# Kiến thức cần có trước khi học CUDA

* Lập trình C/C++ cơ bản:
  + **Lập trình đa luồng với pthreads/OpenMP** (nếu có thể).

## Kiến trúc máy tính & Hệ điều hành

Bộ nhớ trong máy tính: RAM, Cache, VRAM

A black and white text on a black background

AI-generated content may be incorrect.

* Cache :
  + là bộ nhớ tạm thời giúp lưu trữ dữ liệu được truy xuất thường xuyên để tăng tốc độ xử lý và giảm thời gian chờ. Nó giúp CPU, RAM, ổ cứng, trình duyệt web,... hoạt động nhanh hơn.
  + Các loại: CPU Cache, Disk Cache, RAM Cache, DNS Cache, Application Cache
  + Cách hoạt động:
    - Khi ta yêu cầu dữ liệu, hệ thống sẽ tìm trong Cache trước
    - Nếu có dữ liệu trong Cache => tải nhanh hơn
    - Nếu dữ liệu không có trong Cache => Hệ thống sẽ đi lấy dữ liệu gốc về
    - Dữ liệu mới sẽ được lưu vào Cache để dùng lần sau
  + Ví dụ:
    - Mở 1 trang web lần đầu, trình duyệt phải tải toàn bộ hình ảnh, CSS, Js. Lần truy cập sau, trình duyệt lấy từ cache
    - Khi chơi game, dữ liệu đồ họa được lưu vào VRAM cache để hiển thị nhanh hơn
  + Ưu điểm:
    - Tăng tốc độ xử lý
    - Tiết kiệm tài nguyên và giảm tải
    - Giảm băng thông mạng đối với trình duyệt
  + Nhược điểm:
    - Nhiều cache làm chậm máy
    - Cần xóa cache định kỳ để tránh lỗi và giải phóng dung lượng

Multithreading và Multiprocessing

* Đa tiến trình:
  + A process is an instance of a computer program that is being executed
    - Dữ liệu ở hard drive sẽ được load vào cpu và ram
    - Context: collection of data about process like (memory address, state) which allow processor to suspend or hold the execution of a process and restart the execution later.
  + Mỗi tiến trình có bộ nhớ riêng, tài nguyên riêng, ko ảnh hưởng bởi tiếng trình khác
  + Đa tiến trình là nhiều tiến trình chạy độc lập song song để tận dụng đa core của CPU
  + Ưu điểm:
    - Chạy song song trên nhiều nhân CPU
    - Khi 1 tiến trình lỗi, các tiến trình khác ko bị ảnh hưởng
  + Nhược điểm:
    - Tốn nhiều bộ nhớ RAM do mỗi tiến trình có bộ nhớ riêng
    - Việc giao tiếp giữa các tiến trình phức tạp
* Đa luồng:
  + Khi khởi chạy ứng dụng, hệ thống tạo ra 1 tiến trình. Tiến trình sẽ khởi chạy 1 thread chính. Là entry point cho chương trình. Từ thread chính sinh ra các thread con
  + Các thread có thể chạy song song, cùng chia sẻ bộ nhớ với tiến trình cha.
  + Đa luồng là chạy nhiều luồng bên trong 1 tiến trình để thực hiện nhiều tác vụ cùng lúc. Các tác vụ này được chia nhỏ từ tác vụ lớn, chúng có thể thực hiện độc lập và gộp lại thành kết quả tổng.
  + Các luồng có thể chia sẻ dữ liệu với nhau
  + Nếu 1 luồng bị lỗi có thể làm crash toàn chương trình
  + Lập trình đa luồng phức tạp do phải xử lý tranh chấp tài nguyên
* Currency và parallel:
  + Currency: trên 1 core, thực hiện đồng thời nhiều tác vụ. Tốn context switching để chuyển đối giữa các tác vụ
  + Parallel: có nhiều core nên simultaneous trên đa luồng trên đa nhân có thể thực hiện parallel.
    - Gồm :
      * task parallel: different task on same or different data
      * data parallel: same task on different data

## Cách CPU và GPU giao tiếp trên PC và jetson nano

* Trên máy tính thông thường dùng GPU GeForce thì cpu và gpu giao tiếp qua Pcle
  + GPU GeForce có bộ nhớ VRAM riêng
  + PCIe : Peripheral Component Interconnect Express
    - Là bus tốc độ cao kết nối các linh kiện phần cứng như CPU, GPU,RAM,…
      * Bus là hệ thống dây dẫn truyền thông tin giữa các thành phần
  + Các giao tiếp qua Pcle:
    - Cpu gửi dữ liệu cho cpu qua pcle
    - Gpu xử lý và gửi kết quả về cpu
    - Khi cần dữ liệu từ cpu, gpu phải sao chép từ RAM sang VRAM qua Pcle
    - Tránh truyền dữ liệu quá nhiều giữa RAM và VRAM do Pcle chậm so với VRAM
  + Có các công nghệ giúp giao tiếp tốt hơn giữa cpu và gpu:
    - Cho phép cpu truy cập toàn bộ vram
    - Gpu đọc trực tiếp từ ssd mà ko cần qua cpu
    - Tạo cầu nối trực tiếp giữa 2 gpu
* Trên jetson nano, cpu và gpu giao tiếp qua **unified memory**
  + GPU trên jetson nano ko có VRAM, mà dùng chung RAM với CPU
  + Cpu-gpu giao tiếp không cần Pcle.
  + CPU và GPU chia sẻ chung RAM ( CPU chỉ cần ghi vào RAM, GPU có thể đọc dữ liệu từ RAM

## Lập trình Python

* + NumPy → Tính toán ma trận trên CPU
  + CuPy → Tương tự NumPy nhưng chạy trên GPU
  + PyCUDA → Gọi trực tiếp CUDA từ Python.

# Hiểu về CUDA và GPU

Graphics Processing Unit thiết kế để xử lý đồ họa, tính toán song song trong các ứng dụng khoa học, AI, và mô phỏng.

A screenshot of a black screen

AI-generated content may be incorrect.

* CPU phù hợp với xử lý từng tác vụ phức tạp một cách chính xác.
* GPU phù hợp với các bài toán có thể chia nhỏ và tính toán đồng thời

CUDA (Compute Unified Device Architecture) là mô hình lập trình của NVIDIA tận dụng sức mạnh của GPU để tăng tốc xử lý dữ liệu.

Ưu điểm của CUDA:

* Tận dụng sức mạnh của GPU để xử lý nhanh hơn CPU nhiều lần
* Lập trình dễ dàng với C, C++, Python (thông qua thư viện CuPy)
* Ứng dụng rộng rãi trong AI bigdata
* Coder tùy ý sử dụng bộ nhớ mong muốn

Ứng dụng của CUDA:

* **Deep Learning** (TensorFlow, PyTorch)
* **Xử lý hình ảnh & video** (OpenCV, FFmpeg)
* **Mô phỏng vật lý** (CFD, FEA, Rendering)
* Tài chính & phân tích dữ liệu

Cấu trúc chi tiết của GPU NVIDIA:

* Streaming Multiprocessor:
  + Là đơn vị xử lý chính của GPU, mỗi SM gồm nhiều CUDA Cores
  + 1 GPU gồm nhiều SMs, mỗi SM có thể chạy nhiều warps cùng lúc
  + SM cũng chứa Shared Memory, giúp các threads trong cùng block chia sẻ dữ liệu nhanh hơn
* CUDA cores:
  + Là đơn vị xử lý nhỏ nhất của GPU, tương đường ALU của CPU
  + GPU chứa hàng nghìn CUDA Cores
* Warp. Thread, Block, Grid
  + Cuda sử dụng mô hình song song dựa trên Threads, Blocks, Grids
    - Thread: 1 luồng xử lý đơn lẻ
    - Warp: nhóm 32 thread chạy song song
    - Block: nhóm các thread được xử lý trong 1 SM
    - Grid: Nhóm các blocks launch for a kernel

Các loại bộ nhớ trong GPU:

* Global Memory:
  + Dung lượng lớn (~GBs), truy cập chậm
  + Dùng để lưu dữ liệu chung cho toàn bộ Grid.
  + Sử dụng **memory coalescing** để truy xuất hiệu quả.
* Shared Memory:
  + Bộ nhớ chia sẻ giữa các threads trong cùng một Block (~48KB/SM).
  + Dùng shared memory để giảm số lần truy cập Global Memory.
* Register Memory:
  + Dùng để lưu biến cục bộ cho từng thread.
* Constant Memory và Texture Memory:
  + **Constant Memory**: Chỉ đọc, dùng để lưu hằng số, truy cập nhanh
  + Texture Memory: Dùng trong xử lý ảnh và video

A black and white screen with white text

AI-generated content may be incorrect.

## Luồng thực thi của 1 chương trình sử dụng GPU

1. Chuẩn bị dữ liệu trên CPU
2. Cấp phát bộ nhớ trên GPU
3. Sao chép dữ liệu từ CPU sang GPU
4. Khởi chạy Kernel function trên GPU
5. Các Threads trên GPU thực hiện tính toán song song
6. Chuyển kết quả từ GPU sang CPU
7. Giải phóng bộ nhớ trên GPU

A screenshot of a computer program

AI-generated content may be incorrect.

