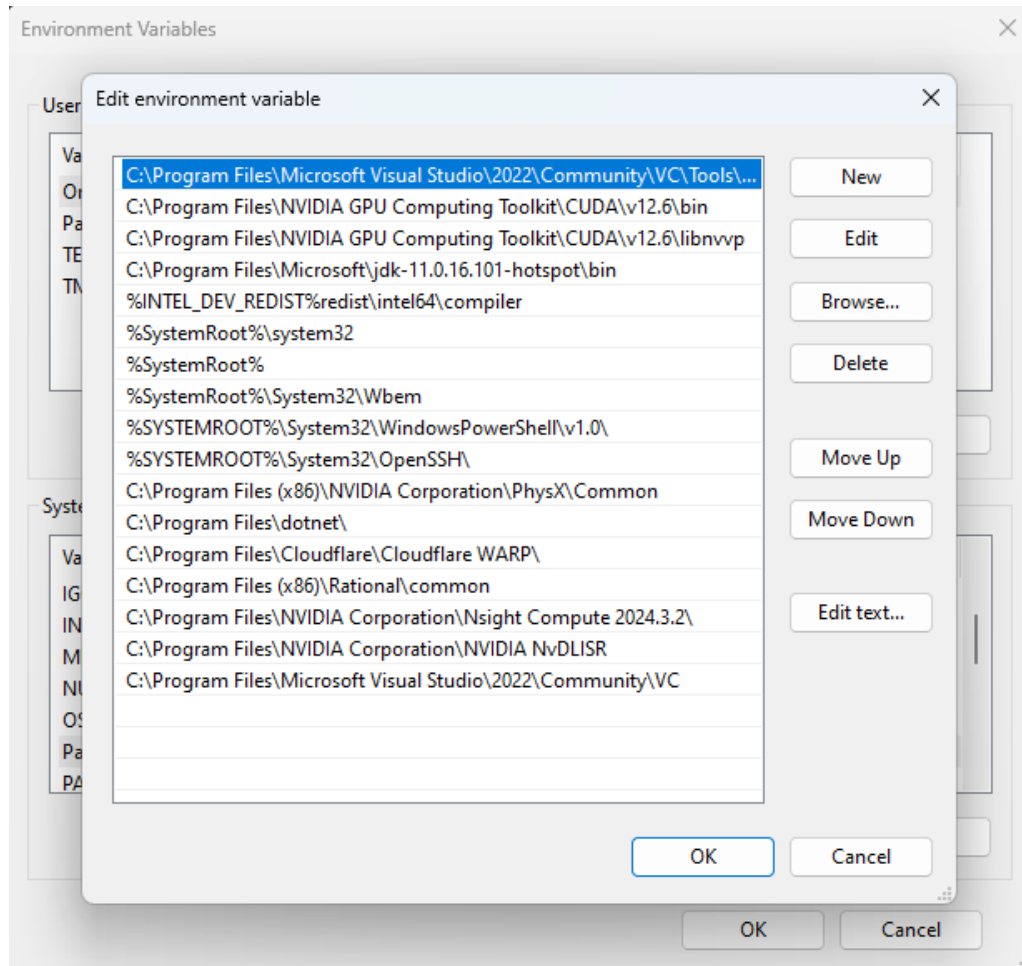
Chạy lệnh sysdm.cpl



Chọn Advanced-> Environment Variables



Trong system variables chọn Path->Edit



Chọn new thêm đường dẫn :

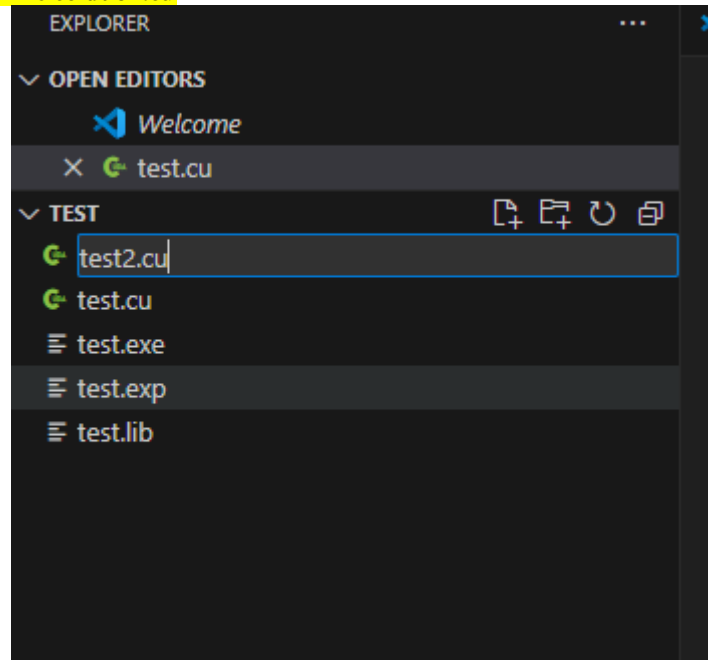
-VSCode 2022

"C:\ProgramFiles\MicrosoftVisualStudio\2022\Community\VC\Tools\MSVC\14.36.32532\bin\Hostx64\x64"

-Các phiên bản thấp hơn

"D:\Program Files\Microsoft Visual Studio 11.0\VC\bin"

Tại VSCode , tạo new file có đuôi .cu



- Chạy chương trình ví dụ cộng hai mảng

```
3 // Kernel chạy trên GPU
4 __global__ void addArrays(const int *a, const int *b, int *c, int size) {
5     int idx = threadIdx.x + blockIdx.x * blockDim.x;
6     if (idx < size) {
7         c[idx] = a[idx] + b[idx];
8     }
9 }
10
11 int main() {
12     // Kích thước mảng
13     const int arraySize = 10;
14
15     // Cấp phát bộ nhớ trên host
16     int hostA[arraySize], hostB[arraySize], hostC[arraySize];
17     for (int i = 0; i < arraySize; ++i) {
18         hostA[i] = i;
19         hostB[i] = i * 2;
20     }
21
22     // Cấp phát bộ nhớ trên device
23     int *devA, *devB, *devC;
24     cudaMalloc((void**)&devA, arraySize * sizeof(int));
25     cudaMalloc((void**)&devB, arraySize * sizeof(int));
26     cudaMalloc((void**)&devC, arraySize * sizeof(int));
27
28     // Copy dữ liệu từ host sang device
29     cudaMemcpy(devA, hostA, arraySize * sizeof(int), cudaMemcpyHostToDevice);
30     cudaMemcpy(devB, hostB, arraySize * sizeof(int), cudaMemcpyHostToDevice);
31
32     // Khởi chạy kernel
33     int blockSize = 256; // Số thread trong một block
34     int numBlocks = (arraySize + blockSize - 1) / blockSize; // Số block cần thiết
35     addArrays<<<numBlocks, blockSize>>>>(devA, devB, devC, arraySize);
36
37     // Copy kết quả từ device về host
38     cudaMemcpy(hostC, devC, arraySize * sizeof(int), cudaMemcpyDeviceToHost);
39
40     // In kết quả
41     for (int i = 0; i < arraySize; ++i) {
42         printf("hostA[%d] = %d, hostB[%d] = %d, hostC[%d] = %d\n", i, hostA[i], hostB[i], hostC[i], hostC[i]);
43     }
44 }
```

```
[Running] cd "c:\Users\Admin\Downloads\test\" && nvcc test.cu -o test && "c:\Users\Admin\Downloads\test\test.exe"
tmpxft_000002c64_000000000-10_test.cudafec.cpp
  Creating library test.lib and object test.exp
Result: 0 3 6 9 12 15 18 21 24 27
[Done] exited with code=0 in 2.365 seconds
```

- Khởi tạo mảng trên CPU:

- hostA: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.
- hostB: {0, 2, 4, 6, 8, 10, 12, 14, 16, 18}.



