

# Các loại bộ nhớ trong CUDA

Trần Trung Kiên – Phạm Trọng Nghĩa

[ttkien@fit.hcmus.edu.vn](mailto:ttkien@fit.hcmus.edu.vn)

[ptnghia@fit.hcmus.edu.vn](mailto:ptnghia@fit.hcmus.edu.vn)








KHOA CÔNG NGHỆ THÔNG TIN  
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

# Nhìn lại các nội dung đã học

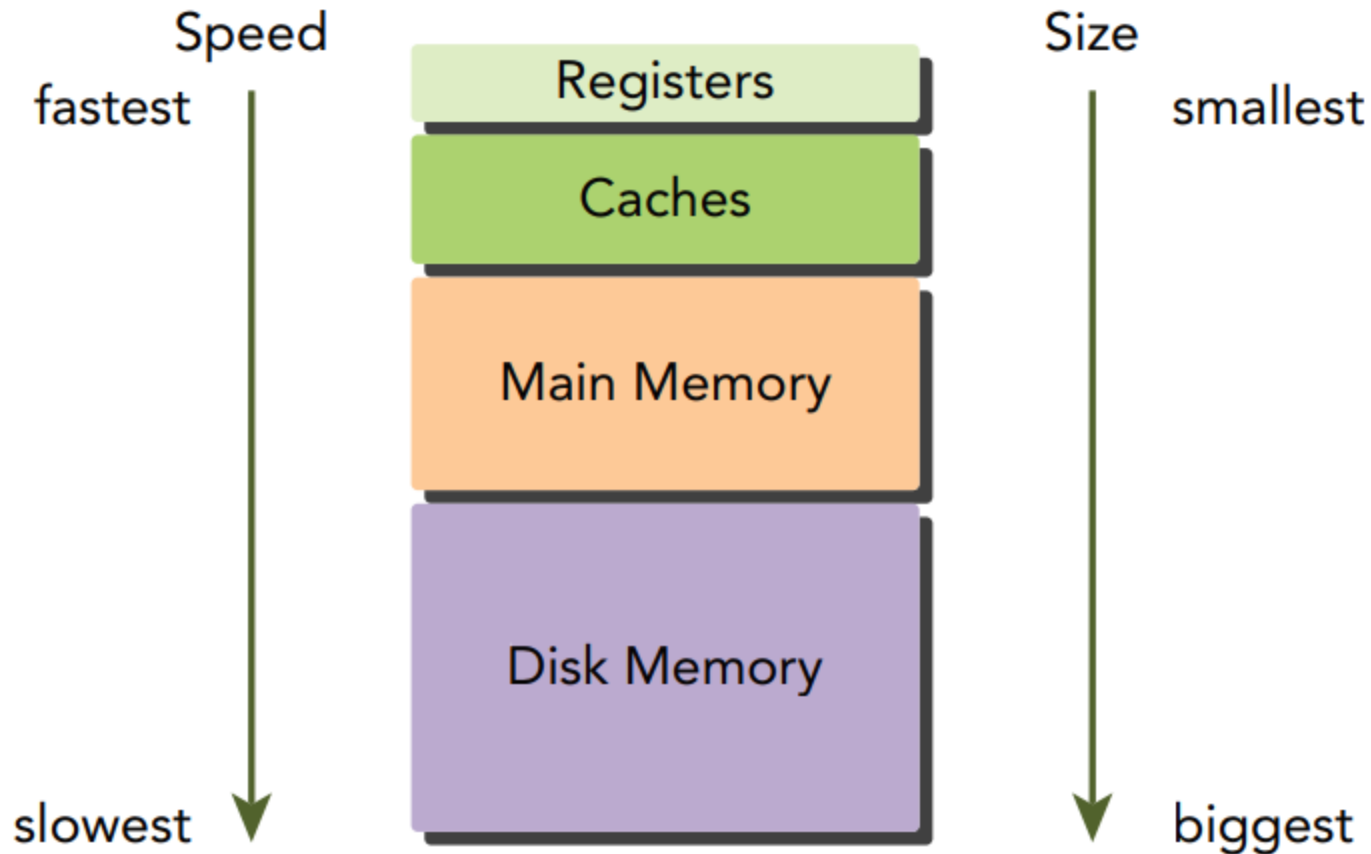
## Nội dung môn học:

- ☒ Giới thiệu CUDA
- ☒ Các dạng tính toán song song thường gặp **Convolution**  
**Reduction**
- ☒ Cách thực thi song song trong CUDA
- Hôm nay*
- ☐ Các loại bộ nhớ trong CUDA
- ☐ Quy trình tối ưu hóa chương trình CUDA
- ☐ Các chủ đề mở rộng (nếu có thời gian)