## **Cài đặt và thiết lập môi trường**

Kiểm tra version cuda:

/usr/local/cuda/bin/nvcc –version

Hoặc

cat /usr/local/cuda/version.txt

* Nếu 2 lệnh trên check mà có cuda version mà nvcc –version vẫn báo “nvcc: command not found” thì tức chưa thêm nvcc vào biến môi trường:
  + Kiểm tra CUDA đã được thêm vào PATH chưa:

echo $PATH | grep cuda

* + Nếu chưa, cần thêm thủ công:

nano ~/.bashrc

và thêm các dòng sau vào cuối file:

export PATH=/usr/local/cuda/bin:$PATH

export LD\_LIBRARY\_PATH=/usr/local/cuda/lib64:$LD\_LIBRARY\_PATH

Lưu file (Ctrl X, rồi Y, rồi Enter),sau đó lưu thay đổi bằng lệnh:

source ~/.bashrc

Kiểm tra lại:

Nvcc –version

Kiểm tra gcc:

gcc –version

Thu đươc gcc 7.5, cuda 10.2

A screenshot of a table

AI-generated content may be incorrect.

Cần nâng cấp lên gcc8 để tương thích với cuda10.2

Nâng cấp lên gcc8:

Bước 1: Cập nhật danh sách gói

sudo apt update && sudo apt upgrade -y

Bước 2: Cài đặt GCC 8 và G++ 8

sudo apt install gcc-8 g++-8 -y

Bước 3: Kiểm tra phiên bản mới đã cài đặt

gcc-8 –version

Bước 4: Đặt GCC 8 làm mặc định

sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-7 50

sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-8 100

sudo update-alternatives --config gcc

Bước 5: Kiểm tra lại phiên bản GCC mặc định

gcc –version

## Lập trình CUDA cơ bản

🔹 **Cấu trúc chương trình CUDA**:

* **Host** : Là CPU, nơi quản lý bộ nhớ và điều khiển việc thực thi các kernel trên GPU
* **Device** : Là GPU, nơi thực hiện các tính toán song song trên nhiều thread.
* **Kernel functions** – Hàm chạy trên GPU
* **Memory Management** – Quản lý bộ nhớ trong CUDA
* **Note**:
  + **Code chạy trên CPU là code C/C++ thông thường**
  + **Code chạy trên GPU là các kernel function được xác định bằng từ khóa \_\_global\_\_** b
* Các thành phần chính của chương trình CUDA:
  + Trước khi sử dụng dữ liệu trên GPU, cần cấp phát bộ nhớ trên GPU bằng cudaMalloc()
  + Sao chép dữ liệu từ Host sang Device bằng cudaMemcpy()
  + Gọi kernel trên GPU
    - 1 kernel là 1 hàm thực thi trên nhiều thread song song. Kernel được gọi với cú pháp kernel\_name<<<gridDim, blockDim>>>(arguments)
  + Sao chép dữ liệu từ Device sang Host
    - Sau khi tính toán xong, kết quả từ GPU được sao chép lại về CPU
  + Giải phóng bộ nhớ GPU bằng cudaFree()

### Chương trình đơn giản

* Tính bình phương các số:
  + Bước 1: tạo file

nano test.cu

* + Bước 2: paste

#include <stdio.h>

#include <cuda\_runtime.h>

#define N 10

\_\_global\_\_ void a() {

printf("%d\n", threadIdx.x \* threadIdx.x);

}

int main() {

a<<<1, N>>>(); // Chạy kernel với 1 block và N threads

cudaDeviceSynchronize(); // Đồng bộ GPU trước khi kết thúc chương trình

return 0;

}

Sau đó Ctrl X rồi Y rồi Enter

* + Bước 3: Biên dịch

nvcc test.cu -o test

* + Bước 4: Run

./test

**4. Quản lý bộ nhớ và tối ưu hiệu suất**

🔹 Các loại bộ nhớ trong CUDA:

* **Global Memory**
* **Shared Memory** (bộ nhớ chung trong block)
* **Constant Memory**
* **Texture & Surface Memory**

🔹 Tối ưu hiệu suất:

* Giảm **global memory access**
* Tận dụng **shared memory**
* Sử dụng **memory coalescing** để tăng tốc độ đọc ghi

**5. Lập trình CUDA nâng cao**

🔹 **Streams và Concurrent Execution** – Tận dụng nhiều kernel chạy song song  
🔹 **Atomic Operations** – Giảm điều kiện race khi nhiều thread truy cập dữ liệu  
🔹 **Thrust Library** – Thư viện hỗ trợ tính toán CUDA nhanh chóng  
🔹 **CUDA Graphs** – Tăng tốc độ xử lý bằng cách tối ưu các task GPU

**6. Ứng dụng CUDA vào AI và HPC**

🔹 Tích hợp với **TensorFlow/PyTorch** để tăng tốc Deep Learning  
🔹 Ứng dụng vào **Big Data Processing** và **Simulation**  
🔹 Tối ưu thuật toán **Machine Learning với CUDA**

📌 **Ví dụ với CuPy (CUDA + Python)**:

python

CopyEdit

import cupy as cp

a = cp.array([1, 2, 3, 4, 5])

b = cp.array([10, 20, 30, 40, 50])

c = a + b

print(c)

**7. Thực hành dự án thực tế**

🔹 Tối ưu thuật toán tính toán ma trận lớn với CUDA  
🔹 Tăng tốc xử lý ảnh/video bằng CUDA  
🔹 Ứng dụng vào tài chính, sinh học tính toán

**Tóm tắt lộ trình**

✅ Hiểu cơ bản về GPU và CUDA  
✅ Viết chương trình CUDA đầu tiên  
✅ Quản lý bộ nhớ GPU  
✅ Tối ưu hiệu suất CUDA  
✅ Ứng dụng vào AI & HPC  
✅ Thực hành qua các dự án thực tế

**Lộ trình học CUDA**

**1. Kiến thức nền tảng**

* Lập trình C/C++ cơ bản
* Kiến thức về hệ thống máy tính và kiến trúc máy tính
* Hiểu biết về lập trình song song
* Kiến thức về xử lý đồ họa GPU

**2. Làm quen với CUDA**

* Tìm hiểu về kiến trúc GPU NVIDIA
* Cài đặt CUDA Toolkit và các công cụ cần thiết
* Hiểu về mô hình lập trình CUDA (kernels, threads, blocks, grids)
* Làm quen với các khái niệm cơ bản: warp, streaming multiprocessors

**3. Lập trình CUDA cơ bản**

* Viết chương trình CUDA Hello World đầu tiên
* Quản lý bộ nhớ (global memory, shared memory, constant memory)
* Hiểu về cơ chế đồng bộ hóa và các kiểu biến
* Thực hiện các thuật toán đơn giản (vector addition, matrix multiplication)

**4. Tối ưu hóa hiệu suất**

* Tối ưu hóa truy cập bộ nhớ
* Sử dụng shared memory hiệu quả
* Tránh divergence trong warp
* Tối ưu hóa occupancy

**5. Lập trình CUDA nâng cao**

* Streams và xử lý bất đồng bộ
* Dynamic parallelism
* Unified Memory
* CUDA Graphs

**6. Thư viện và frameworks**

* cuBLAS (Basic Linear Algebra Subroutines)
* cuDNN (Deep Neural Networks)
* Thrust (thư viện template C++ song song)
* NCCL (NVIDIA Collective Communications Library)

**7. Các chủ đề chuyên sâu**

* Multi-GPU programming
* Liên kết với các framework học máy (TensorFlow, PyTorch)
* Profiling và debugging với NVIDIA tools (Nsight, nvprof)
* CUDA trong các ứng dụng thực tế

# Lập trình song song

Bạn sẽ thu được : Thinking in parallel và có basic understanding of GPU architecture đẻ viết parallel program

## Giới thiệu về CPU-GPU

* CPU:
  + Thiết kế phức tạp và nặng, tối ưu hóa cho chương trình tuần tự
  + Tối ưu hóa latency cho ít thread => gọi là tối ưu hóa short latency
  + 1 cpu chất lượng dựa trên:
    - Clock speed : tốc độ xung nhịp
    - Cores: số lượng các đơn vị xử lý độc lập trong CPU
  + Vấn đề gặp phải khi sd cpu 1 core:
    - Không thể tăng công suất của CPU lên mãi do phải tăng điện áp theo công thức: P = UxI
    - Độ trễ do đọc ghi từ bộ nhớ
* Sinh ra khái niệm dual core và gpu
* GPU:
  + Thiết kế nhẹ hơn cpu, được tối ưu hóa cho tác vụ song song theo dữ liệu
  + xử lý số lượng lớn threads nhẹ cùng lúc => tối đa hóa throughput
  + Logic điều khiển đơn giản hơn, tập trung vào tối đa hóa throughput của các chương trình song song
  + Gọi là hardware accelerator
  + 2 tính năng quan trọng mô tả GPU capability:
    - Num of cuda cores
    - Memory size
  + 2 metric đo GPU performance:
    - **Memory bandwidth**
    - **Peak** **computational performance**
* CPU thread vs GPU thread:
  + CPU thread: heavyweight. CPU phải swap thread vào và ra khỏi bộ thực thi để đa luồng => context switch tốn
  + GPU thread: lightweight. Có hàng nghin thread để switch mà ko ảnh hưởng lớn về performance
* Heterogeneous architecture:
  + Là nền tảng kết hợp cpu với gpu
  + Do GPU chưa là nền tảng độc lập mà đóng vai trò như co-processor => GPU cần kết hợp với CPU thông qua bus PCI => CPU được gọi là host, GPU gọi là device
  + Heterogeneous application gồm:
    - Host code: run on CPU, manage the env, code, data for device
    - Device code
* Parallelism:
  + Chia nhỏ nhiệm vụ lớn thành các nhiệm vụ nhỏ có thể xử lý đồng thời
  + Có 2 loại song song cơ bản:
    - Task parallelism:
      * Phân phối các function có thể hoạt động độc lập trên nhiều cores.
    - Data parallelism:
      * Phân phối data vào nhiều core cùng lúc.
* CUDA Programming :
  + Cuda là nền tảng tính toán song song của Nvidia thông qua GPU trên ngôn ngữ C/C++
  + Cho phép coder kiểm soát sự tương tác giữa ứng dụng và nền tảng phần cứng như quản lý threads, bộ nhớ,… =>
  + Support sự kết hợp việc thực thi của CPU + GPU cho 1 application.

