TÍNH TOÁN TRÊN BỘ XỬ LÝ ĐỒ HỌA ĐA DỤNG VÀ MÔ HÌNH LẬP TRÌNH CUDA

Ts. Vũ Văn Thiệu

Bộ môn Khoa học máy tính

Viện CNTT & TT, Trường ĐHBKHN

Tài liệu tham khảo

- Googling:
  - CUDA by examples
  - CUDA C programming guide

Nội dung

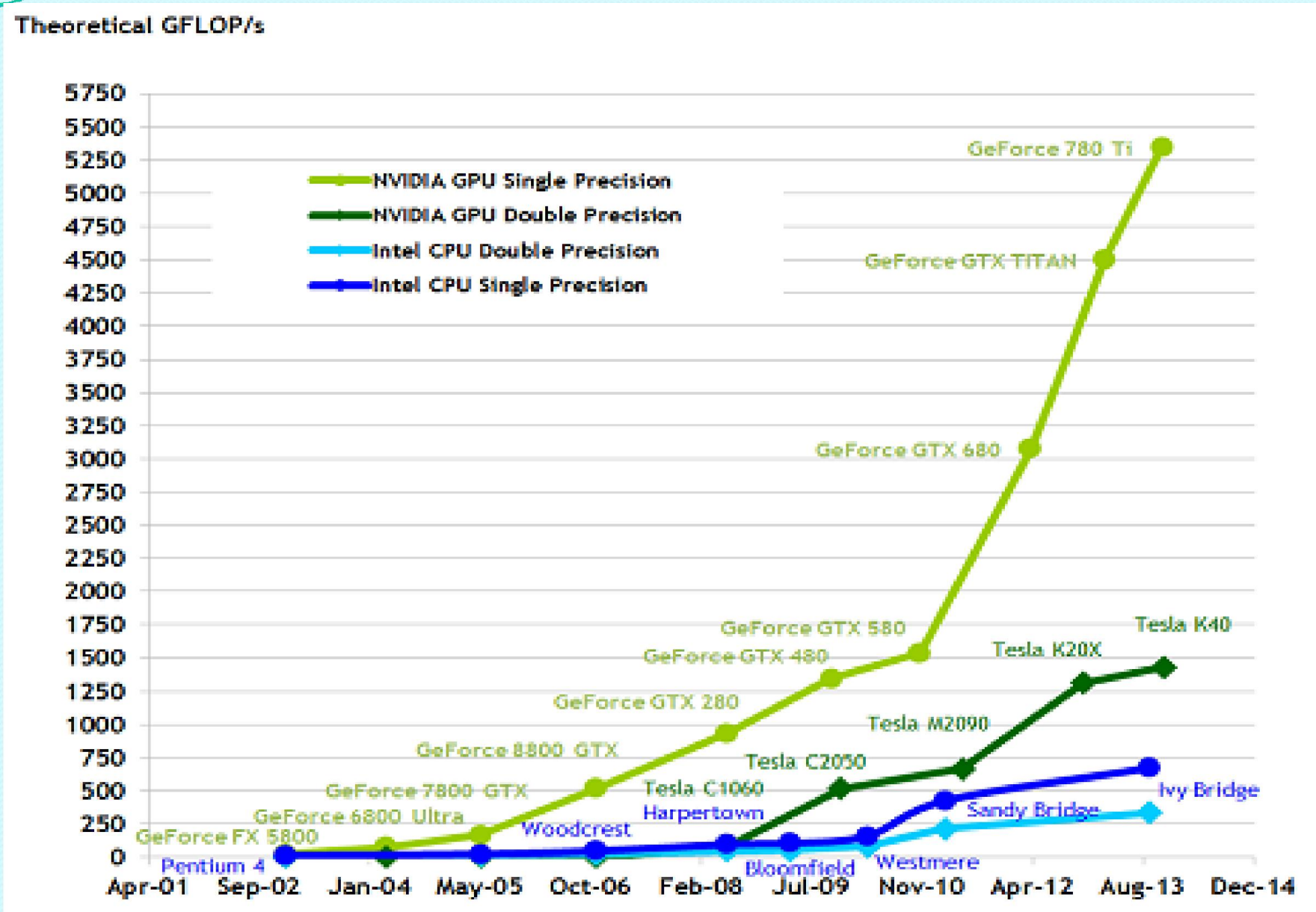
- Kiến trúc Bộ xử lý đồ họa đa dụng
  - GPGPU Architecture
- Mô hình lập trình CUDA
  - CUDA programming model
- Tối ưu hóa chương trình CUDA
  - Synchronization on GPU
  - Optimization of communication in CUDA
- Ví dụ