- **Copy dữ liệu sang GPU:**

- devA và devB trên GPU nhận dữ liệu từ hostA và hostB.

- **Chạy Kernel:**

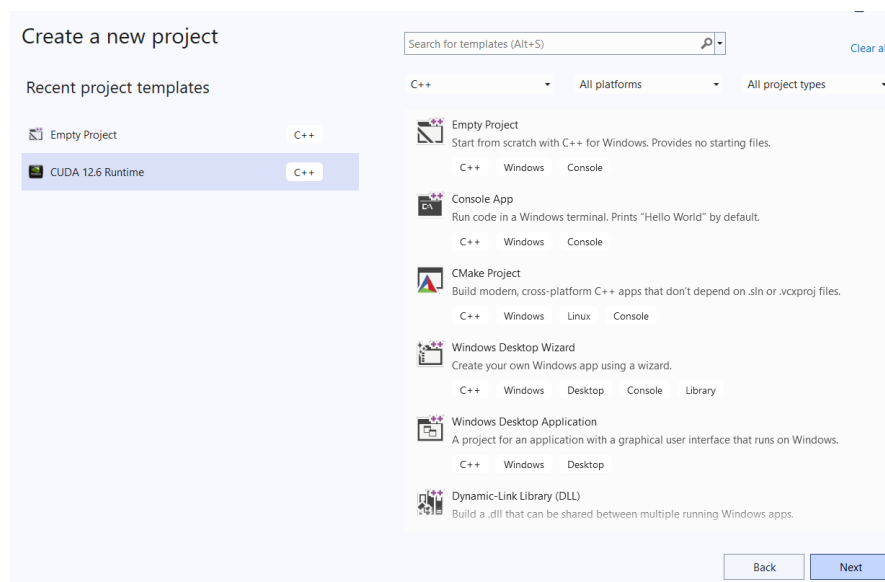
- Mỗi thread thực hiện phép tính  $c[idx] = a[idx] + b[idx]$  trên GPU.
- Kết quả:
  - devC: {0, 3, 6, 9, 12, 15, 18, 21, 24, 27}.

- **Copy kết quả về CPU:**

- hostC: {0, 3, 6, 9, 12, 15, 18, 21, 24, 27}.

**TẠI VISUAL STUDIO 2019/2022**

Tại bước tạo project, chọn CUDA Runtime -> chọn next:



Tạo tên project và chọn nơi lưu trữ:

Configure your new project

CUDA 12.6 Runtime C++ CUDA Windows Linux Cloud Console DataScience Desktop Machine Learning

Project name

Location  
 --

Solution name [?](#)

☐ Place solution and project in the same directory

Project will be created in "D:\1\_PROJETS\HPC\code\Cuda\_Project1\Cuda\_Project1"

Back Create

Hoàn thành:

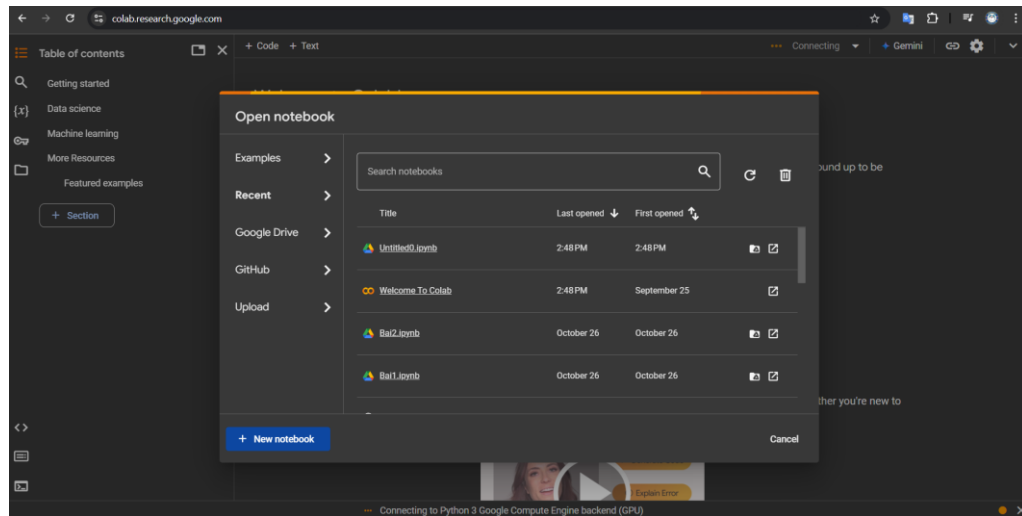
```

1
2 #include "cuda_runtime.h"
3 #include "device_launch_parameters.h"
4
5 #include <stdio.h>
6
7 cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int size);
8
9 _global__ void addKernel(int *c, const int *a, const int *b)
10 {
11     int i = threadIdx.x;
12     c[i] = a[i] + b[i];
13 }
14
15 int main()
16 {
17     const int arraySize = 5;
18     const int a[arraySize] = { 1, 2, 3, 4, 5 };
19     const int b[arraySize] = { 10, 20, 30, 40, 50 };
20     int c[arraySize] = { 0 };
21
22     // Add vectors in parallel.
23     cudaError_t cudaStatus = addWithCuda(c, a, b, arraySize);
24     if (cudaStatus != cudaSuccess) {
25         fprintf(stderr, "addWithCuda failed!\n");
26         return 1;
27     }
28
29     printf("(1,2,3,4,5) + (10,20,30,40,50) = (%d,%d,%d,%d,%d)\n",
30           c[0], c[1], c[2], c[3], c[4]);
31
32     // cudaDeviceReset must be called before exiting in order for profiling and
33     // tracing tools such as Nsight and Visual Profiler to show complete traces.

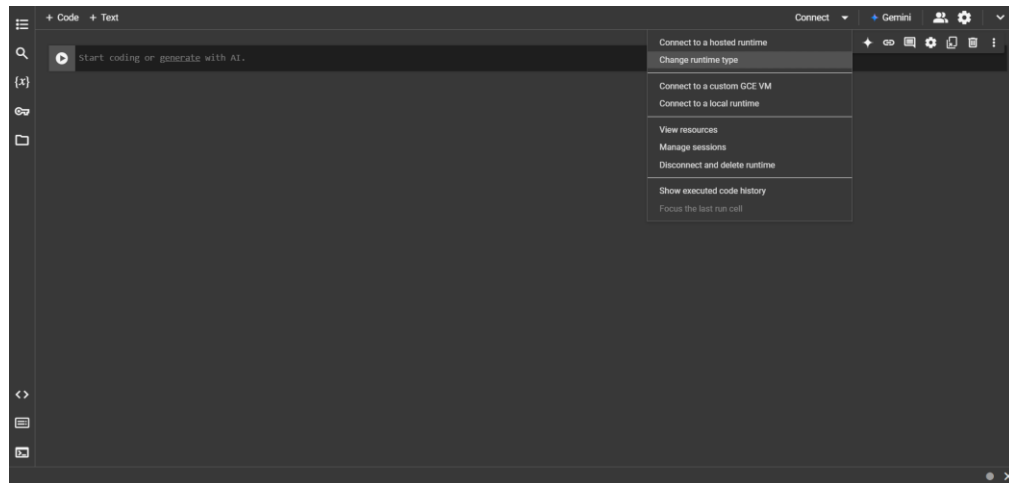
```