A group of rectangular objects with text

AI-generated content may be incorrect.

* + giải quyết các bài toán song song theo dữ liệu. Tức ánh xạ các data elements vào các parallel threads.
  + Cuda cung cấp 2 api để quản lý gpu device và tổ chức các thread:
    - Cuda driver api
    - **Cuda runtime api**
* First step của data parallel program là partition data across threads, with each thead working on a portion of the data.
* Có 2 cách chia dữ liệu:
  + Block partitioning:
    - many consecutive elements of data are chunked together to a block
    - 1 thread xử lý 1 block tại 1 thời điểm
    - Each thread takes one portion of the data, usually an equal portion ofthe data.
  + Cyclic partitioning: Each thread takes more than one portion of the data

## Cài đặt môi trường code CudaC

* Cuda version cần tương thích với gcc version khi code với cpp
* NVIDIA cung cấp một môi trường cho C và C++ :
  + NVIDIA Nsight™ – môi trường phát triển tích hợp (IDE)
  + CUDA-GDB – trình gỡ lỗi dòng lệnh
  + Visual and command line profiler for performance analysis– hỗ trợ cả giao diện đồ họa và dòng lệnh
  + CUDA-MEMCHECK – công cụ kiểm tra lỗi bộ nhớ
  + GPU device management tools

## Cách thức hoạt động của máy tính

* Nguyên lý xử lý dữ liệu của máy tính:
  + Xử lý xong data này rồi mới xử lý data tiếp theo chứ không cùng xử lý các data tại 1 thời điểm
  + Các data ko xử lý trực tiếp tại chỗ nó được Read mà phải copy data qua 1 khu vực memory nào đó để xử lý

## Các thuật ngữ cơ bản

* Latency is the time it takes for an operation to start and complete, dùng microseconds
* Bandwidth : amount of data that can be processed per unit of time, MB/s, GB/s
* Throughput : amount of operations that can be processed per unit of time, gflops
* Physical : những thứ thấy được, số lượng cố định. Ví dụ học sinh
* Logical : những thứ không thấy được, nhưng hình dung được, số lượng ko cố định. Ví dụ số công việc
* Locality: đề cập việc reuse data nhằm giảm memory access latency
  + Temporal locality: reuse data and resource within a small time duration
  + Spatial locality: use data elements nằm gần nhau trong bộ nhớ
* Streaming Processor (SP hay core) là đơn vị xử lý chính trên GPU và **có khả năng thực hiện các phép toán đồng thời trên nhiều dữ liệu**. Ví dụ 1 học sinh như 1 SP
* Streaming Multiprocessor (SM) là tập các SP, tức 1 SM có nhiều SP.

Physical ứng với SM và SP (số lượng cố định)

* SM và SP ứng với Physical
* Logical ứng với thread, block, grid:
  + Block là tập các thread, max = 1024 thread
  + Grid là tập các block
  + Thread là đơn vị xử lý mỗi công việc.
  + Công việc là data. Mỗi bài toán có số lượng data khác nhau nên thread ko xác định trước được, block cũng vậy
  + SM(s) là những lớp học, 1 SP là 1 học sinh
  + 1 Thread là 1 công việc, 1 Block là 1 nùi công việc
  + Mỗi SM sẽ xử lý 1 số lượng blocks (tùy vào số lượng data để chia ra các block
* 1 SP có thể xử lý nhiều hơn 1 thread
* Vấn đề: làm sao phân phối các block cho từng SM vì SM, Block là 2 khái niệm riêng biệt (physical và logical) => thông qua trung gian WARP
* Warp:
  + Warp vừa là physical, vừa là logical
  + Trong grid chứa blocks, trong block chứa threads.
  + Về mặt physical, warp quản lý các SP. Số lượng warp tùy thuộc vào từng máy tính.
  + Về mặt logical : warp quản lý số lượng thread
  + Nhiệm vụ warp:
    - Phân phối các blocks cho các SM, sau đó lấy các blocks trong SM chia cho các SP.
    - Tại 1 thời điểm, 1 warp phân phối tối đa 32 thread. Sau khi xử lý xong mới phân phối tiếp 32 thread để xử lý công việc. Mà 1 SM có nhiều warp => tại 1 thời điểm sẽ thực hiện số công việc là: 32 \* số warp

## Hello world cuda-C

* Cấu trúc chương trình CUDA:
  + Mã tuần tự và mã song song theo task chạy trên host
  + Mã song song theo dữ liệu chạy trên device
* Quá trình nvidia cuda comipler (nvcc) hoạt động:
  1. Nvcc phân tích cú pháp mã nguồn host và device theo các từ khóa \_\_global\_\_, \_\_device\_\_, \_\_host\_\_
  2. Host compilation: nvcc tạo mã chạy trên cpu (c/c++) để gọi các hàm cuda trên gpu
  3. Device compilation: nvcc sử dụng ptx compiler để biên dịch mã device => dạng PTX ( 1 dạng assembly trung gian của CUDA) => cubin : mã thực thi gpu hoặc vẫn ở dạng PTX để sd vs gpu khác
  4. Linking: liên kết host code và device code + CUDA runtime để quản lý việc gọi kernel trên gpu.
  5. Tối ưu mã: nvcc tối ưu mã cuda để tăng hiệu suất tính toán trên gpu
  6. Sinh mã máy: nvcc tạo mã máy thành file .out để chạy chương trình cuda trên gpu

### Copy data

* COPY dữ liệu từ cpu sang gpu và ngược lại do cpu và gpu là các memory riêng biệt nên ko thể truy cập trực tiếp mà phải qua các PCI

A white rectangular box with black text

AI-generated content may be incorrect.

* + \_\_host\_\_ <datatype>: gọi và thực thi hàm trên cpu. Các hàm ko có chỉ định gì thêm thì mặc định trên cpu
  + \_\_global\_\_ void:
    - hàm được gọi trên cpu và thực thi trên device, còn gọi là kernel function.
    - \_\_global\_\_ và void luôn chung 1 cặp vì cpu và gpu là riêng biệt nên ko thể trả vè giá trị. Phải copy data qua PCI
  + \_\_device\_\_ <datatype>:
    - Là 1 function được gọi bởi device và thực thi trên device.
* Ví dụ:

#include <stdio.h>

\_\_global\_\_ void kernel()

{

printf("hello world");

}

int main()

{

kernel<<<1,1>>>();

cudaDeviceSynchronize();

return 0;

}

* + <<<1,1>>> : 1 first là số block, 1 second là số thread trong 1 block ~ số SP trong block, tức: chỉ định số SP thực hiện công việc
* cudaDeviceSynchronize() : là 1 hàm đồng bộ hóa, đảm bảo rằng tất cả các tính toán trên gpu đã hoàn thành trước khi chương trình tiếp tục thực hiện các tác vụ tiếp theo trên cpu
* A white background with black text

  AI-generated content may be incorrect.
* A black background with white text

  AI-generated content may be incorrect.

## Cách thức hoạt động của CPU-GPU

* CPU: hướng đến giảm độ trễ và thời gian phản hồi, thiên về sử dụng cache và điều khiển với kỹ thuật **hide short latency**
* GPU: hướng đến xử lý 1 lượng task/data lớn nhưng đơn giản trong thời gian ngắn => càn cung cấp lượng lớn data cho gpu => kiến trúc với bandwidth và memory của gpu hơn cpu rất nhiều
* A black table with white text

  AI-generated content may be incorrect.
* Đặc điểm design của cpu:
  + Hướng tới độ trễ thấp và xung nhịp cao (high clock speec/high clock frequency)
  + Ít core
  + Large cache
  + Xử lý tác vụ phức tạp
  + Optimize hardware
  + Đa luồng để hide short latency:
    - Short latency xảy a khi tải dữ liệu từ cache, đồng bộ thread trong gpu,…
    - Dùng multithread để 1 thread đang chờ => thread khác nhảy vào thực thi
* Đặc điểm design của gpu:
  + Hướng tới thông lượng cao với xung nhịp vừa phải (moderate clock freq)
  + Small cache
  + Xử lý tác vụ đơn giản
  + Ko optimize hardware
  + Massive number of cores để ẩn very high latency:
    - Long latency xảy ra khi truy cập global memory hoặc tải dữ liệu từ RAM => nếu có ít luồng và phải chờ dữ liệu => gây hiệu suất thấp
    - Massive number of cores keep it always busy.

## SIMD vs SIMT

* SIMD:
  + 1 instruction áp dụng cho nhiều data cùng 1 lúc
  + Sd trong cpu
  + Máy tính tự chia 1 bài toán lớn thành **N bài toán con**
* SIMT:
  + Kiến trúc được phát triển bởi nvidia
  + Mô hình mở rộng của SIMD dành cho gpu, 1 instruction áp dụng cho nhiều thread nhất có thể
  + Gpu tổ chức các thread thành warp và xử lý các warp song song.
  + Coder thực hiện chia các thread để xử lý N bài toán độc lập => việc chia bao nhiêu thread cho mỗi block cho hợp lý???

A black screen with white text

AI-generated content may be incorrect.

## Các bộ nhớ trong GPU

* Bộ nhớ vật lý:
  + Gồm các thanh RAM và thiết bị lưu trữ như ổ cứng
  + 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 (virutal memory)
  + Là không gian địa chỉ mà hệ điều hành và các chương trình có thể truy cập
  + Hdh quản lý việc ánh xạ giữa địa chỉ logic và địa chỉ vật lý
* Vì cơ chế SIMT nên phải hiểu sự phân bố của thread, block trong bộ nhớ của gpu:

A diagram of a memory

AI-generated content may be incorrect.