Một số khái niệm

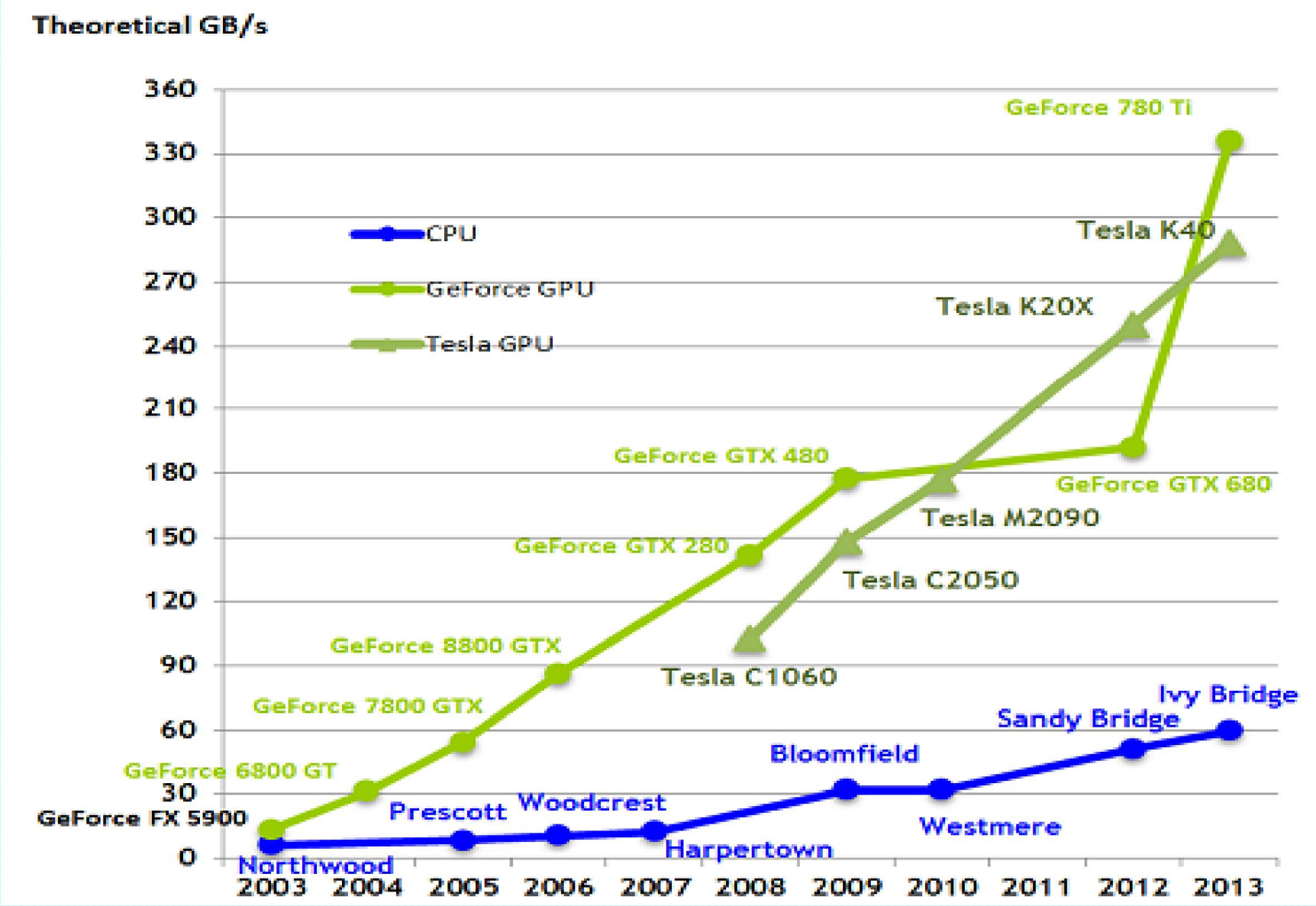
- GPU: Graphic Processing Unit
- Nvidia GPU
- CUDA: Compute Unified Device Architecture
- Programmable GPU
- GPGPU: General Purpose GPU
- GPU computing

GPUs đã có sự phát triển nhanh chóng về tốc độ tính toán trong những năm gần đây

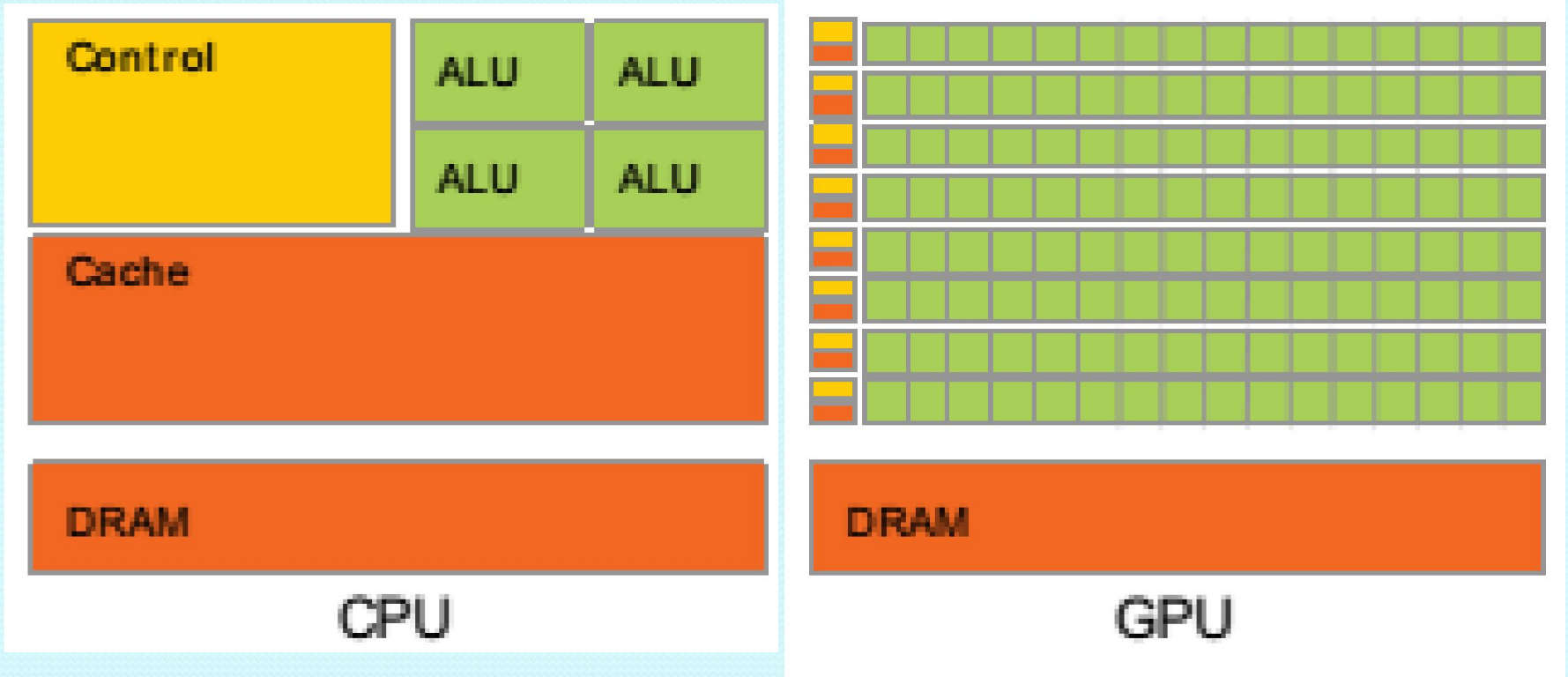
Hiệu năng của GPU và CPU



Tốc độ truy cập bộ nhớ GPU và CPU



Kiến trúc của GPU và CPU



GPUs dùng rất nhiều khối ALU để xử lý song song

Mục tiêu của CPU và GPU

- CPUs chú trọng vào hiệu suất của từng core, cả bộ nhớ cache và khối ALU đều có kích thước lớn
  - 2, 4 cores
  - Mục tiêu: Thực hiện từng lệnh nhanh nhất có thể
- GPUs chú trọng vào xử lý song song
  - Có hàng trăm đến hàng nghìn cores
  - Mục tiêu: Thực hiện nhiều lệnh đồng thời nhất có thể

Ví dụ ứng dụng GPU:

Using GPU to run Next Generation Weather Model

- Models being designed for global cloud resolving scales (3-4km)
- Requires PetaFlop Computers