## 1.2 Cài đặt CUDA trên Google Colab

Bước 1: Truy cập vào trang web: <https://colab.research.google.com/> trong trình duyệt và nhấp chuột vào New notebook.



Bước 2: Click change run time type, sau đó đổi Hardware accelerator thành T4 GPU, save lại rồi nhấn nút connect.



## Change runtime type

### Runtime type

Python 3



### Hardware accelerator



CPU



T4 GPU



A100 GPU



L4 GPU



TPU v2-8

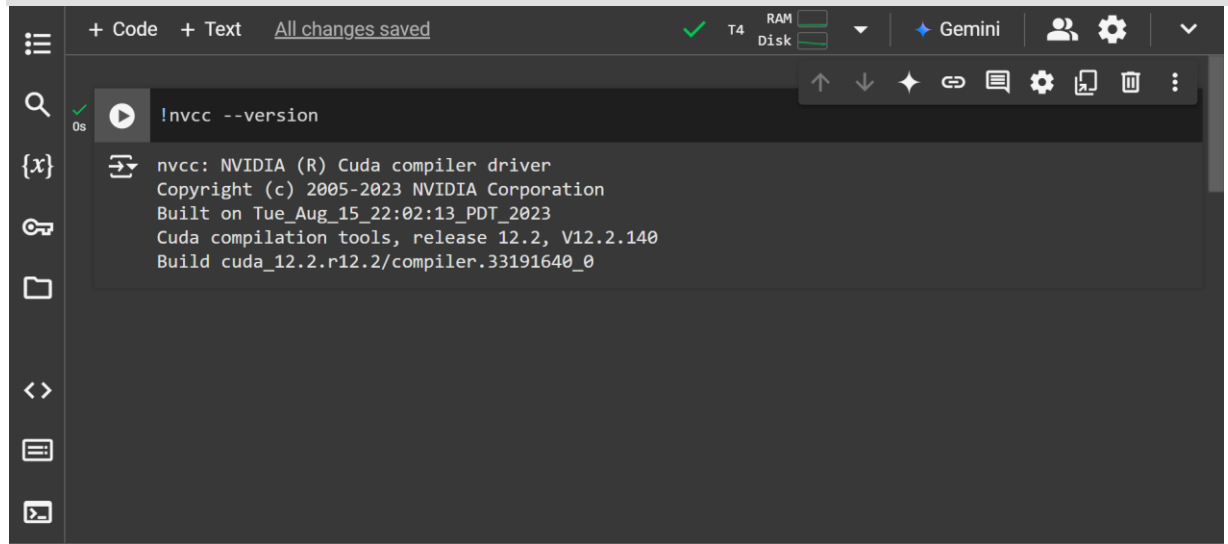
Want access to premium GPUs? [Purchase additional compute units](#)

Cancel

Save

Bước 3: Kiểm tra phiên bản CUDA đang chạy bằng cách chạy dòng lệnh:

```
!nvcc --version
```

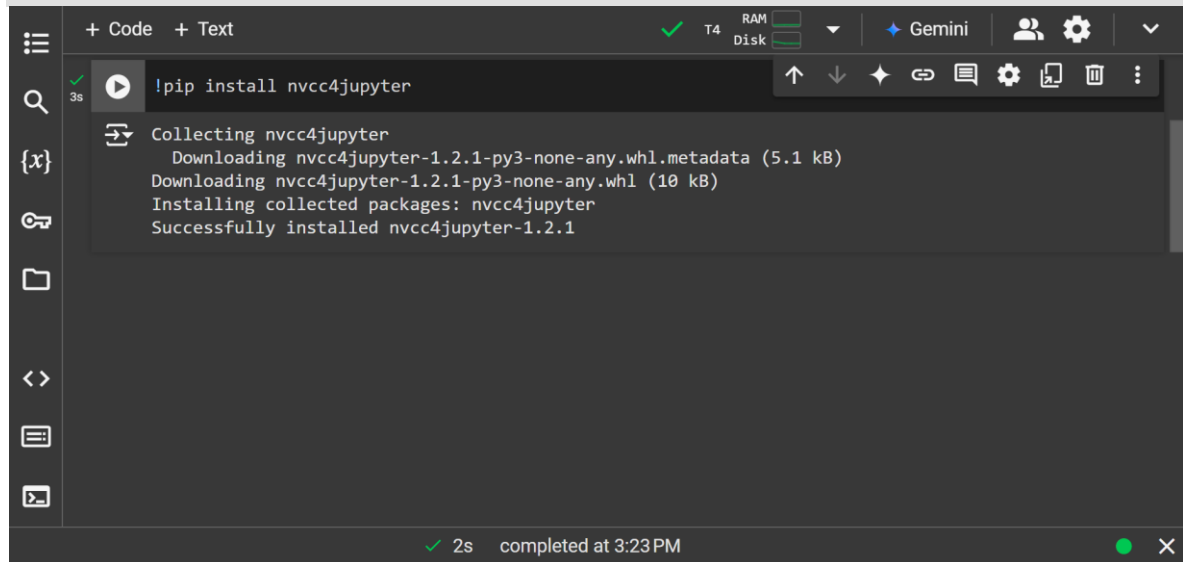


The screenshot shows a Jupyter Notebook interface with a terminal window open. The terminal displays the output of the command `!nvcc --version`. The output text is as follows:

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Tue_Aug_15_22:02:13_PDT_2023
Cuda compilation tools, release 12.2, V12.2.140
Build cuda_12.2.r12.2/compiler.33191640_0
```

Bước 4: Chạy dòng lệnh sau để cài đặt tiện ích mở rộng nhỏ giúp chạy nvcc từ các ô Notebook:

```
!pip install nvcc4jupyter
```



The screenshot shows a Jupyter Notebook interface with a terminal window open. The terminal displays the output of the command `!pip install nvcc4jupyter`. The output text is as follows:

```
Collecting nvcc4jupyter
  Downloading nvcc4jupyter-1.2.1-py3-none-any.whl.metadata (5.1 kB)
  Downloading nvcc4jupyter-1.2.1-py3-none-any.whl (10 kB)
Installing collected packages: nvcc4jupyter
Successfully installed nvcc4jupyter-1.2.1
```

At the bottom of the terminal window, a status bar indicates: `✓ 2s completed at 3:23 PM`.

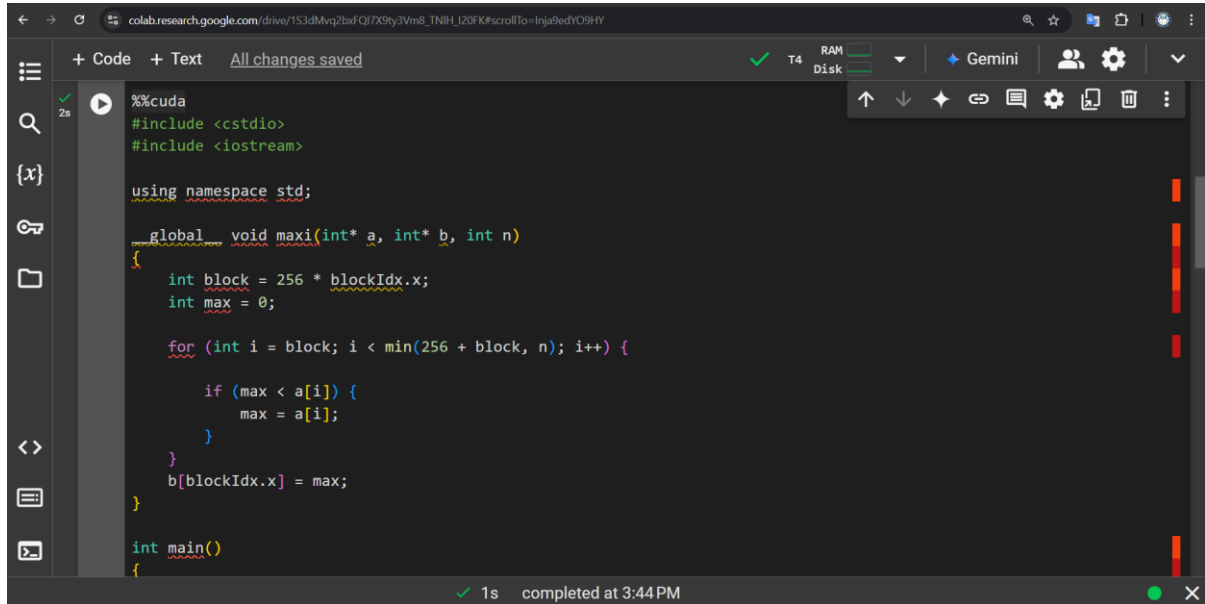
Bước 5: Tải tiện ích mở rộng hỗ trợ CUDA bằng câu lệnh:

```
%load_ext nvcc4jupyter
```

Khi tải extension này, nếu muốn chạy một đoạn code của CUDA trong 1 ô notebook thì chỉ cần dùng command %%cuda ở đầu ô đó để báo rằng tại ô notebook này có chứa code CUDA.

### Chạy chương trình ví dụ:

Đoạn code CUDA sau thực hiện tìm phần tử lớn nhất trong một mảng số nguyên sử dụng GPU. Mảng được tạo ngẫu nhiên với số phần tử là 1024.



```
%%cuda
#include <stdio>
#include <iostream>

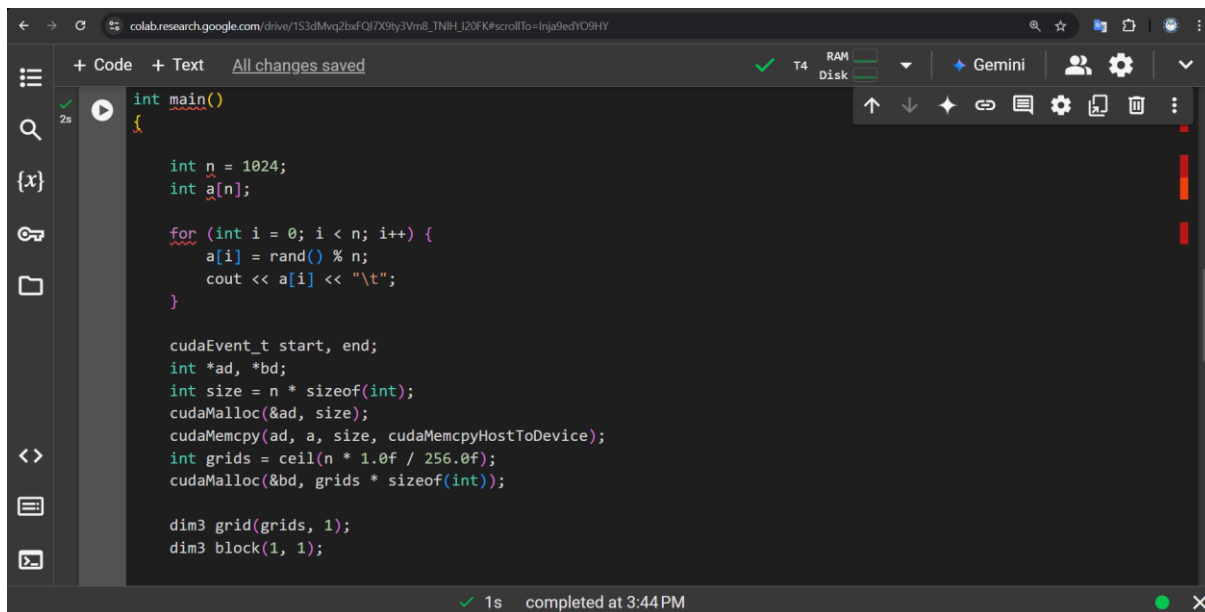
using namespace std;

__global__ void maxi(int* a, int* b, int n)
{
    int block = 256 * blockIdx.x;
    int max = 0;

    for (int i = block; i < min(256 + block, n); i++) {
        if (max < a[i]) {
            max = a[i];
        }
    }
    b[blockIdx.x] = max;
}