* Local memory:
  + là bộ nhớ có phạm vi truy cập nhỏ nhất và dành riêng cho mỗi thread
  + mỗi thread có thể sử dụng local memory riêng để lưu trữ các biến tạm thời
* Shared memory:
  + Là nơi giao tiếp và chia sẻ dữ liệu của các thread trong cùng 1 block
  + Nhanh hơn so với việc truy cập global mem
* Global memory:
  + Bộ nhớ lớn nhất trong gpu và được truy cập bởi tất cả threads trên mọi block
  + Truy cập thường chậm hơn so với các loại bộ nhớ khác
* Texture memory, constant memory:
  + Được tối ưu cho truy xuất các dữ liệu như hình ảnh và hằng số
  + có thể truy cập bởi tất cả các Thread trên mọi Block
  + giảm tải dữ liệu cho shared memory

A screenshot of a computer

AI-generated content may be incorrect.

* Global memory (device memory):
  + Có bộ nhớ lớn nhất trong gpu
  + Tốc độ truy cập chậm nhất (sau pcle)
  + Khởi tạo 1 giá trị bất kì mà ko chỉ định bộ nhớ lưu trữ => auto lưu vào global memory
* Lưu trữ dữ liệu lớn
* Shared/cache memory:
  + Tốc độ truy xuất nhanh => khó kiểm soát dữ liệu => bank conflict
  + Dung lượng ko lớn bằng global memory
  + Quá trình copy dữ liệu

Cpu => gpu => global memory => shared memory => global memory

### Prefetch

* Là kỹ thuật tải trước dữ liệu từ bộ nhớ chính vào bộ nhớ cache trước khi nó cần dùng
* Nhằm tối ưu hóa hiệu suất truy cập dữ liệu
* Nó dự đoán trước các truy cập dữ liệu tiếp theo chương trình cần nạp dựa trên các mẫu truy cập trước đó => tải trước các dữ liệu này vào bộ nhớ cache

### L1 cache và L2 cache

* Cache là bộ nhớ tạm thời : tạm thời chứa dữ liệu cho đến khi dữ liệu thực sự được truy cập thì chúng sẽ được đẩy lên bộ nhớ chính để xử lý. Vùng nhớ trống lại được thay thế bởi dữ liệu kế tiếp.
* Vùng L1 cache là bộ nhớ để cache cho shared memory để áp dụng cơ chế prefetch cho shared memory
* Vùng L2 cache là bộ nhớ để cache cho global memory để áp dụng cơ chế prefetch cho global memory

### Code demo

* Ko thể truyền tham chiếu từ CPU vào GPU vì:
  + CPU và GPU có bộ nhớ riêng biệt
  + Tham chiếu: lấy địa chỉ của biến trên RAM và xử lý trực tiếp trên đó. CPU có toàn quyền truy cập vào RAM, nó có thể truy xuất dữ liệu mà không gặp vấn đề.
  + GPU kernel chỉ chạy trên **bộ nhớ của GPU (VRAM)**
  + Nếu ta truyền một tham chiếu từ CPU (int &a), nó sẽ trỏ đến bộ nhớ của CPU, nhưng GPU không có quyền truy cập trực tiếp vào bộ nhớ CPU
  + Ví dụ:

A computer screen with white text and numbers

AI-generated content may be incorrect.

* <https://viblo.asia/p/lap-trinh-song-song-bai-6-su-dung-cac-bo-nho-trong-gpu-bXP4WzzoV7G#_global-memory-2>
* Lưu ý: dùng cudaMemcpy là data auto copy vào global memory
* Local memory và register:
  + Là loại bộ nhớ nhanh nhất có sẵn cho mỗi thread.
  + Độ ưu tiên: register files => local memory:
    - Khi regisrer file đầy , gọi là register spilling=> các biến được đẩy xuống bộ nhớ local

### Synchronization vs Asynchronization

* Trong cùng 1 warp sẽ có các thread hoàn thành công việc với tốc độ khác nhau. Nhưng do đồng bộ, warp sẽ hoàn thành nếu thread chậm nhất hoàn thành.
* Nhưng ko để 1 thread chậm làm ảnh hưởng cả warp => sinh ra khái niệm latency hiding
* Nếu 1 warp thực thi quá lâu thì có 1 cơ chế sẽ tự động chuyển sang 1 warp khác đã sẵn sàng thực thi, tức sẽ mapping 32 thread của warp mới vào các SP của SM => always keep thread busy
* Thread divergence là hiện tượng các thread đợi nhau khi có điều kiện xảy ra giữa chúng.
  + Các thread trong cùng 1 block thì chỉ synchronize khi qua công việc tiếp theo
  + Còn trong cùng 1 công việc thì vẫn là asynchronize.

### Unified memory

* Là cầu nối giữa ko gian bộ nhớ của host và device:
  + Cho phép truy cập memory của cpu và gpu thông qua single pointer
  + hệ thống sẽ tự động di chuyển dữ liệu giữa host và device mà ko cần quan tâm copy data từ h2d hay d2h.
* Nói đến memory, luôn đi kèm 2 khái niệm physical mem và logical mem
* Nhìn unified memory ở 2 góc độ này:
  + Logical memory (developer view) : unified mem là 1 bộ nhớ thống nhất giữa cpu và gpu
  + Physical mem (computer view): vẫn là 2 bộ nhớ riêng biệt cpu và gpu. Nhưng nhờ cơ chế zero copy nên ta thấy unified memory là bộ nhớ thống nhất giữa cpu và gpu
* Zero copy:
  + Là phương thức tối ưu hóa truyền dữ liệu: dữ liệu tự động chuyển trực tiếp từ bộ nhớ của cpu sang gpu mà ko cần bước trung gian (như sd buffer)
* Cơ chế của unified memory:
  + Khi có sự thay đổi dữ liệu ở unified memory thì dữ liệu đó sẽ được đồng thời thay đổi ở cpu và gpu theo cơ chế mapping
  + Nhưng ko biết khi nào dữ liệu sẽ được mapping về cpu và gpu => tình trạng page faults: ko tìm thấy dữ liệu yêu cầu
  + Máy tính sẽ thực hiện cơ chế MMU (memory management unit) để ktra xem dữ liệu có tồn tại ko. Nếu có thì sẽ được tải về.
  + <https://viblo.asia/p/lap-trinh-song-song-bai-8-unified-memory-vlZL932d4QK>

### Pinned memory

* Vcv

# Professional cuda c programming

* Programming model là nhìn ở **logical’s view**: developer tương tác với phần cứng thông quan ngôn ngữ lập trình, tổ chức các thread thành blocks và grids, tổ chức bộ nhớ.
* Execution model là nhìn ở **hardware’s view**: gpu hardware thực thi các lệnh được lập trình theo programming model. Việc lập lịch thực thi thread, đồng bộ hóa luồng và quản lý bộ nhớ là 1 phần của mô hình thực thi.
  + Cung cấp cái nhìn hữu ích trong việc viết code hiệu quả về instruction throughput và memory access

A diagram of a computer hardware

AI-generated content may be incorrect.

## CUDA programming model

### Introduction to CUDA programming

* Khi thiết kế thuật toán, ta quan tâm ở domain level: cách phân rã data và phân rã function để giải quyết vấn đề trong môi trường song song
* Khi lập trình, ta quan tâm ở logic level: cách tổ chức các luồng song song
* Hoạt động độc lập giữa host và device:
  + Khi một kernel được khởi chạy, CPU ngay lập tức lấy lại quyền kiểm soát, cho phép tiếp tục thực thi các tác vụ khác song song với quá trình tính toán dữ liệu trên GPU.
* Mô hình lập trình CUDA là bất đồng bộ:
  + Tính toán trên GPU có thể chồng lấp với quá trình giao tiếp giữa CPU và GPU, giúp tăng hiệu suất.

### Organizing threads

A diagram of a computer

AI-generated content may be incorrect.

* Nhìn thread theo grid và block là hướng nhìn theo logical view
* Grid:
  + Chứa tất cả các thread được tạo ra từ 1 lần khởi chạy kernel
  + Đặc điểm:
    - Tất cả các luồng trong một grid chia sẻ cùng một global memory
    - Grid bao gồm nhiều thread block.
  + Dimensions of a grid: dim3 gridDim
* Block:
  + Chứa tập các thread
  + Các thread trong **cùng 1 block** đều chung:
    - Block-local synchronization
    - Block-local shared memory
  + Dimensions of a block: dim3 blockDim
  + Lựa chon block size dựa vào:
    - Đặc điểm hiệu suất của kernel
    - Giới hạn của GPU resource
* Phân biệt 2 thread dựa vào :
  + blockIdx : block index within a grid
  + threadIdx : thread index within a block
* Nhờ cordinate của thread mà có thể gán các phần data cho các thread khác nhau
* Data type của block, thread:
  + Trên host: dim3
  + Trên device: uint3
* Do kernel là asynchronous function nên khi host gọi kernel thì nó sẽ ko đợi kernel hoàn thành mà sẽ chạy lệnh tiếp theo luôn => để host đợi kernel function hoàn thành và trả về kết quả thì thêm cudaDeviceSynchronize()
* cudaDeviceReset() để giải phóng tài nguyên trên gpu
* dim3 là kiểu dữ liệu sử dụng để xác định kích thước của grid và block khi khởi chạy kernel:
  + dim3 grid(x, y, z);// x, y, z - Số lượng block trong grid theo từng chiều
  + dim3 block(a, b, c); // Số lượng thread trong mỗi block theo từng chiều
  + Note:
    - block size: x, y <= 1024, z <= 64; x \*y \*z<= 1024
    - grid size cũng có giới hạn.
* Tổ chức của các thread trong cuda program:
  + Cuda runtime tự khởi tạo threadIdx, blockIdx, blockDim, gridDim cho mỗi thread khi kernel được thực thi

A screen shot of a computer

AI-generated content may be incorrect.