DOE Jaguar System

- 2.3 PetaFlops
- 250,000 CPUs
- 284 cabinets
- 7-10 MW power
- ~$100 million
- **Reliability in hours**

Equivalent GPU System

- 2.3 PetaFlop
- 2000 Fermi GPUs
- 20 cabinets
- 1.0 MW power
- ~$10 million
- **Reliability in weeks**

< 500 Geforce 780 Ti

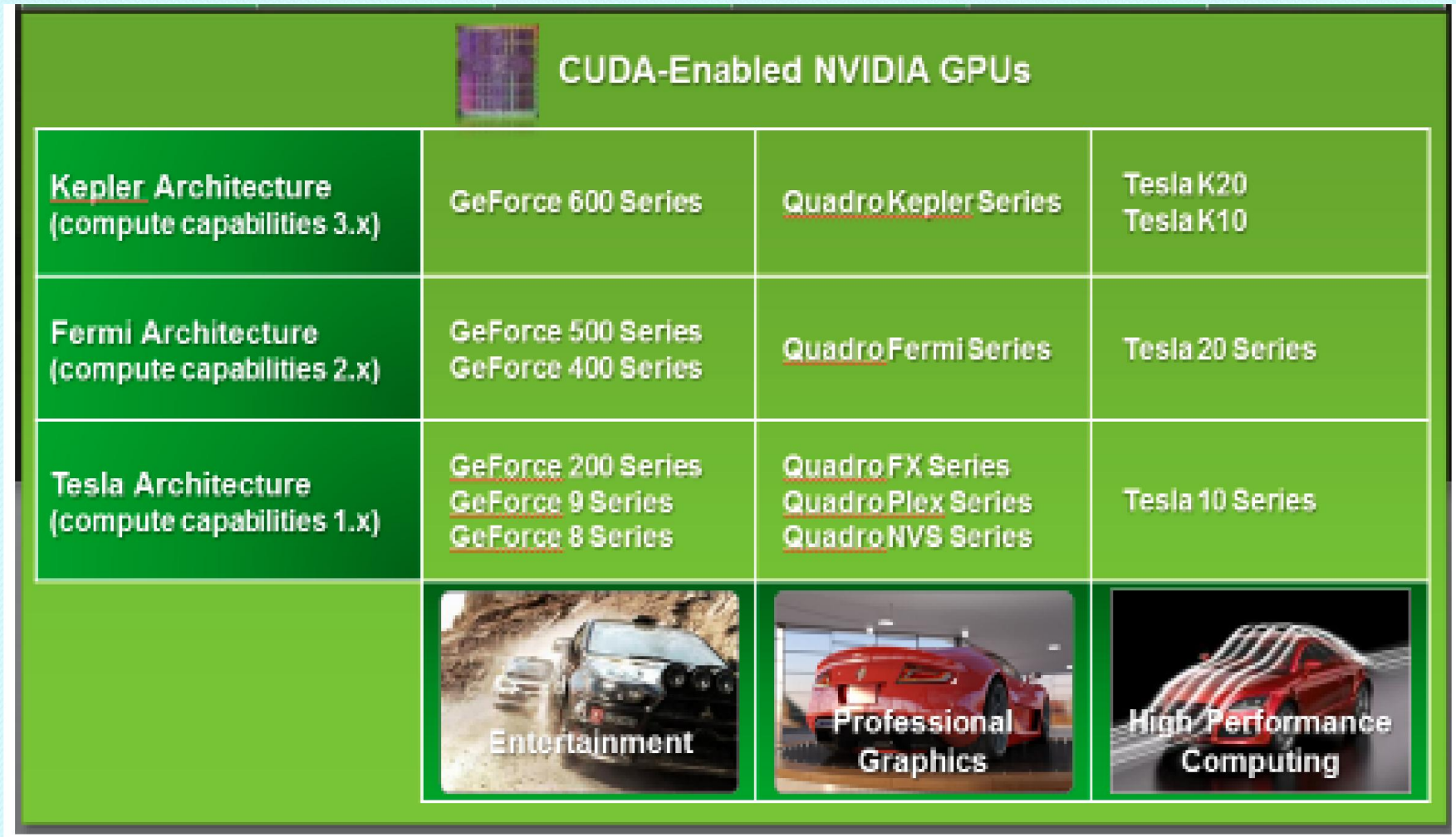
- Large CPU systems (>100 thousand cores) are unrealistic for operational weather forecasting
  - Power, cooling, reliability, cost
  - Application scaling

Valmont Power Plant ~200 MegaWatts Boulder, CO

Các lĩnh vực đã ứng dụng tính toán trên GPU

- GPUs đang được sử dụng như là nền tảng tính toán giá rẻ, tiêu thụ ít năng lượng, hiệu năng tính toán rất cao, áp dụng hiệu quả cho nhiều bài toán tính toán hiệu năng cao
  - Climate and Weather Predictions
  - Bioinformatics
  - Image processing
  - Health
  - … Many other areas

Một số loại GPGPU



GPU Tesla K20

Stream Cores	2496
Max Memory Size	5 GB
Max Memory Bandwidth	208 (GB/sec)
Peak Double Precision floating point performance (GFLOP)	1.17 Tflops
Peak Single Precision floating point performance (GFLOP)	3.52 Tflops
NVIDIA CUDA™ Technology	Yes
Cooling	Active
Dual Slot	Yes
Wattage	225W

Hạ tầng tính toán tại viện ICSE

- Cụm BKPARAM : 5325.8 GFLOPS
- Cụm BKLUSTER : 1664 GFLOPS
- Cụm BKCMS : 1254.4 GFLOPS

Cụm BKPARAM

- Được tài trợ từ Ấn Độ, 11/2013
- Headnode/login node
  - bkparam.hust.edu.vn
  - 2 x Intel Xeon E5-2670 Processor (20M Cache, 2.6 GHz, 8 Core)
  - 32 GB DDR3 RAM
  - 6x300 GB HDD
  - FDR 56Gbps Infiniband
- 15 x Compute nodes
  - 2 x Intel Xeon E5-2670 Processor (20M Cache, 2.6 GHz, 8 Core)
  - 32 GB DDR3 RAM
  - 2x250 GB HDD
  - FDR 56Gbps Infiniband
- Năng lực tính toán lý thuyết: 5325.8 GFLOPS

Liệu GPU có thể thay thế CPU?