int main()
{
    // ... (code continues) ...
}
```

1s completed at 3:44 PM



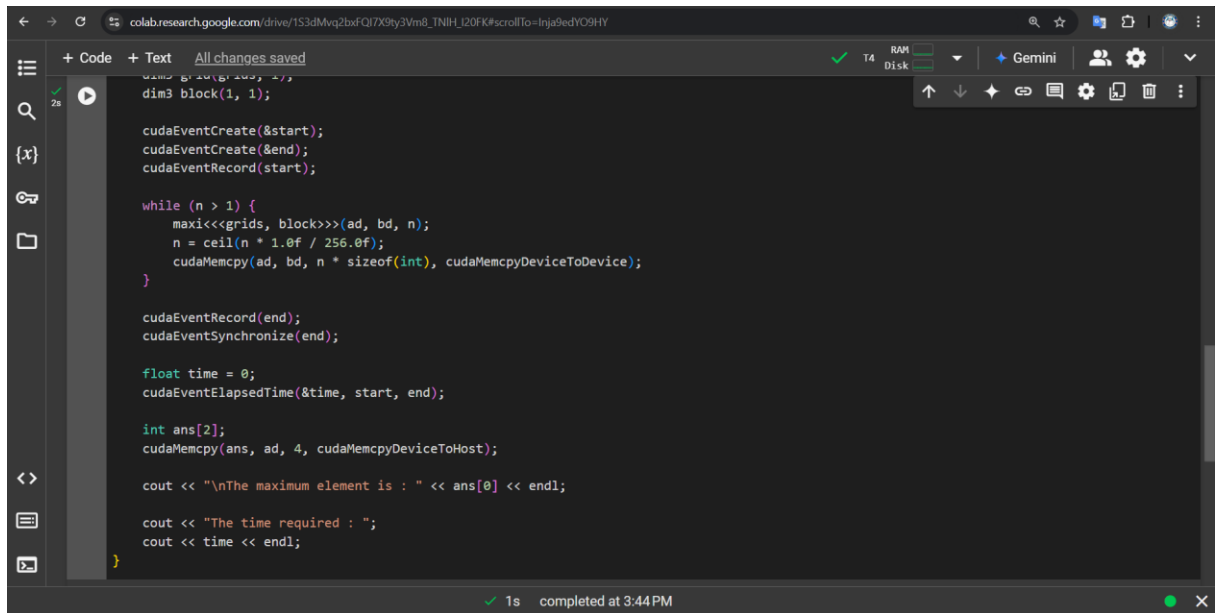
```
int main()
{
    int n = 1024;
    int a[n];

    for (int i = 0; i < n; i++) {
        a[i] = rand() % n;
        cout << a[i] << "\t";
    }

    cudaEvent_t start, end;
    int *ad, *bd;
    int size = n * sizeof(int);
    cudaMalloc(&ad, size);
    cudaMemcpy(ad, a, size, cudaMemcpyHostToDevice);
    int grids = ceil(n * 1.0f / 256.0f);
    cudaMalloc(&bd, grids * sizeof(int));

    dim3 grid(grids, 1);
    dim3 block(1, 1);
    // ... (code continues) ...
}
```

1s completed at 3:44 PM



```
colab.research.google.com/drive/1S3dMvq2bxfQf7X9ty3Vm8_TNIH_I20FK#scrollTo=lnja9edYO9HY

+ Code + Text All changes saved

dim3 grids(1024, 1);
dim3 block(1, 1);

cudaEventCreate(&start);
cudaEventCreate(&end);
cudaEventRecord(start);

while (n > 1) {
    maxi<<<grids, block>>>(ad, bd, n);
    n = ceil(n * 1.0f / 256.0f);
    cudaMemcpy(ad, bd, n * sizeof(int), cudaMemcpyDeviceToDevice);
}

cudaEventRecord(end);
cudaEventSynchronize(end);

float time = 0;
cudaEventElapsedTime(&time, start, end);

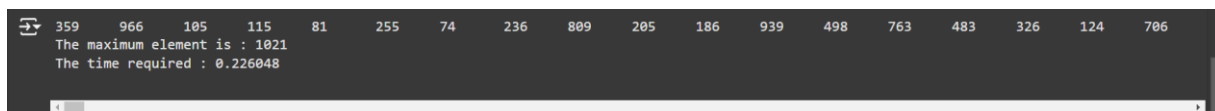
int ans[2];
cudaMemcpy(ans, ad, 4, cudaMemcpyDeviceToHost);

cout << "\nThe maximum element is : " << ans[0] << endl;

cout << "The time required : ";
cout << time << endl;
}
```

1s completed at 3:44 PM

Với kết quả sau khi chạy:



```
359 966 105 115 81 255 74 236 809 205 186 939 498 763 483 326 124 706
The maximum element is : 1021
The time required : 0.226048
```

## Phần 2:

### 2.1 Các thuật ngữ trong GPU

#### **PHYSICAL và LOGICAL**

Đây là 2 thuật ngữ quan trọng, mong các bạn đọc thật kĩ

Ta xét một ví dụ:

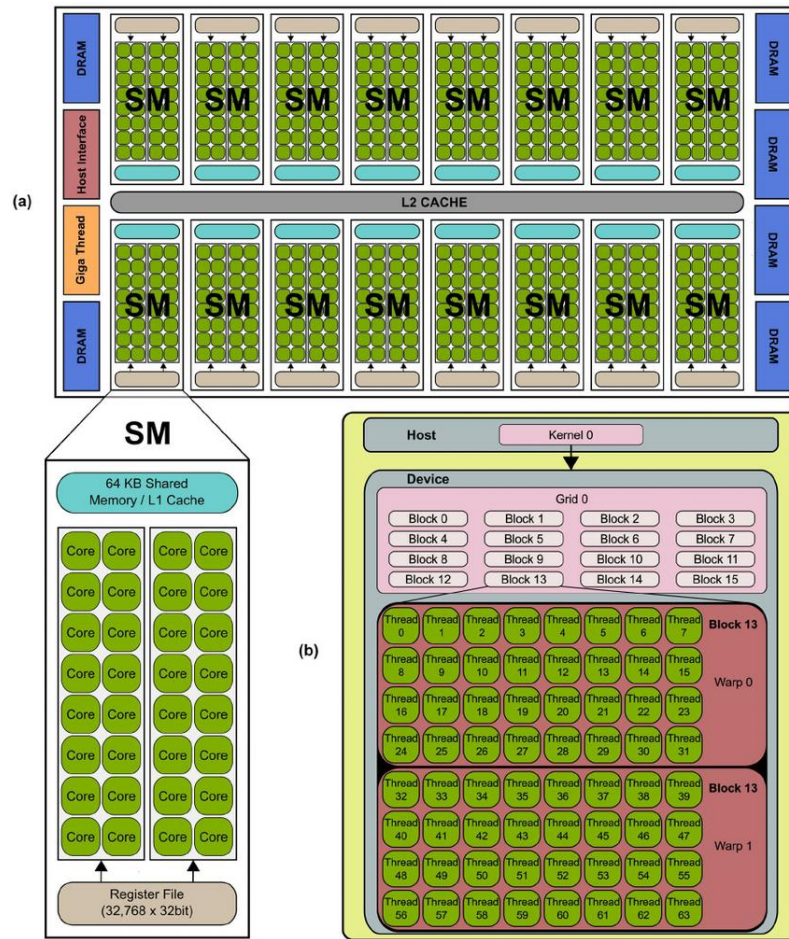
Một ngôi trường sẽ có nhiều lớp học, trong mỗi 1 lớp học sẽ có nhiều học sinh( số lượng lớp học và số lượng học sinh mỗi lớp tùy thuộc vào từng trường khác nhau). Chúng ta có 1 núi các công việc( rất nhiều) cần phân phối cho các bạn học sinh xử lý và chúng ta phải tuân thủ 2 quy tắc:

1. Mỗi 1 lớp học chỉ nhận **tối đa 1024 công việc**.
2. Tại một thời điểm, trong một lớp học sẽ nhận (**32 \* số warp**) công việc được thực thi (số lượng warp tùy thuộc vào mỗi máy tính, warp là gì sẽ được giải thích sau) => tức là nếu chúng ta có 5 lớp học => sẽ có  $32 * \text{số warp} * 5$  công việc được thực thi => N lớp học thì có  $32 * \text{số warp} * N$  công việc được thực thi

**Physical** có thể hiểu đơn giản là những thứ có thể thấy được, số lượng cố định và trong ví dụ này là “các bạn học sinh”, **logical** nói nôm na là thứ không thấy được nhưng mình có thể hình dung và lường tượng ra được, nhưng số lượng không cố định và trong trường hợp này là “số công việc”.



Physical ứng với SM, SP



Như trong hình chúng ta có thể thấy có 16SM và trong mỗi SM có 32 core. Vậy SM và core là gì?

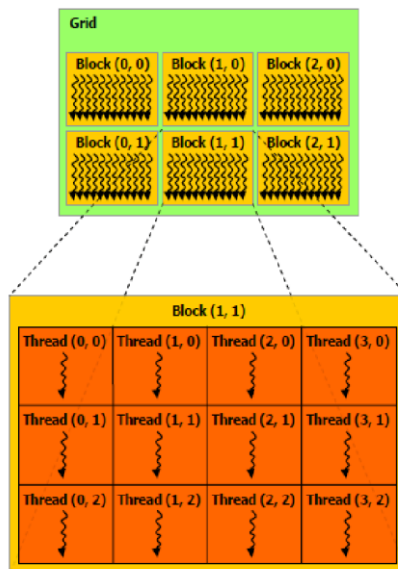
1. **Streaming Processor**(viết tắt là SP hay còn gọi quen thuộc là core) là những đơn vị xử lý chính trên GPU và ***có khả năng thực hiện các phép tính đồng thời trên nhiều dữ liệu*** => giống như các bạn học sinh (1 học sinh là 1 SP) => chúng ta có càng nhiều SP (học sinh) thì số lượng công việc được xử lý ***đồng thời*** càng nhiều
2. **Streaming Multiprocessor**(viết tắt là SM, có 1 số tài liệu ghi là multiprocessor) có thể hiểu đơn giản là tập hợp của các SP, tức là 1 SM có nhiều SP ( SM giống như lớp học vậy)

Số lượng SM và SP sẽ tùy thuộc vào máy tính của mỗi người ( số lượng cố định)

Logical ứng với thread, block, grid

- Thread hiểu đơn giản là công việc ( 1 thread là 1 công việc)

- Block là tập hợp của nhiều các thread ( 1 núi công việc **nhưng tối đa là 1024 threads** vì cơ chế của máy tính là vậy áp dụng cho mọi máy tính)
- Grid là tập hợp của các Block
- **Công việc ở đây là data, tức là ở mỗi bài toán chúng ta sẽ có số lượng data khác nhau (vì vậy nên mình mới nói là không xác định trước được)**



- Như trong hình chúng ta có thể thấy có 6 blocks và mỗi block có 12 threads
- Các con số (0,0) (0,1) là index dùng để xác định block nằm vị trí nào và thread nằm vị trí nào, giống như ma trận vậy  **$a[1][2]$**  .
- Ở đây các bạn sẽ có câu hỏi là tại sao chúng ta lại chia các thread ra những block riêng mà không gộp lại thành 1 block siêu lớn đi cho dễ. Nếu chúng ta làm vậy chúng ta sẽ vi phạm **QUY TẮC 1 là Mỗi lớp học chỉ nhận tối đa 1024 công việc** vậy nên chúng ta mới cần chia nhỏ các thread ( tức là số lượng công việc) ra thành các block.
- Và 1 điểm cộng lớn khi chia nhỏ các thread thành các block là vì **QUY TẮC 2** : nếu chúng ta có 1024 thread thì chỉ cần 1 block là đủ nhưng tại 1 thời điểm nó chỉ xử lý đc (32\* 1) công việc ( giả sử số warp =1 ) thì chúng ta phải đợi nó xử lý xong 32 cái đầu rồi mới tới 32 cái sau và lặp đi lặp lại 1 cách tuần tự.
- Nếu lúc này chúng ta chia ra thành 32 block, mỗi block 32 thread (32 \* 32=1024) thì có phải chỉ tại 1 thời điểm nó đã xử lý hết 1024 thread rồi không 32\* 1\* 32(32\* số warp \* N )

WARP: physical và logical

SM(s) là các lớp học, 1 SP là 1 học sinh(core), 1 Thread là 1 công việc, Block là 1 núi công việc, ở đây bạn có thể tưởng tượng block là 1 cái hộp chứa các công

việc(thread) cần xử lý. Và mỗi SM sẽ xử lý 1 số lượng blocks (tùy thuộc vào số lượng data để chia ra các block) $\Rightarrow$  1SP có thể xử lý nhiều hơn 1 thread (1 học sinh có thể làm nhiều hơn 1 công việc)

Ở đây sẽ có 1 vấn đề là làm sao để phân phối các công việc(blocks) cho từng lớp học(SM) vì SM,Block là 2 khái niệm riêng biệt(physical và logical) nên chúng nó sẽ không thể tương tác trực tiếp mà phải thông qua trung gian gọi là WARP, vậy thì warp là gì và con số 32 có liên quan gì?

- **Physical** và **logical** là 2 khái niệm riêng biệt nên sẽ không thể tương tác trực tiếp được nên phải tương tác giao tiếp thông qua trung gian gọi là **WARP**. Warp ở đây vừa là physical và là logical.