* + Trong trường hợp 2D block, các thread trong cùng block sẽ truy cập các vùng nhớ liên tiếp hoặc các phẩn tử liên tiếp trong array:

A screenshot of a computer

AI-generated content may be incorrect.

* + - Công thức:
      * row\_offset = (blockDim.x\*blockDim.y\*gridDim.x)\*blockIdx.y
      * block\_offset = blockDim.x \* blockDim.y\*blockIdx.x
      * tid = threadIdx.x + threadIdx.y\*blockDim.x
      * gridId = grid\_offset + block\_offset + threadIdx.x

### Launching a CUDA Kernel

* Cấu trúc 1 hàm kernel:
  + kernel\_name <<<grid\_dimension, block\_dimension>>>(argument list)
  + grid dimension : number of blocks to launch
  + block dimension : number of threads within each block

A white paper with black text

AI-generated content may be incorrect.

### Verifying the result

* có thể đặt configuration là <<<1,1>>> để kernel chạy tuần tự nếu gặp vấn đề thứ tự của các operation.

### Error handling

* Gồm 2 loại:
  + Compile time errors : do lỗi syntax
  + Runtime error : while program is running
* Do cuda làm việc giữa 2 hardware nên phải transfer error từ device to host :
  + cudaError is a enum defined in CUDA API:
    - return cudaSuccess : tức chạy thành công trên device
    - hoặc các giá trị lỗi khác:
      * sử dụng cudaGetErrorString(error) để get text của error
      * Mọi lệnh **gọi** CUDA, trừ kernel launch, đều trả về cudaError\_t
* Nguyên tắc trong làm việc:
  + error check là bắt buộc
  + **Kiểm tra lỗi sau mỗi API call**
  + **Kiểm tra lỗi khi launch Kernel**
  + **Sử dụng cudaDeviceSynchronize() để debug lỗi bất đồng bộ**
  + **Dùng cuda-memcheck để phát hiện lỗi nâng cao**
* cudaError\_t là kiểu dữ liệu trong CUDA để biểu diễn error code trả về từ API của cuda runtime

### Execution time comparation

* muốn so sánh hiệu suất giữa GPU và CPU để tìm ra solution tốt nhất:
  + đo thời gian thực thi:
    - Đo thời gian trên CPU : ghi lại thời điểm trước và sua thực thi hàm
    - Đo thời gian trên GPU: đo thời gian truyền dữ liệu giữa cpu và gpu   
      + đo thời gian thực thi kernel
  + Tối ưu hóa hiệu suất GPU:
    - Tìm cấu hình block tốt nhất bằng thử nghiệm các block 128, 256, 512, 1024; grid, shared memo và so sánh thời gian thực thi => gọi trial and error method.
* Ngoài thời gian thực thi, các yếu tố như tiêu thụ điện năng, ko gian phần cứng và chi phí cũng cần xem xét trong ứng dụng công nghiệp
  + 64 :
  + 128
  + 256
  + 512

### ORGANIZING PARALLEL THREADS

* Thông thường, trong global memory, matrix 8x6 thường lưu trữ theo hàng:

A green rectangular object with black text

AI-generated content may be incorrect.

* Để truy cập data từ global memory, ta phải tính toán “global index” dựa vào block index và thread index. Sau đó dùng global index để lấy data trong global memory.
* Tính global index với grid 2D và block 2D:

A grid of squares with red dots

AI-generated content may be incorrect.

Ví dụ ma trận 8x6 được biểu diễn theo bộ nhớ với block(4,2), grid(2,3)

A grid of numbers and symbols

AI-generated content may be incorrect.

Note:

* + Changing execution configurations affects performance
  + A naive kernel implementation does not generally yield the best performance.
  + For a given kernel, trying different grid and block dimensions may yield better performance

### Managin device

* GPU device information can help set the kernel execution configuration
* Có các cách để query GPU device infor:
  + Dùng cuda runtime api
  + Dùng nvidia systems management interface (nvidia-smi)
* **CUDA Toolkit** ra mắt lần đầu vào năm 2007 với phiên bản **1.0**.
* Các thiết bị CUDA có **compute capability** khác nhau (**từ 1 đến 7**), ảnh hưởng đến các giới hạn và tài nguyên phần cứng.
* Một số thuộc tính quan trọng của thiết bị CUDA:
  + **Tên thiết bị**
  + **Compute capability (major.minor)** – Ví dụ: 5.2 (major = 5, minor = 2)
  + **Tổng bộ nhớ toàn cục (TotalGlobalMem)**
  + **Số lượng tối đa thread/block** (thường là 1024)
  + **Kích thước tối đa của block và grid**
  + **Clock rate** – Tốc độ xử lý (KHz)
  + **Shared memory/block** – Bộ nhớ chia sẻ tối đa (KB)
  + **Warp size** – Kích thước warp (hiện tại là 32)
* Để truy vấn thông tin thiết bị trong CUDA, ta sử dụng:
  + **cudaGetDeviceCount()** – Kiểm tra số lượng thiết bị CUDA có sẵn.
  + **cudaGetDeviceProperties()** – Lấy thông tin thiết bị.

### Exercise

A screenshot of a computer program

AI-generated content may be incorrect.

## CUDA Execution model

* Ta đã thực hiện phương pháp “trial and error” (thử và sai) để thử nghiệm nhiều configuration để tìm ra cấu hình tốt nhất của grid và block.
* Bây giờ, chương này sẽ cung cấp 1 số guideline cho việc chọn cấu hình cho grid và block mà ko phải “thử và sai” nữa.
* Ở chương này, focus vào **instruction throughput theo hardware view**

### Introduction to cuda execution model

#### GPU architecture overview

* GPU xây dựng dựa trên **trái tim** là mảng các SM. 1 GPU có thể có nhiều SM. Mỗi SM hỗ trợ thực thi đồng thời hàng trăm thread.
* Register và share memory trong SM rất hạn chế nên tổ chức tốt những cái này để get the best performance.
* Khi kernel được launched, các block được chia cho các SM, số lượng tùy theo resource của SM.
* Các thread của 1 block chỉ thực thi trong SM được phân.
* Bên cạnh việc thread-level parallelism, trong 1 thread, GPU cũng thực thi theo instruction-level parallelism nếu các instruction độc lập.
* CUDA thực hiện SIMT theo các warps:
  + SM chia blocks thành các warp. Mỗi warp gồm 32 thread.
  + Các thread trong 1 warp thực thi cùng 1 lệnh tại same time.
  + Mỗi thread có instruction address counter và register state riêng. Thực thi lệnh đó trên chính data của nó
* Phân biệt SIMT và SIMD:
  + Giống: đều phân phối cùng 1 instruction đến nhiều execution unit.
  + Khác:
    - SIMD các phần tử dữ liệu được 1 lệnh xử lý cùng 1 lúc và thống nhất.
    - SIMT: các thread trong cùng warp thực thi độc lập. Mỗi thread có trạng thái và hành vi khác nhau, như nhanh chậm, rẽ nhánh, truy bộ bộ nhớ,..
* Shared memory is partitioned among blocks on SM. Registers are partitioned among threads => các thread có thể cooperate và communicate với nhau.
* Logic view, all thread trong block chạy song song. Hardware view, không phải all threads trong 1 block đều thực thi cùng 1 lúc => chúng có tiến độ thực thi khác nhau => Việc share data giữa các thread gây race condition => cần synthread trong 1 block để các thread chạy tới 1 điểm chung.
* 1 warp đã được lên lịch nhưng chưa sẵn sàng thì SM chuyển sang warp khác.
  + Việc SM swap warp ko tốn overhead do hardware resource được phân chia sẵn cho các thread và block trong 1 SM => ko tốn chi phí lưu và khổi phục trạng thái như cpu

#### Profile-driven optimization

* Profiling là hoạt động phân tích hiệu năng chương trình qua đo lường:
  + Space/time complexity của application code
  + Các lệnh cụ thể
  + Tần suất và thời gian thực thi hàm
* Giúp phân tích hiệu năng và xác định bottleneck của kernel.
* Sử dụng nvvp (1 phần của Nsight) và nvprof để phân tích và tìm các bottle nếu có
* Event is countable activity, corresponds to a hardware counter, during kernel execution.
* A metric là một chỉ số hiệu suất của GPU, được tính toán từ một hoặc nhiều event (sự kiện phần cứng) trong quá trình thực thi kernel.
* Các performance limiters của kernel là:
  + Memory bandwidth
  + **Compute resource**
  + **Instruction** **and memory latency**

### Understanding the nature of warp execution

#### Warp and block

* Khi block được lên lịch vào 1 SM, threads trong block được chia thành các warp.
* 1 warp gồm 32 consecutive threads, thực thi theo SIMT, tức “all threads execute the same instruction, and each thread carries out that operation on its own private data “

A green rectangular box with black text

AI-generated content may be incorrect.

* From the logical perspective, a thread block is a collection of threads organized in a1D, 2D, or 3D layout
* Hardware view: tất cả thread được sắp xếp theo 1 chiều. Mỗi thread có 1 id duy nhất trong block. Các thread có id liên tiếp được nhóm thành 1 warp.
* Ví dụ: block 1D với 128 thread:

A close up of text

AI-generated content may be incorrect.