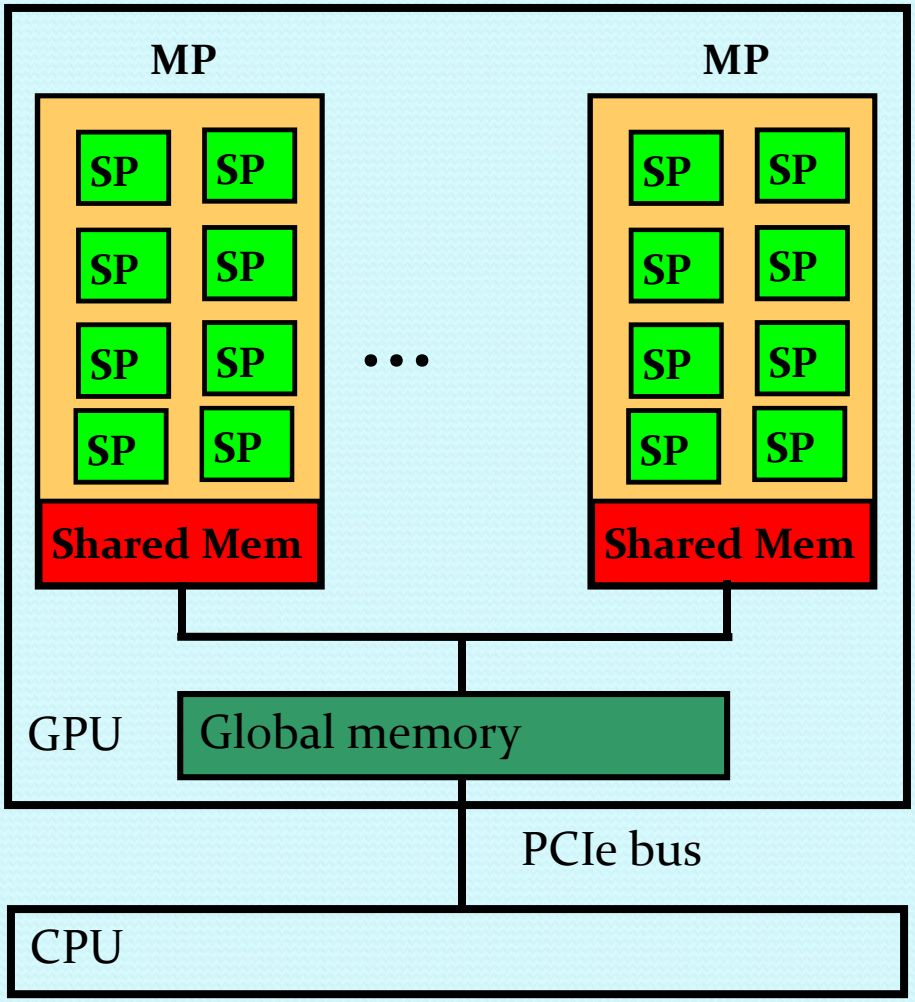
- Không:
  - Không phải vấn đề/bài toán nào cũng có thể được xử lý song song (một cách hiệu quả) trên hàng trăm luồng (core) như trên GPU
  - GPU chỉ hiệu quả đối với những bài toán/vấn đề có dữ liệu phân phối đều và rộng

Mô hình lập trình CUDA
(Compute Unified Device Architecture)

Cài đặt: MCUDA

- MCUDA là một công cụ thiết kế cho Linux để chạy giả lập chương trình CUDA trên CPU (có nhiều core)
- Download và cài đặt:
  - Googling: download MCUDA

Mô hình phần cứng



SP: Single processor

MP: Multiple processor

Mô hình phần mềm

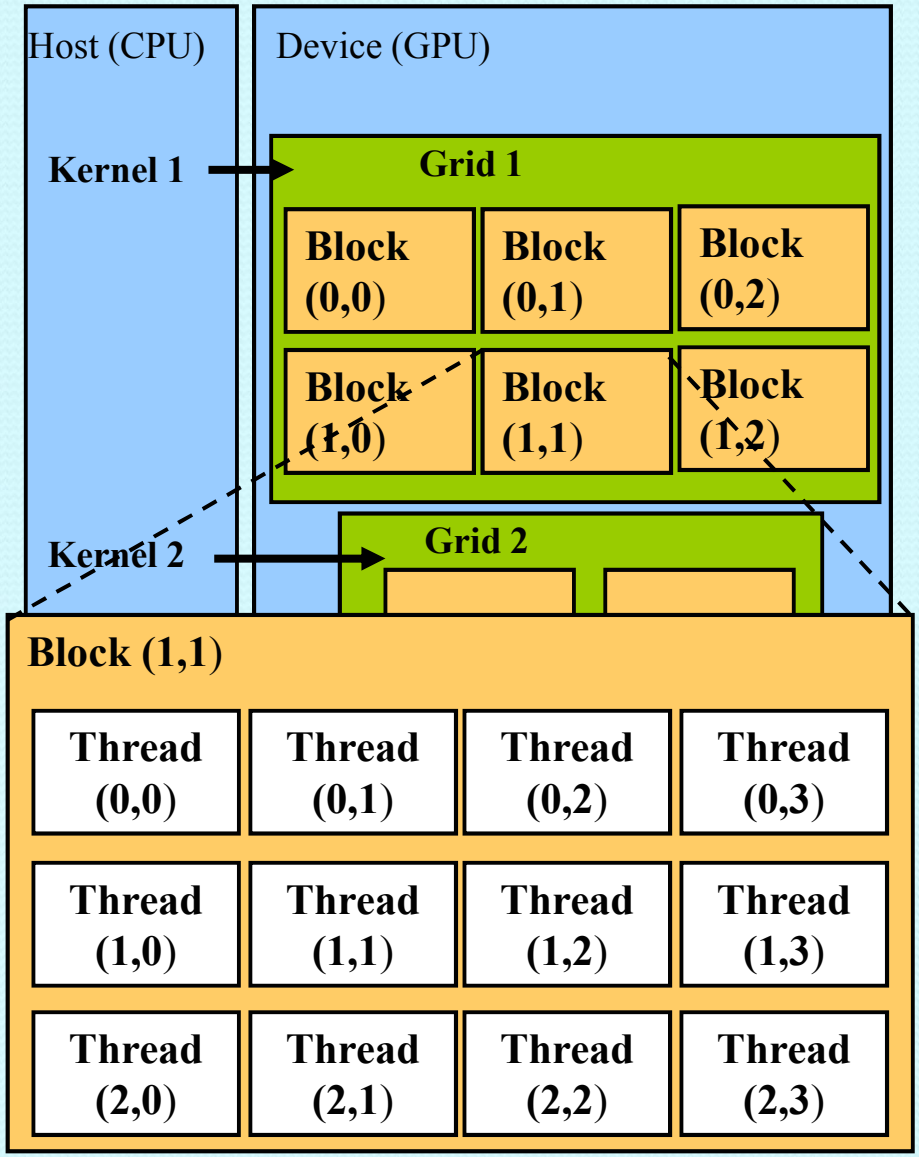
Chương trình CUDA bao gồm 2 phần chính:

- Kernel code: Thông thường là các hàm, gọi từ CPU và thực hiện trên GPU
- Host code: Phần chương trình thực hiện trên CPU

Kernel code

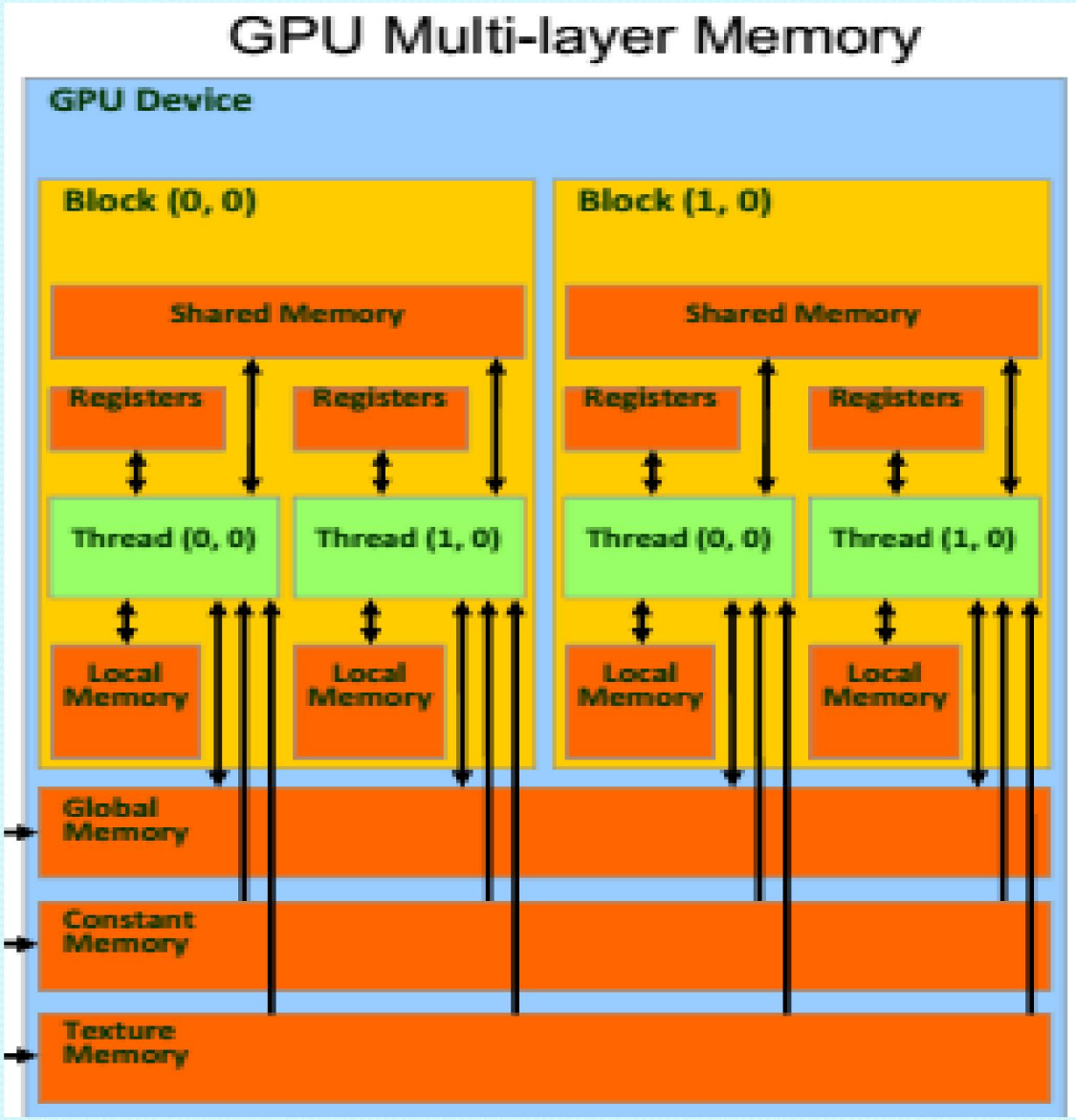
- Thực hiện tính toán trên một lưới các threads (Grid)
- Grid bao gồm một số blocks
- Mỗi block lại bao gồm một số threads

Tổ chức Blocks và Threads



Cấu trúc bộ nhớ

- Per thread:
  - Registers file
  - Local Mem.
- Per block:
  - Shared Mem.
- Per grid:
  - Global Mem.
  - Texture Mem.
  - Constant Mem.



Kernel code: VD bài toán cộng hai vector

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
        int i = threadIdx.x;
        C[i] = A[i] + B[i];
}
```

- Block Dimension: 1
- Number of Block: 1

Kernel code: VD bài toán cộng 2 ma trận

```
__global__ void MatAdd(float A[N][N], float B[N][N],float C[N][N])
{
        int i = threadIdx.x;
        int j = threadIdx.y;
        C[i][j] = A[i][j] + B[i][j];
}
```

- Block Dimension: 2
- Number of Block: 1

Kernel code: VD bài toán cộng 2 ma trận

```
__global__ void MatAdd(float A[N][N], float B[N][N],float C[N][N])
{
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        int j = blockIdx.y * blockDim.y + threadIdx.y;
        if (i < N && j < N)
                C[i][j] = A[i][j] + B[i][j];
}
```

- Block Dimension: 2
- Number of Block: >1

Host code

- Thực hiện một phần tính toán
- Điều khiển việc thực hiện chương trình trên GPU

1) Khai báo biến trên GPU
2) Copy Input CPU -> GPU
3) Define structure: Thread, block
4) Invoke Kernel
5) Copy Output GPU -> CPU
6) Giải phóng bộ nhớ trên GPU

Host code: Khai báo biến trên GPU

```
// Allocate vectors in device memory
float* d_A;
cudaMalloc(&d_A, size);
```

Host code: Copying data

```
// Copy Input CPU -> GPU
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);

// Copy Output GPU -> CPU
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```

Host code: Define structure of Thread & block

```
dim3 dimBlock(Number_Threads_x, Number_Threads_y);

dim3 dimGrid(Number_Blocks_x, Number_Blocks_y);
```

Host code: Invoke Kernel

```
//__global__ void MatAdd(float A[N][N], float B[N][N],float C[N][N])

MatAdd<<<dimGrid,dimBlock>>>(A,B,C);
```

Host code: Giải phóng bộ nhớ trên GPU

```
// Free device memory
cudaFree(d_A);
```