- Quay lại ví dụ về trường học, thì warp ở đây là các tổ trưởng của mỗi lớp (số lượng warp (tổ trưởng) sẽ tùy thuộc vào từng máy tính. Các tổ trưởng (các warps) có 2 nhiệm vụ
  - o là đi lấy các blocks để đem về cho tổ mình xử lý  $\Rightarrow$  ở đây các blocks đã nằm sẵn trong từng lớp học (SM)
  - o Sau khi warp mang các block về cho tổ của mình thì warp sẽ làm thêm 1 nhiệm vụ nữa là phân chia công việc cho các thành viên trong tổ. Và mỗi lần phân chia tối đa là 32 công việc 1 lần. Sau khi 32 công việc đó được xử lý xong mới đưa tiếp 32 công việc tiếp theo để xử lý. Tức là tại mỗi thời điểm 1 warp phân phối 32 thread mà mỗi lớp sẽ có nhiều warp (tổ trưởng) nên số công việc tại 1 thời điểm =  $(32 * \text{số warp})$  công việc
    - $\Rightarrow$  1 warp chỉ có thể phân phối tối đa 32 threads thì là do chức năng của máy tính là vậy, áp dụng cho mọi máy tính
- **Warp (Physical)** : tức là các bạn tổ trưởng này sẽ dẫn dắt các bạn học sinh trong tổ của mình hay nói cách khác warp kiểm soát các SP trong việc xử lý công việc ( ví dụ bạn A ĐƯỢC PHÂN CÔNG làm công việc A, bạn B ĐƯỢC PHÂN CÔNG làm công việc B ....)
- **Warp(Logical)** : tức là các bạn tổ trưởng này sẽ kiểm soát số lượng thread ( công việc)
- 1 lưu ý nhỏ là mình lấy ví dụ warp là tổ trưởng nhưng sẽ không được tính là 1 thành viên trong lớp, tức là 1 lớp học có 50 bạn học sinh và 5 tổ trưởng thì SP vẫn là 50 chứ không phải 55

## 2.2 Các bộ nhớ trong GPU

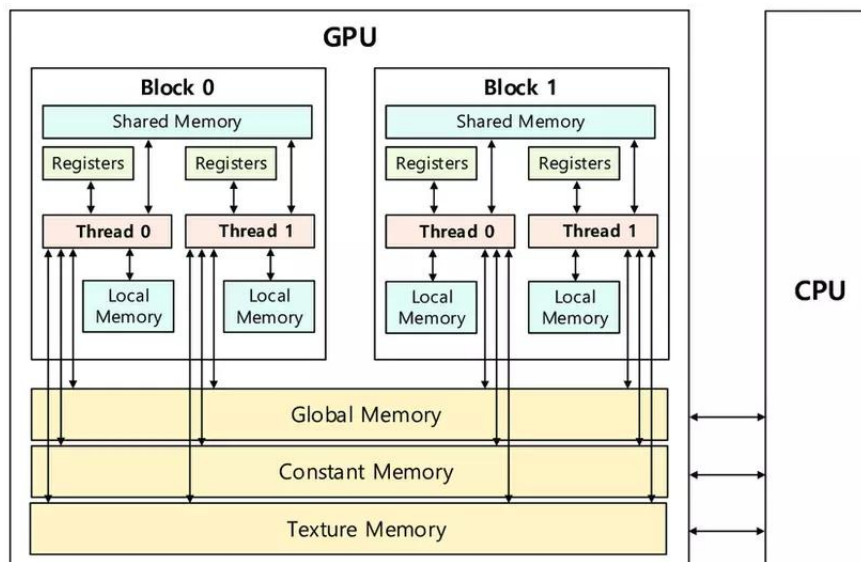
- Chúng ta có thể **tùy ý sử dụng bộ nhớ nào mà ta thích** (tức là khi khởi tạo 1 giá trị, 1 biến bất kỳ ta có thể chỉ định nó được lưu vào bộ nhớ nào 1

cách tùy ý ) chứ không phải là để máy tính tự quyết định bộ nhớ nào sẽ được dùng và nhờ vào việc đó chúng ta có thể tận dụng triệt để các bộ nhớ để tối ưu chương trình.

- trước khi đi qua các bộ nhớ trong GPU thì chúng ta cần biết là khi **nói về bộ nhớ**, ta thường chia thành hai loại chính: **bộ nhớ vật lí (physical memory)** và **bộ nhớ logic (logical memory)**.
- **Bộ nhớ vật lí (Physical Memory):** Đây là bộ nhớ thực sự trên phần cứng của máy tính. Nó bao gồm các thanh RAM và các thiết bị lưu trữ như ổ cứng (HDD/SSD). Bộ nhớ vật lí là nơi dữ liệu và chương trình được lưu trữ trực tiếp và có thể truy cập nhanh chóng từ bộ xử lý.
- **Bộ nhớ logic (Logical Memory - hay còn có cái tên quen thuộc là virtual memory):** Đây là không gian địa chỉ mà hệ điều hành và các chương trình có thể truy cập. Bộ nhớ logic không nhất thiết phải tương ứng với bộ nhớ vật lí một cách trực tiếp. Hệ điều hành thường quản lý việc ánh xạ (mapping) giữa địa chỉ logic và địa chỉ vật lí. Nó giúp quản lý việc cấp phát và quản lý bộ nhớ cho các chương trình chạy trên hệ thống.

⇒ Có thể hiểu 1 cách đơn giản là khi ta code thì chúng ta sẽ tương tác với logical memory, và khi code xong các dữ liệu đang nằm trên logical memory sẽ được mapping qua physical memory (tức là máy tính sẽ hoạt động ở physical memory) .

Góc nhìn Logical



**Block, Thread là logical** và vì là cơ chế SIMT (Single Instruction, Multiple Threads **một lệnh duy nhất** được phát hành và thực thi đồng thời bởi **nhiều thread**) nên chúng ta cần phải biết các Thread, Block được phân bố như thế nào trong các bộ nhớ của GPU ( logical memory )

Ở đây chúng ta sẽ có 1 khái niệm khá thân thuộc là **scope ( phạm vi truy cập )**: đóng một vai trò quan trọng trong việc hiểu cách các tài nguyên như Thread và Block được phân bố và quản lý trong bộ nhớ logic của GPU.

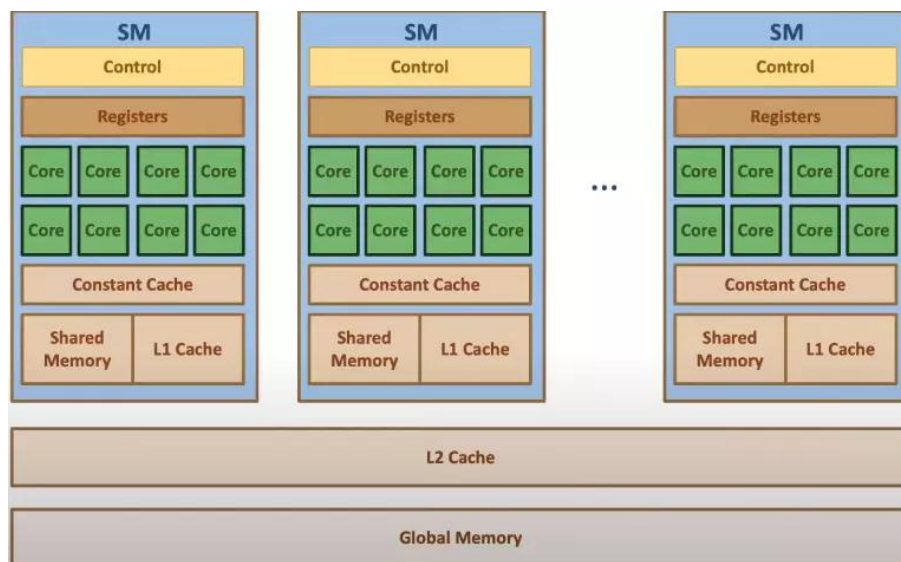
**Local Memory: Mỗi Thread** có thể sử dụng **local memory riêng**, nơi mà nó có thể lưu trữ các biến tạm thời. Đây là **phạm vi truy cập có phạm vi nhỏ nhất và chỉ dành riêng cho mỗi Thread**.

**Shared Memory: Các Thread trong cùng một Block** có thể **chia sẻ dữ liệu thông qua shared memory**. Điều này cho phép các Thread trong cùng một Block giao tiếp và truy cập dữ liệu nhanh hơn so với việc truy cập global memory.

**Global Memory: Đây là bộ nhớ lớn nhất** trong GPU và **có thể truy cập bởi tất cả các Thread trên mọi Block**. Tuy nhiên, truy cập vào global memory thường chậm hơn so với các loại bộ nhớ khác, do đó cần phải tối ưu để tránh hiệu năng bị giảm.

**Texture Memory và Constant Memory: Đây là các loại bộ nhớ đặc biệt trên GPU**, được tối ưu cho việc truy xuất các loại dữ liệu cụ thể như **hình ảnh (texture)** hoặc các giá trị hằng số. Các memory này có thể truy cập bởi tất cả các Thread trên mọi Block

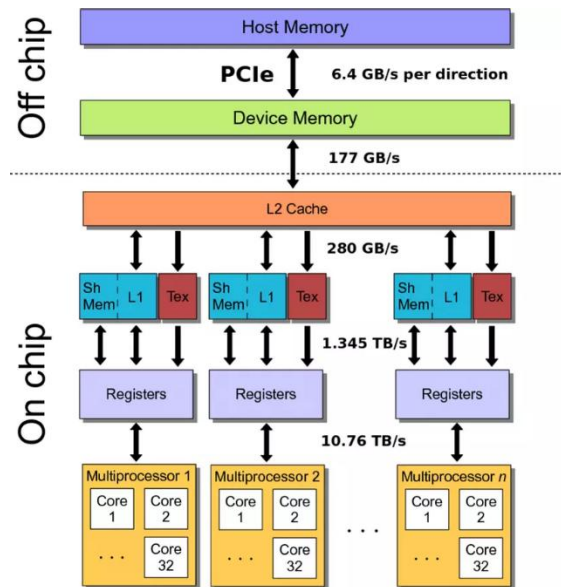