* 1 warp chỉ chứa các thread trong 1 block. Nếu block size ko là bội của warp size, 1 số thread của warp cuối sẽ inactive.
  + Ví dụ: block 2D có size (40, 2), tức có 80 thread. Hardware sẽ allocate 3 warp cho block này. Tức tốn 96 hardware thread để support 80 software thread. 16 thread cuối của warp cuối sẽ inactive, nhưng vẫn chiếm SM resource, such as register.

A diagram of a thread block

AI-generated content may be incorrect.

#### Warp divergence

* Lý thuyết: Nếu 1 thread trong warp thực thi 1 lệnh thì all threads trong warp đó phải thực thi lệnh đó
* Warp divergence xảy ra khi các threads trong một warp thực thi các nhánh khác nhau của mã lệnh do câu lệnh rẽ nhánh
* Nếu bị warp divergence, the warp serially executes each branch path, disabling threads that do not take that path=> Càng nhiều branch càng nhiều thread bị disable, làm giảm hiệu suất
* Take note that branch divergence occurs only within a warp. Different conditional values in different warps do not cause warp divergence.

A colorful squares with black text

AI-generated content may be incorrect.

* Solution:
  + Hạn chế câu lệnh rẽ nhánh trong warp. Nhưng ko phải câu lệnh rẽ nhánh nào cũng gây warp divergence. Vì nếu all thread trong warp đều thực thi theo 1 path thì vẫn ok.
  + Phân vùng lại dữ liệu để các thread trong cùng 1 warp thực thi cùng path.
    - Chia dữ liệu theo warp appoarch:
      * Dùng điều kiện (tid/warpSize) % 2 => chia công việc theo warp chẵn, lẻ
* Để đo ratio của non-divergent branch to total branch, dùng metric là “ branch efficiency”:
  + Công thức:

A close up of a name

AI-generated content may be incorrect.

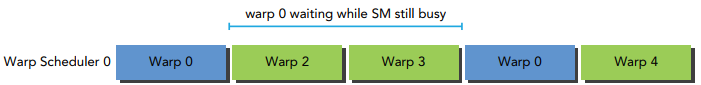
* + Có tool đo branch efficient dùng câu lệnh cmd:
    - Khi biên dịch .cu thì thêm tag -G để tắt các trình optimize của device (nhưng ko tắt được hết, vẫn còn các optimizer cơ bản)
    - nvprof –metrics branch\_efficiency file.out
* CUDA compiler cũng có cơ chế optimization với predicated instruction for short, conditional code segment để giảm divergence branch.

#### Resource partitioning

* Các resource cần cho 1 warp để thực thi gồm:
  + Program counters
  + Registers
  + Shared memory
* Gọi là execution context
* Execution context được xử lý bởi SM, được duy trì trên chip trong suốt vòng đời của warp => switch giữa các warp ko tốn chi phí.
* Mỗi SM có :
  + 1 tập các register 32-bit lưu trong register file, chia cho các thread => more thread with fewer register per thread.
    - Số register phân vào mỗi thread phụ thuộc vào kernel function được biên dịch.
    - Register lưu trữ các biến cục bộ, biến vòng lặp, con trỏ, địa chỉ bộ nhớ, kết quả tạm thời, program counter của mỗi thread
    - Nếu một kernel sử dụng quá nhiều thanh ghi, CUDA có thể đẩy một số biến ra local memory, làm giảm hiệu suất do phải truy cập bộ nhớ chậm hơn.
  + 1 lượng cố định shared memory chia cho các block => more block with fewer shared memory per block.
* Bộ nhớ chia sẻ có thể truy cập bởi tất cả các luồng trong một block, trong khi thanh ghi là riêng tư cho từng luồng.
* Block được allocate resource, gọi là active block. Active block chứa các active warp. Active warp chia làm ba loại:
  + Selected warp: Warp đang được thực thi.
  + Eligible warp: Warp sẵn sàng thực thi nhưng chưa được chọn để thực thi ngay
  + Stalled warp: Warp chưa sẵn sàng thực thi do đang chờ tài nguyên hoặc dữ liệu.
* Mỗi warp scheduler trên mỗi SM sẽ chọn 1 warp trong mỗi chu kì thực thi và gửi nó đến đơn vị thực thi
* Một warp là eligible nếu thỏa mãn hai điều kiện sau:
  + Có sẵn 32 nhân CUDA để thực thi.
  + Tất cả các đối số của lệnh hiện tại đã sẵn sàng.
* 1 warp bị stall thì scheduler sẽ chọn warp khác thay thế
* Compute resource giới hạn số active warp. Ta cần phải keep lượng lớn warp active để hide latency gây ra bởi warp stalling => Mục đích để max performance.

#### Latency hiding

* Instruction latency là số chu kỳ xung nhịp (clock cycle) từ khi một lệnh được bắt đầu đến khi hoàn thành
* CPU minimize latency for one or 2 threads at a time
* GPU handle a large number of concurrent and lightweight threads to maximize throughput
* GPU hide latency bằng cách thực thi các warp khác.
* Mỗi chu kỳ, bộ lập lịch chỉ chọn 1 warp để thực thi
* Các phép toán số học và truy cập bộ nhớ có độ trễ, đo bằng số chu kỳ xung nhịp, có thể được ẩn bằng cách duy trì độ chiếm dụng cao (nhiều warp hoạt động trong mỗi SM).
* Cơ chế ẩn độ trễ:
  + Khi một warp đang thực thi thao tác có độ trễ, nếu trên SM có đủ số eligible warps cần thiết để keep the cores occupied, tiến hành chuyển đổi warp để ẩn độ trễ. Do execution context của mỗi warp là on-chip nên ko có độ trễ trong việc swap



* Cách estimate num\_of\_active\_warps cần thiết để ẩn latency:



* + Latency of an instruction (cycle)
  + Throughput (warp/cycle)
* Còn cách nữa nhưng đọc ở sách chỗ bôi vàng

#### Occupancy

* Occupancy tập trung vào tỷ lệ số lượng luồng hoặc warp đồng thời trên mỗi SM
* Đạt occupancy tối đa không phải là mục tiêu duy nhất trong tối ưu hóa hiệu suất.



* Công thức : occupancy = active warps/maximum waprs
  + Active warps có thể tính theo:
    - Nếu có kernel cụ thể: mỗi SM của device có bao nhiêu register và mỗi thread của kernel dùng bao nhiêu register
* Tăng occupancy có thể gây ra:
  + Tăng số lượng lệnh thực thi
  + Làm tràn thanh ghi sang bộ nhớ cục bộ (là bộ nhớ ngoài chip – off chip memory)
  + divergent branches
* Guiline for block and grid:
  + Số thread trong block nên là bội của 32. Nên bắt đầu từ 128, 256,…
  + Tùy chỉnh block size theo yêu cầu tài nguyên của kernel
  + Giữ num of block luôn lớn hơn num of SMs
  + Thực nghiệm để tìm ra cấu hình tốt nhất

#### Synchronization

* Trong Cuda , có thể thực hiện đồng bộ ở 2 level:
  + System : đợi cho các work trên cả host và device đều complete
  + Block : Đợi tất cả các thread trong một block đạt đến cùng một điểm thực thi trên device
* Vì lệnh nhiều cuda api và kernel là asyn => cung cấp cudaDeviceSynchronize() blocks the host đến khi các work trên device hoàn thành:

cudaError\_t cudaDeviceSynchronize(void);

* Không có đồng bộ hóa giữa các block khác nhau, cách duy nhất là kết thúc *kernel* hiện tại và khởi chạy một *kernel* mới để tiếp tục công việc sau khi đồng bộ hóa toàn cục
* \_\_device\_\_ void \_\_syncthreads(void) : syn các thread trong block
* Các thread trong 1 block có thể chia sẻ dữ liệu qua shared memory và register. Khi chia sẻ dữ liệu giữa các thread, cần tránh race condition do :
  + Về mặt logical, các thread trong block chạy song song
  + Nhưng về mặt hardware, ko phải các luồng đều thực thi cùng 1 lúc, có luồng trước, có luồng sau.

#### Scalability

* Mã tuần tự ko có tính mở rộng, chạy trên 1 lõi hay nhiều lõi vẫn giống nhau
* MÃ song song có tính mở rộng, phụ thuộc thiết kế thuật toán và đặc điểm phần cứng.
* có thể mở rộng quy mô trên nhiều GPU mà ko cần thay đổi mã
* Các block trong grid có thể thực thi theo bất kì thứ tự nòa, song song hoặc tuần tự
* transparent scalability : Khả năng thực thi cùng một mã trên nhiều core

### Exposing parallelism (Khai thác tính song song)

Thực hành tính tổng ma trận với grid 2D và block 2D và experiment with block and grid configurations

**Check active warps**

* Occupancy cao thể hiện active warp cao
* Nhưng ko phải occupancy cao là luôn có hiệu suất tốt

Check memory operation

* Load throughput: Là lượng dữ liệu được tải từ bộ nhớ trong một đơn vị thời gian
* **Check Load Throughput** 
  + A higher load throughput does not always equal to higher performance. Nguyên nhân liên quan memory transaction trong GPU (chapter 4)
* **Check Load Efficiency**
  + Là kiểm tra tỷ lệ phần trăm số byte dữ liệu được sử dụng hợp lệ so với tổng số byte được tải từ bộ nhớ.
  + Nếu chương trình tải nhiều dữ liệu nhưng chỉ sử dụng một phần nhỏ của chúng, thì hiệu suất tải sẽ thấp.