Matrix Mul.: C Function vs CUDA Kernel

```
Void MatrixMul_C ( float *A, float *B, float *C) {
   for ( i = 0 ; i < m ; i++ )
      for ( j = 0 ; j < p ; j++ ){
         *(C + i*p + j) =  0.0;
         for ( k = 0 ; k < n ; k++ )
            *(C + i*p + j)  =  *(C + i*p + j)  + *(A + i*n + k)*(*(B + k*p + j));
      }
   }
```

C Function
A[m][n]xB[n][p]=C[m][p]

```
__global__ void MatrixMul_CUDA(float *A, float *B, float *C) {
   // Calculate the Index (address) of threads in Grid
      int i = blockIdx.y*blockDim.y + threadIdx.y;
      int j = blockIdx.x*blockDim.x + threadIdx.x;
   // Each thread computes a single element of C
      *(C + i*p + j) =  0.0;
      for ( k = 0 ; k < n ; k++ )
         *(C + i*p + j)  =  *(C + i*p + j)  + *(A + i*n + k)*(*(B + k*p + j));
   }
```

CUDA Kernel

Matrix Mul.: C Main vs CUDA Host code

C Main

```
Main {
  // Step 1: Khai báo và cấp phát bộ nhớ
  // Step 2: Gọi hàm nhân 2 ma trận
       MatrixMul_C (A, B, C)
  // Step 3: Giải phóng bộ nhớ cho A, B, C
}
```

CUDA Host code

```
Main {
  // Step 1: Khai báo và cấp phát bộ nhớ CPU và trên GPU
     float * A_gpu; cudaMalloc (&A_gpu, size);  . . . .
  // Step 2: Copy Input from CPU to GPU
     cudaMemcpy(A_gpu, A, size, cudaMemcpyHostToDevice);
  // Step 3: Define structure of threads and blocks:
     dim3  dimBlock(blockX, blockY, blockZ);  dim3  dimGrid(gridX, gridY);
  // Step 4: Gọi hàm nhân 2 ma trận (Invoke Kernel)
       MatrixMul_CUDA <<<dimGrid,dimBlock>>>(A,B,C);
  // Step 5: Copy Output from CPU to GPU
     cudaMemcpy(C, C_gpu, size, cudaMemcpyDeviceToHost);
  // Step 6: Giải phóng bộ nhớ cho trên CPU và trên GPU
     cudaFree(A_gpu); cudaFree(B_gpu); cudaFree(C_gpu);
}
```

Lập trình CUDA khó hay dễ

- Dễ cho những bài toán đơn giản, VD: Matrix mul.
  - Tính toán đơn giản
  - Không có sự phụ thuộc dữ liệu trong tính toán
  - Chỉ cần copy input và output trong quá trình tính toán
- Khó ?

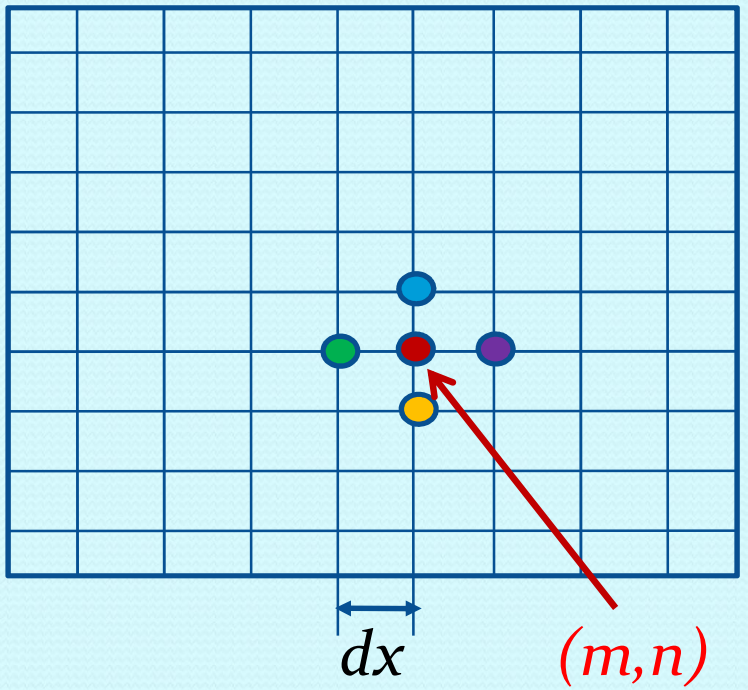
Thách thức

- Bài toán có sự phụ thuộc dữ liệu trong tính toán
  - Cần đồng bộ trong tính toán
  - Cần truyền thông trong tính toán
- Làm thế nào để viết chương trình CUDA tối ưu?
  - Sử dụng mô hình bộ nhớ nhiều cấp của GPU hiệu quả
  - Tối ưu hóa đồng bộ
  - Tối ưu hóa truyền thông
  - . . .
- Lập trình trên mô hình có nhiều GPU kết hợp với một hoặc nhiều CPU
  - Rất khó cài đặt

Bài toán có sự phụ thuộc dữ liệu trong tính toán

- Phương trình nhiệt

$$\nabla^2 C = \frac{\partial^2 C}{\partial x^2} + \frac{\partial^2 C}{\partial y^2}$$



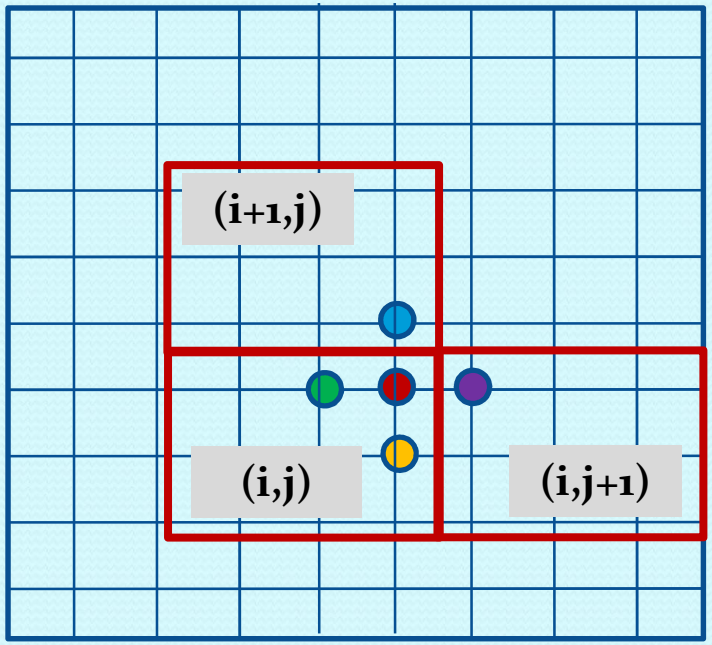
$$\nabla^2 C_{m,n} = \frac{\left(C_{m+1,n} + C_{m-1,n} + C_{m,n+1} + C_{m,n-1} - 4C_{m,n}\right)}{dx^2}$$