Góc nhìn Physical



Nó cũng khá giống với Block và Thread nhưng ở đây là SM và SP. **Mỗi SM** sẽ sở hữu riêng cho mình các **shared/Cache/Constant/Register memory**. Và các **SM** sẽ dùng chung 1 **global memory**.

**L2 Cache trong GPU** là một lớp bộ nhớ đệm (cache) cấp hai nằm giữa **Global Memory** (bộ nhớ toàn cục) và các **Streaming Multiprocessors (SM)**. L2 cache giúp tăng tốc độ truy cập dữ liệu từ global memory bằng cách lưu trữ tạm thời các dữ liệu mà các SM có thể cần dùng lại. Điều này giúp giảm thời gian chờ đợi dữ liệu, giảm tải cho global memory và tăng hiệu suất tổng thể của GPU.

Bandwidth of memory



PCIe

**CPU( host )** và **GPU ( device )** là 2 thành phần riêng biệt nên vì vậy chúng sở hữu các **memory riêng biệt** và không thể truy cập trực tiếp mà phải **copy data** qua lại **thông qua PCIe** (hay còn gọi với cái tên quen thuộc là bus )

Một trong những yếu tố chính để quyết định là liệu chúng ta có nên đưa data từ CPU về GPU để tính toán hay không là do PCIe vì như trong hình ta có thể thấy **PCIe sở hữu tốc độ transfer data chậm nhất**.

Để giải quyết vấn đề khi copy 1 lượng lớn data từ CPU về GPU thì NVIDIA đã đưa ra 3 phương pháp:

- Sử dụng Unified memory
- Sử dụng Pinned memory
- Phương pháp streaming ( hay còn gọi là hidden latency )



## Global memory

- **Global memory ( hay còn gọi là device memory )** là memory sở hữu **bộ nhớ lớn nhất** nằm trong GPU và vì là lớn nhất nên cũng là bộ nhớ có **tốc độ truy cập chậm nhất** chỉ sau PCIe
- **Global memory** cũng **giống** như **RAM ở CPU** vậy, khi chúng ta khởi tạo 1 giá trị bất kì nào đó ở GPU mà không chỉ định nó sẽ được lưu trữ vào bộ nhớ nào thì sẽ mặc định lưu vào Global memory

**Từ đây có thể thấy mục đích chính của global memory là dùng để lưu trữ dữ liệu lớn**

## Shared/Cache memory

## Texture Memory và Constant Memory

## 2.3 Host và Device trong CUDA

Trong CUDA, Host và Device là hai khái niệm quan trọng mô tả các thành phần khác nhau trong môi trường lập trình GPU.

- **Host:** Là CPU và bộ nhớ hệ thống (main memory) mà chương trình chạy trên đó. Tất cả các chương trình CUDA đều chạy trên host trước khi chuyển một phần công việc cho GPU.
- **Device:** Là GPU và bộ nhớ của nó (device memory), nơi thực hiện các phép toán song song và xử lý dữ liệu tính toán cao.

Trong một chương trình CUDA, mã của bạn sẽ được chia thành hai phần:

1. Mã chạy trên host (CPU): Đảm nhận việc điều khiển, thiết lập và quản lý các tác vụ tính toán, đồng thời xử lý kết quả.
2. Mã chạy trên device (GPU): Thực hiện các phép toán tính toán song song, xử lý dữ liệu một cách hiệu quả nhờ hàng nghìn thread.

## Cấu trúc chương trình CUDA

Một chương trình CUDA cơ bản bao gồm hai phần chính:

1. Phần mã chạy trên host:
  - Thiết lập dữ liệu đầu vào.
  - Gọi các kernel từ host để gửi chúng tới device.
  - Quản lý bộ nhớ và đồng bộ hóa các tác vụ.

## 2. Phần mã chạy trên device (kernel):

- Được chạy trên GPU, thực thi song song qua các thread.
- Các kernel function này sẽ được gọi từ host.

### 2.4 Các hàm gửi dữ liệu giữa Host và Device trong CUDA

Hàm	Mô tả	Cú pháp	Loại Sao Chép
<b>cudaMemcpy</b>	Sao chép dữ liệu giữa Host và Device.	cudaMemcpy(void *dst, const void *src, size_t count, cudaMemcpyKind kind)	cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, cudaMemcpyHostToHost
<b>cudaMalloc</b>	Cấp phát bộ nhớ trên Device.	cudaMalloc(void **devPtr, size_t size)	Cấp phát bộ nhớ trên Device.
<b>cudaFree</b>	Giải phóng bộ nhớ trên Device.	cudaFree(void *devPtr)	Giải phóng bộ nhớ trên Device.
<b>cudaHostAlloc</b>	Cấp phát bộ nhớ có thể chia sẻ giữa Host và Device (Pinned memory).	cudaHostAlloc(void **pHost, size_t size, unsigned int flags)	Cấp phát bộ nhớ trên Host có thể chia sẻ với Device.
<b>cudaHostGetDevicePointer</b>	Lấy con trỏ Device cho bộ nhớ trên Host.	cudaHostGetDevicePointer(void **devPtr, void *hostPtr, unsigned int flags)	Chia sẻ con trỏ giữa Host và Device.



<b>cudaMemcpyAsync</b>	Sao chép dữ liệu bất đồng bộ giữa Host và Device.	cudaMemcpyAsync(void *dst, const void *src, size_t count, cudaMemcpyKind kind, cudaStream_t stream)	Sao chép bất đồng bộ giữa Host và Device.
<b>cudaMemset</b>	Thiết lập giá trị bộ nhớ trên Device.	cudaMemset(void *devPtr, int value, size_t count)	Thiết lập giá trị bộ nhớ trên Device.