Exposing More Parallelism

* Block.x là số quan trọng trong việc quyết định performance
* Ko có metric đơn lẻ nào là phản ánh tốt performance => Cần cân bằng giữa các metric
* Kiểm tra kernel ở các góc độ khác nhau
* Grid/block config là khởi đầu tốt cho optimize

### AVOIDING BRANCH DIVERGENCE

Việc thực thi có điều kiện bên trogn 1 warp có thể gây divergence => hiệu suất kém

By rearranging data access patterns => giảm hoặc tránh được divergence

#### The Parallel Reduction Problem

* Reduction problem (bài toán rút gọn) là bài toán thực hiện 1 phép toán giao hoán và kết hợp trên 1 vector để thu được 1 kết quả duy nhất. Phép toán có thể là +, -, \*, max, min,… miễn là có 2 tính chất trên.
* Parallel reduction : là quá trình thực hiện bài toán trên theo cách song song
* Dựa vào cách cộng các phần tử theo cặp theo quy luật khác nhau => ta có các cách:
  + Neighbored pair: cặp các phần tử ngay cạnh nhau

A screenshot of a computer

AI-generated content may be incorrect.

* + Interleaved pair: cặp các phần tử cách nhau 1 stride.

A screenshot of a computer

AI-generated content may be incorrect.

* Sau đây là triển khai của 1 số parallel reduction kernel

#### Divergence in Parallel Reduction

* Với bài toán neighbored pair:
  + Có 2 global memory array: 1 mảng để lưu trữ toàn bộ mảng đầu vào, 1 mảng để lưu trữ tổng tạm thời. Mỗi block hoạt động trên 1 phần dữ liệu của array. Sau mỗi step, giá trị tổng tạm thời sẽ thay thế global memory
* Hàm kernel có điều kiện duyệt index của thread :

tid % (2 \* stride)) == 0

A diagram of a memory

AI-generated content may be incorrect.

* + Điều kiện này khiến tất cả các warp của block đều bị phân đoạn do chỉ xét thread id chẵn ở mỗi warp trong block.
  + Để cải tiến, ta dùng: ha

index = 2 \* stride \* tid

A diagram of a memory

AI-generated content may be incorrect.

* + - Ví dụ 1 block có 512 threads, 8 warp đầu của block thực thi trong first round, 8 warp còn lại rảnh. Round 2, 4 warp đầu thực thi còn 12 warp còn lại rảnh => ko có warp divergency
    - Chỉ xảy ra warp divergence khi ở round 5.
  + Để cải tiến tiếp, ta dùng Interleaved Pairs

#### Reducing with Interleaved Pairs

* Xử lý bài toán rút gọn (reduction problem) bằng cách xử lý các cặp phần tử xen kẽ (interleaved pairs) trong mảng dữ liệu. Thay vì xử lý các phần tử tuần tự, nó xử lý các phần tử có khoảng cách đều nhau.
* Ví dụ, trong một mảng có 8 phần tử, nó sẽ xử lý các cặp (phần tử 0, phần tử 4), (phần tử 1, phần tử 5), (phần tử 2, phần tử 6), và (phần tử 3, phần tử 7) cùng một lúc.
* Lặp lại cho đến khi thu được giá trị duy nhất
* Câu lệnh :

A computer code with black text

AI-generated content may be incorrect.

A diagram of a memory

AI-generated content may be incorrect.

* Kết luận: sự khác nhau về cách tính toán để chọn thread id để thực thi và cách lưu trữ giá trị tạm thời trên global memory là điểm khác chính giữa 3 thuật toán trên

### Unrolling loops

* Kỹ thuật mở vòng lặp tối ưu việc thực thi vòng lặp bằng cách giảm frequencyof branches and loop maintenance instructions, tạo ra nhiều lệnh độc lập
* Thay vì 1 viết thân vòng lặp 1 lần và sử dụng vòng lặp để thực thi nó lặp đi lặp lại => thân vòng lặp được viết nhiều lần trong code
* The number of copies made of the loop body is called the loop unrolling factor.
* Đạt hiệu năng tốt nhất là trong xử lý các vòng lặp mảng tuần tự : biết trước số vòng lặp

A close up of a white background

AI-generated content may be incorrect.

A white background with black text

AI-generated content may be incorrect.

* Ở high-level, ko nhận sự khác biệt. chỉ ở low-level mới thấy sự khác biệt.

#### Reducing with Unrolling

* Đối với mỗi thread block, dữ liệu từ hai data block sẽ được cộng lại
* Mỗi thread sẽ xử lý nhiều hơn 1 khối dữ liệu và làm việc với 1 phần tử dữ nhất từ mỗi data block
* Cách giải:
  + Xác định số phần tử data = n
  + Xác định số thread block và số thread trong 1 block

#### Reducing with Unrolled Warps

#### Reducing with Complete Unrolling

#### Reducing with Template Functions

### Dynamic Parallelism

#### Nested Execution

#### Nested Hello World on the GPU

#### Nested Reduction

### Mastering grid and block configuration heuristics

### Learning various CUDA performance metrics and events

### Probing dynamic parallelism and nested execution

Nội dung tập trung vào thực thi warp, phân chia tài nguyên, ẩn độ trễ và tối ưu hóa chương trình CUDA dựa trên mô hình thực thi của CUDA.

* Các components là những component của chip, là những resource cực limit trong chip:
  + cuda core giống cpu core, sẽ thực thi những câu lệnh được fetch vào core.
  + Share memory và L1 cache giúp truy cập bộ nhớ nhanh hơn
  + Register là bộ nhớ giới hạn để lưu những fly value để thực thi
  + Load-Store unit để load memory và store request
  + Warp scheduler xác định warp nào hoặc set các thread nào sẽ thực thi tiếp theo
* Computer architecture :

A black and white text on a black background

AI-generated content may be incorrect.

* + Cuda follows different architecture, called SIMT
    - Các block được thực thi trong single SM
    - Các block được thực thi song song trong cùng SM phụ thuộc vào resource của SM
    - 1 thread ko thể thực thi ở nhiều SM
    - Nếu device ko thể chạy 1 block trong SM => throw error cho kernel function.
    - A single thread in a thread block is going to execute by one single core
    - Các thread trong cùng 1 block sẽ chia sẻ tài nguyên của cùng 1 SM
    - Ko thể chia nhỏ 1 block để chạy trên nhiều SM
    - Ví dụ: 1 block có 512 thread = 16warp, 1 SM có 128 core. Thì chỉ thực thi song song 4 warp 1 lần.
    - Khi launch grid, các block được chia vào các SM
    - Software view: block có thể có 1,2,3D
    - Hardware view : all thread sắp xếp theo 1 chiều. Each thread có unique id. Và nhóm theo warp
    - blockIdx.x nên có giá trị là 32 để tất cả thread trong 1 warp có active state

### Parallel reduction

* cudaDeviceSynchronize() :
  + Được sử dụng để đồng bộ giữa Host và Device
  + Đảm bảo tất cả các thao tác trên GPU hoàn thành trước khi Host tiếp tục thực thi
* syncthreads :
  + Warp không cần \_\_syncthreads() vì luôn đồng bộ trong chính nó
  + Block có nhiều warp → Cần \_\_syncthreads() nếu warp này dùng dữ liệu của warp khác.
  + Giao tiếp giữa block → Dùng atomic hoặc global synchronization, không dùng \_\_syncthreads()
* Lưu ý khi sử dụng \_\_syncthreads():
  + Nếu sử dụng trong điều kiện if, có thể xảy ra branch divergence, khiến một số luồng không đến được lệnh \_\_syncthreads(), gây lỗi không xác định.
  + Đảm bảo tất cả các luồng đều thực thi đến lệnh \_\_syncthreads() để tránh deadlock.
  + Không cần \_\_syncthreads() nếu không sử dụng \_\_shared\_\_ memory.
  + Không cần \_\_syncthreads() khi chỉ sử dụng global memory
  + Không cần \_\_syncthreads() khi mỗi thread làm việc độc lập trên dữ liệu riêng của nó
* Reduction:
  + là một kỹ thuật trong lập trình song song dùng để tổng hợp một tập hợp dữ liệu lớn thành một giá trị duy nhất
  + Neighbored Pair Reduction:
    - chia tập dữ liệu thành từng cặp phần tử liên tiếp và thực hiện phép toán reduction trên từng cặp này
    - Mỗi bước lặp, số phần tử giảm đi một nửa cho đến khi chỉ còn một giá trị duy nhất.

for (int stride = blockDim.x / 2; stride > 0; stride /= 2) {

if (tid < stride) {

shared\_data[tid] += shared\_data[tid + stride];

}

\_\_syncthreads();

}

* Để cải thiện reduction divergence của đoạn code trên
* Đoạn code trên sau mỗi vòng lặp, stride giảm đi 1 nửa. Chỉ những tid < stride mới thực hiện phép toán. Những tid >= stride bị nhàn rỗi => lãng phí
* Đến khi tid < 32 thì xuất hiện warp divergency, do code bị phân nhánh:
  + 1 số luồng thực hiện phép côngj
  + 1 số luồng ko làm gì

### Loop unrolling và optimization

* Unrolling thread block giúp giảm tải GPU bằng cách xử lý nhiều dữ liệu hơn trong một lần lặp.
* Unrolling warp loại bỏ warp divergence trong warp đầu tiên bằng biến volatile.
* Full loop unrolling trong kernel giảm số lượng lệnh điều khiển vòng lặp.
* Dùng kích thước block làm tham số template để tối ưu hóa thêm.

### CUDA Dynamic Parallelism

* Thông thường, 1 kernel được khởi chạy từ CPU
* Với CUDA Dynamic Parallelism, ta có thể khởi chạy kernel từ 1 kernel khác
* Tuy nhiên cách này có hạn chế về hiệu suất và 1 số giới hạn nhất định:
  + S
  + A
  + A

## Mô hình bộ nhớ CUDA

A diagram of a memory

AI-generated content may be incorrect.