- Tính toán trên điểm lưới, e.g. (m,n), cần thông tin tại các điểm lưới lân cận: {(m-1,n), (m+1,n) , (m,n-1) , (m,n+1)} gọi là sự phụ thuộc dữ liệu trong tính toán

Đồng bộ tính toán

- Dữ liệu phải được cập nhật chính xác trước khi được sử dụng
  - Không chắc chắn nếu data được tính trên threads or blocks khác
  - Vì vậy, cần thiết đồng bộ

  Compute=>Synchronization=>Compute
- Đồng bộ giữa các threads trong blocks
  - CUDA: __syncthreads();



- Đồng bộ giữa các blocks trong grid
  - Không thể, vì trong CUDA các blocks được thực hiện một cách độc lập theo một thứ tự bất kỳ
  - Tự thiết kế giải thuật đồng bộ các blocks
  - Thiết kế giải thuật đồng bộ hiệu quả không đơn giản

Truyền thông trong tính toán

- Trong một GPU, tất cả các threads hay blocks dùng chung một global memory
  - Không cần truyền thông
- Trong mô hình nhiều GPU, GPU này không thể truy cập bộ nhớ của GPU khác
  - Cần truyền thông
  - Khó cài đặt và tốn thời gian
  - Thiết kế một giải thuật truyền thông hiệu quả không đơn giản

Ví dụ ứng dụng:

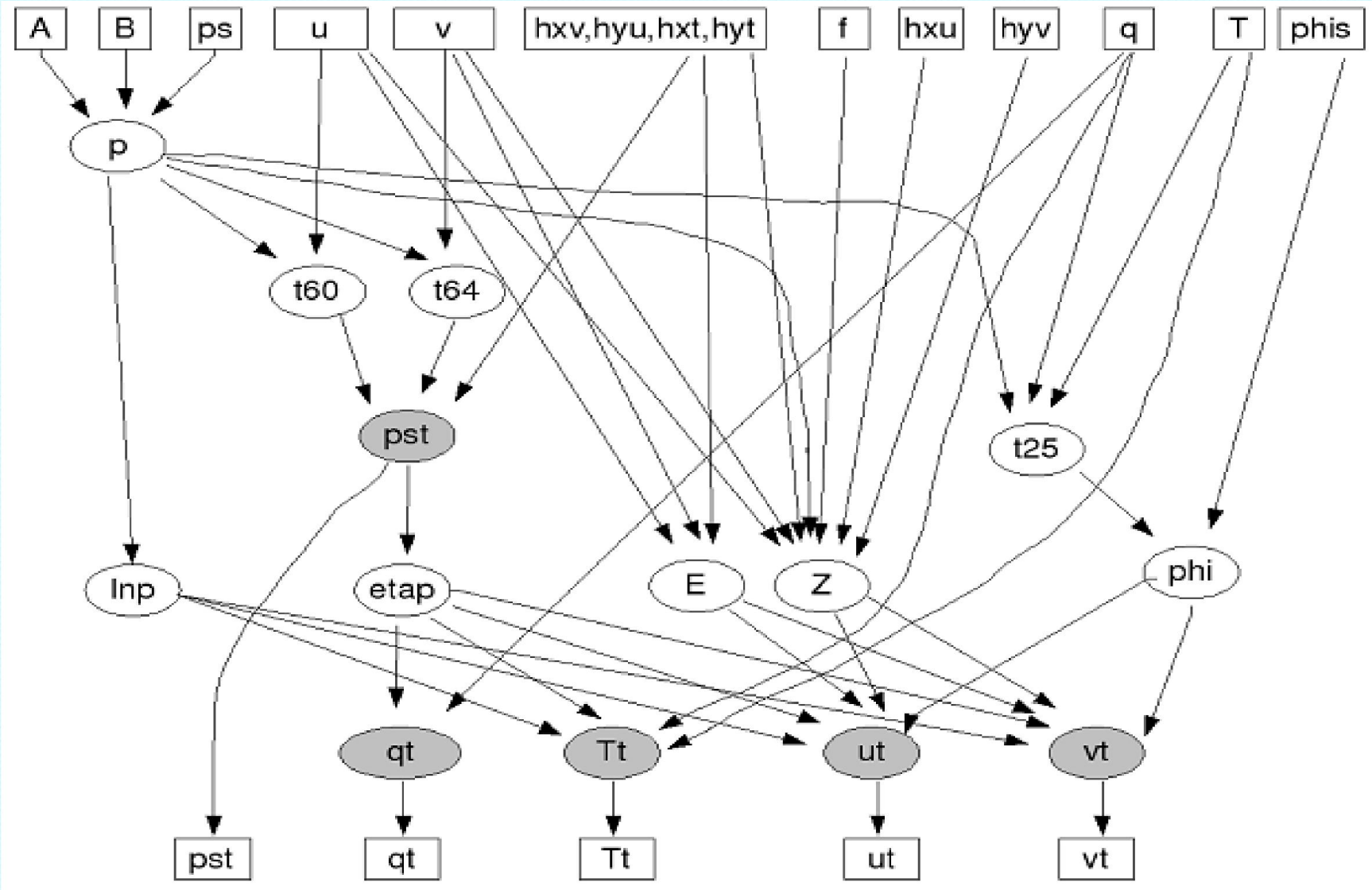
GPUs acceleration of weather forecast model

Weather forecast model

- Khối lượng tính toán rất lớn:
  - Dự báo cho ngày kế tiếp cần tính toán 10^{15} FLOP
  - Thường được chạy trên các Super Computer
- Gồm hai module chính: *Physics, Dynamics*