* Bộ nhớ của GPU có 2 loại quan trọng nhất:
  + Glocal memory:
    - Tương tự bộ nhớ RAM của CPU
    - Dung lượng lớn nhưng tốc độ truy cập chậm so vs các loại mem khác
  + Shared memory:
    - Giống với cache memory của CPU, nhưng điều khiển trực tiếp từ kernel CUDA C
    - Tốc độ truy cập nhanh hơn Global memory
* Tập trung tìm hiểu mô hình bộ nhớ CUDA, đặc biệt là global memory, và các mẫu truy cập bộ nhớ tối ưu

### Các loại bộ nhớ trong CUDA

* Tìm hiểu các loại bộ nhớ trong CUDA, vị trí của chúng và độ trễ khi truy cập.
* Cách khai báo bộ nhớ chia sẻ (shared memory) bằng từ khóa \_\_shared\_\_ và cách các biến tự động (automatic variables) được lưu trong thanh ghi (registers).

### Cơ chế sao chép bộ nhớ & tối ưu hóa

* Pinned memory giúp tăng tốc độ sao chép dữ liệu giữa CPU và GPU.
  + Bộ nhớ không chịu tác động của lỗi trang (page fault).
  + Bộ nhớ này ánh xạ vào device address space
* Unified memory:
  + Bộ nhớ được quản lý tập trung, có thể được truy cập bởi cả host và device.
* Zero-copy memory và Unified memory cho phép CPU và GPU chia sẻ cùng một vùng nhớ, giúp lập trình dễ dàng hơn mà không cần quan tâm đến việc sao chép dữ liệu.

### Global Memory Access Patterns

* Gồm:
  + Align memory access
  + coalesced memory access
* Truy cập bộ nhớ thẳng hàng (coalesced memory access) giúp tăng hiệu suất bằng cách tối ưu hóa băng thông bộ nhớ.
* Truy cập lệch hàng (misaligned access) làm giảm hiệu suất do sử dụng nhiều transaction bộ nhớ hơn mức cần thiết.
* Bộ nhớ L1 cache sử dụng giao dịch 128 byte, trong khi L2 cache sử dụng giao dịch 32 byte.

### So sánh AOS vs. SOA (Array of Structures vs. Structure of Arrays

* So sánh AOS vs. SOA (Array of Structures vs. Structure of Arrays

### Matrix Transpose problem

* mẫu truy cập bộ nhớ ảnh hưởng đáng kể đến hiệu suất, đặc biệt là khi đọc dữ liệu theo hàng nhưng ghi theo cột, gây ra truy cập không kết hợp (non-coalesced writes).
* Aligned, coalesced memory access
* Unrolling
* Avoiding partition campaign with diagonal coordinate system

## Shared Memory and constant memory

### Shared Memory

* + Shared mem banks
  + Access modes
  + Bank conflicts
  + Static and dynamic declared shared mem
  + Shared mem padding
* Bộ nhớ chia sẻ được tổ chức thành 32 ngân hàng bộ nhớ (memory banks) vì warp có kích thước 32 luồng.
* Có hai chế độ truy cập: 32-bit mode và 64-bit mode.
* Xung đột ngân hàng (bank conflict) xảy ra khi nhiều luồng trong một warp truy cập vào cùng một ngân hàng bộ nhớ nhưng ở các địa chỉ khác nhau, dẫn đến nhiều giao dịch bộ nhớ.
* Có thể khai báo bộ nhớ chia sẻ tĩnh (statically) hoặc động (dynamically), nhưng mảng động chỉ hỗ trợ mảng một chiều và cần cung cấp kích thước khi khởi chạy kernel.
* Sử dụng memory padding để tránh xung đột ngân hàng.

### Cải thiện thuật toán chuyển vị ma trận bằng Shared Memory

### Constant Memory

* Chỉ có thể đọc từ device và phải được thiết lập từ host.
* Có cơ chế bộ nhớ đệm (caching) hiệu quả, giúp tăng tốc truy cập dữ liệu không thay đổi.
* Áp dụng bộ nhớ hằng số để tối ưu hóa thuật toán stencil.
* triển khai thuật toán stencil

### Warp Shuffle Instructions

* Cung cấp cách truyền giá trị thanh ghi của giữa các luồng trong một warp mà không cần sử dụng bộ nhớ chia sẻ hoặc bộ nhớ toàn cục.
* Thực hiện thuật toán reduction bằng warp shuffle, giúp tối ưu hiệu suất hơn so với cách dùng shared memory.

## CUDA Stream

### Introduction to CUDA streams and events

* CUDA Streams là các hàng đợi thực thi song song trên GPU
* Events giúp đồng bộ hóa giữa các luồng (streams) hoặc đo thời gian thực thi.
* Chủ đề chính:
  + Tổng quan về streams và cách giúp thực thi không đồng bộ.
  + Events giúp đo thời gian hoặc đồng bộ hóa giữa các streams.

### How to use CUDA asynchronous functions

* CUDA hỗ trợ các hàm bất đồng bộ, giúp CPU có thể tiếp tục thực hiện công việc khác mà không cần chờ GPU hoàn thành.
* Ví dụ về các hàm bất đồng bộ:
  + cudaMemcpyAsync() – sao chép dữ liệu giữa CPU và GPU mà không chặn CPU.
  + cudaMemsetAsync() – khởi tạo bộ nhớ GPU mà không làm chậm chương trình chính.
* Lợi ích: Giúp tăng tốc độ xử lý bằng cách tận dụng tối đa tài nguyên GPU.

### How to use CUDA streams

### Overlapping memory transfer and kernel execution

* CUDA có thể truyền dữ liệu giữa CPU ↔ GPU đồng thời với thực thi kernel.
* Tăng hiệu suất bằng cách giảm thời gian chờ giữa các tác vụ.
* Ví dụ:
  + Gửi dữ liệu từ CPU → GPU (cudaMemcpyAsync) trong một stream.
  + Chạy kernel trong một stream khác (kernel<<<grid, block, 0, stream>>>).

### Stream synchronization and blocking behaviours of NULL stream

* NULL stream là stream mặc định trong CUDA.
* Nếu không chỉ định stream, CUDA sẽ sử dụng NULL stream, có thể gây chặn các tác vụ khác.
* Solution:
  + Sử dụng nhiều streams và đồng bộ hóa bằng cudaStreamSynchronize().
  + Sử dụng cudaEventRecord() để kiểm soát luồng thực thi.

### Explicit and implicit synchronization

* Explicit Synchronization:
  + Gọi cudaDeviceSynchronize(), cudaStreamSynchronize(), hoặc cudaEventSynchronize() để chờ GPU hoàn thành công việc.
  + Khi muốn kiểm soát chính xác thời điểm đồng bộ hóa.
* Implicit Synchronization:
  + Một số hàm như cudaMemcpy() sẽ tự động đồng bộ hóa trước khi thực hiện tác vụ tiếp theo.
  + Khi muốn tối ưu hóa hiệu suất bằng cách giảm số lần chặn không cần thiết.

### CUDA events and timing with CUDA events

* CUDA Events giúp đo thời gian thực thi giữa các lệnh.
* cudaEventCreate() để tạo event.
* cudaEventRecord() để ghi lại thời gian.
* cudaEventElapsedTime() để tính thời gian giữa hai sự kiện.
* Lợi ích: Đo hiệu suất của kernel hoặc các thao tác truyền dữ liệu trên GPU.

### Creating inter-stream dependencies with events

* Vấn đề: Khi có nhiều stream, đôi khi bạn muốn một stream phải đợi stream khác hoàn thành trước.
* Giải pháp: Sử dụng CUDA Events để tạo sự phụ thuộc giữa các streams.
* Cách thực hiện:
  + Dùng cudaEventRecord() để đánh dấu một điểm trong stream.
  + Dùng cudaStreamWaitEvent() để yêu cầu một stream khác chờ event đó hoàn thành.
* Lợi ích: Hạn chế tình trạng tắc nghẽn và tối ưu hóa hiệu suất xử lý song song.

## OpenCv trên cuda

* <https://github.com/datlt4/WLINUX/blob/master/Alias.md>
* <https://chatgpt.com/c/67d85c9e-1440-8011-90cf-ba437b0bcca7>
* <https://fpt-software.udemy.com/course/cuda-programming-masterclass/learn/lecture/12516380#overview>
* <https://modal.com/gpu-glossary>

https://github.com/CisMine/Parallel-Computing-Cuda-C

* <https://docs.nvidia.com/cuda/cuda-c-programming-guide/contents.html>

Kiến thức cần học trước TensorRT:

* Lập trình CUDA cơ bản:
  + Kiến trúc GPU, CUDA Cores, Tensor Cores.
  + Cách viết kernel function, quản lý bộ nhớ (Global, Shared Memory).
  + Lập trình CUDA bằng C++ hoặc Python (với CuPy, Numba).
* Machine Learning & Deep Learning
  + Các framework phổ biến: TensorFlow, PyTorch, ONNX.
  + Mô hình phổ biến: ResNet, YOLO, EfficientNet, Transformer.
  + Cách training và lưu mô hình (.pb, .onnx, .pth).
* Kiến thức về ONNX
  + Cách chuyển đổi mô hình PyTorch/TensorFlow sang ONNX.
  + Thư viện onnxruntime để kiểm tra mô hình ONNX.
* TensorRT – Thực hành cài đặt & sử dụng:
  + **Cách cài đặt TensorRT trên Ubuntu/Window**
  + **Chuyển đổi mô hình từ ONNX/PyTorch sang TensorRT**
  + **Tối ưu mô hình với FP16, INT8 để giảm kích thước & tăng tốc độ**