Weather forecast model: *Dynamics*

- Output: ***pressure ps, temperature T, humidity q, wind (u,v)***



Platform: GTX 480

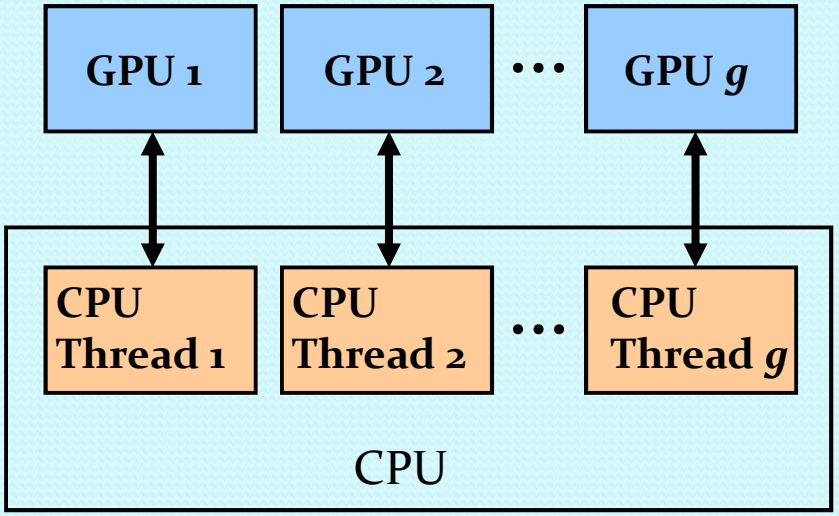
Scalar Processors (SP)	480
Multiple Processors (MP)	15
Processor clock (MHz)	1401
Global memory (MB)	1536
Shared memory per MP (KB)	64
Memory clock (MHz)	1848
Memory bandwidth (GB/sec)	177.4
Peak single-precision performance (Gflops)	1345
Maximum number of threads per block	1024
Recommended maximum number of threads per block	512
Recommended minimum number of threads per block	64
Number of threads that execute in SIMD fashion	32
Number of threads that can access global memory within a single transaction	16

Results

Domain	C code	Non-stream CUDA code			CUDA stream code
		Transfer	Execution	Total	
$512 \times 192 \times 64$	2610	42.2	31.1	73.3	46.9
$512 \times 288 \times 64$	3950	60.7	46.9	107.6	70.1
$512 \times 384 \times 64$	5280	81.4	62.7	144.1	94.4
$512 \times 480 \times 64$	6620	99.8	78.1	177.9	118.1

- Observations:
  - GPU speedup the *Dynamics routine* up to **55 times** over Intel i7-940.
  - The run time of a CUDA program is dominated by transfer time
  - By overlapping communication with calculation, we can decrease over **60%** copy time

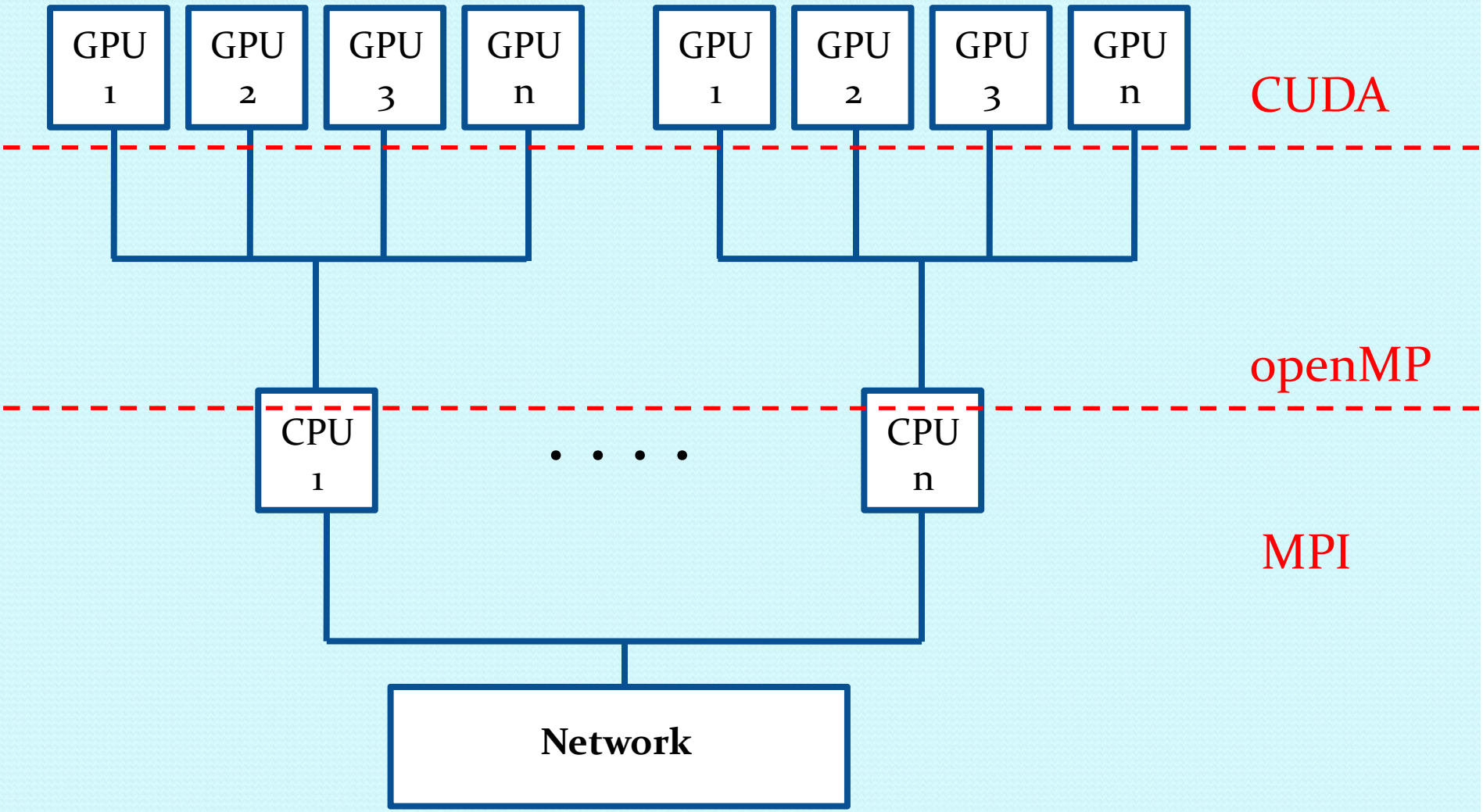
CUDA on multiple GPUs and single CPU




```
// Get the number of available GPUs
   cudaGetDeviceCount(&number_of_gpus);
// Create as many CPU threads
// as the number of GPUs
   omp_set_num_threads(number_of_gpus);
// Calculate in parallel on multiple GPUs
   #pragma omp parallel
    {
     Decompose domain & distribute data to GPUs
     Execute CUDA kernel on multiple GPUs
     Communicate data between GPUs
    }
```

- Use openMP to creates CPU threads and bounds each thread with one GPU

CUDA on multiple GPUs multiple CPUs



Proposition:

"The power of GPUs may threaten the security of using password"