## Sau khi học xong môn học này, SV có thể:

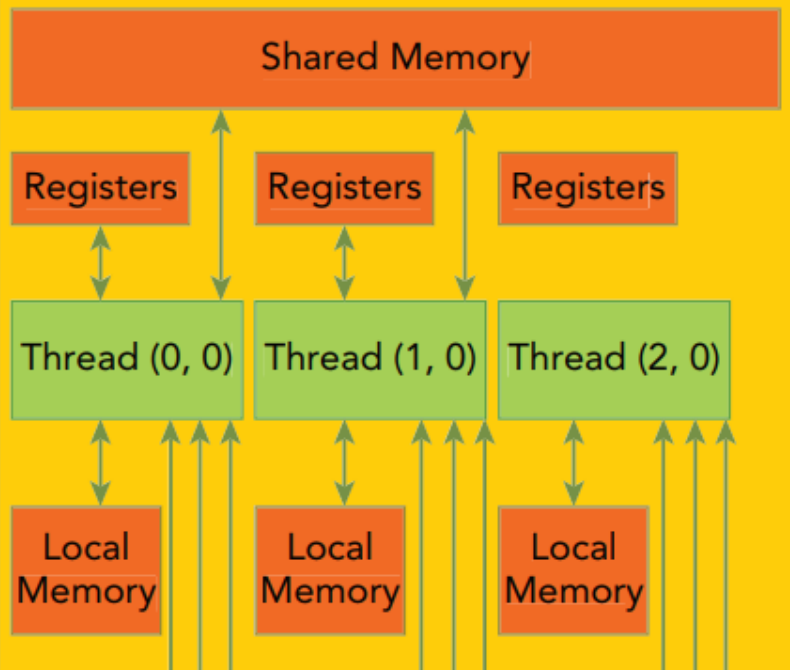
-  Cài đặt được chương trình chạy song song trên GPU bằng CUDA
-  Vận dụng được cách thực thi song song trong CUDA để tăng tốc chương trình
-  Vận dụng được các loại bộ nhớ trong CUDA để tăng tốc chương trình
-  Vận dụng được quy trình tối ưu hóa chương trình CUDA
-  Vận dụng được kỹ năng làm việc nhóm để hoàn thành các bài tập nhóm trong môn học

# Memory Hierarchy



# (Device) Grid

Block (0, 0)



Host

Global  
Memory

Constant  
Memory

Texture  
Memory

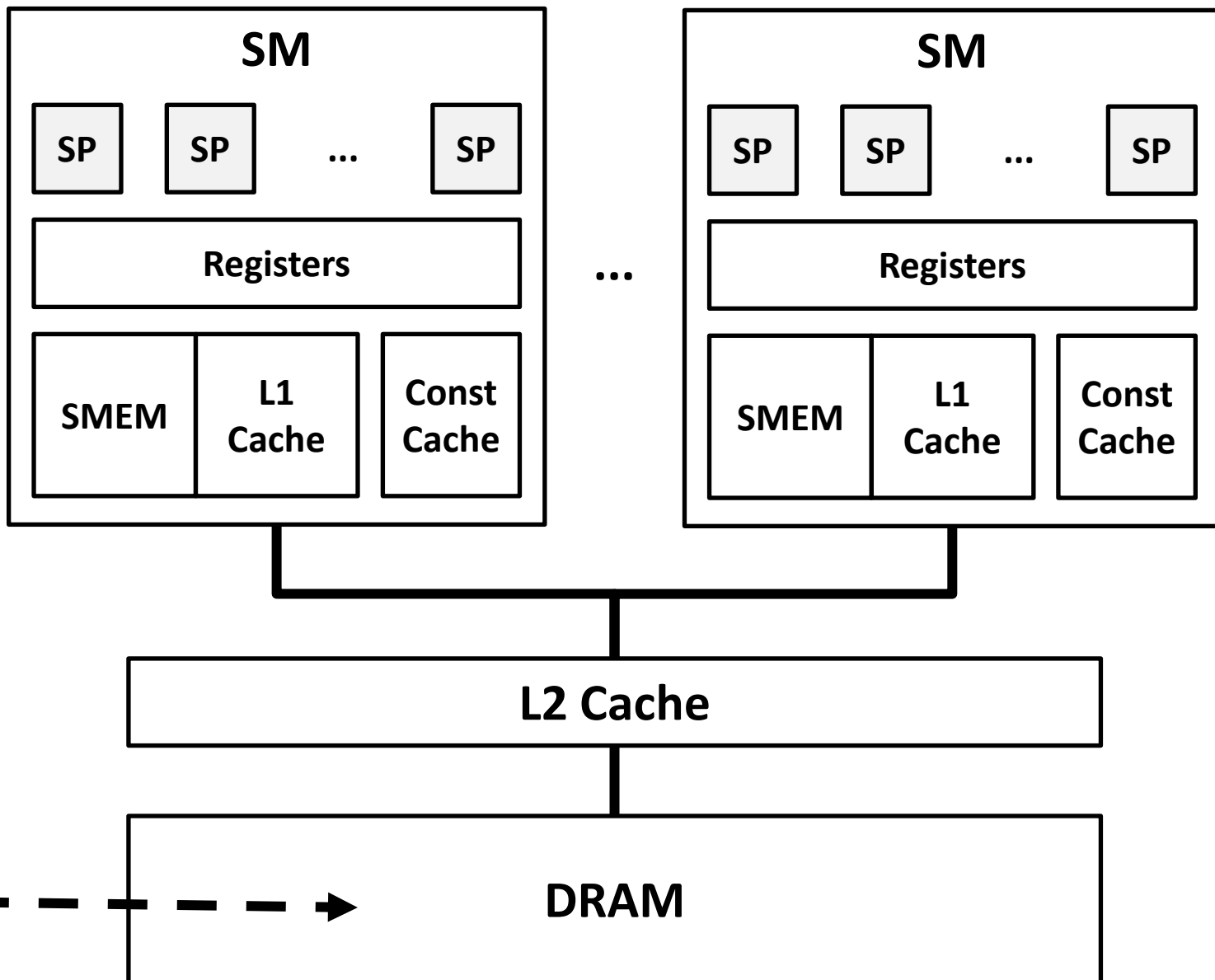
# Tổng thể

- ☐ Bộ nhớ toàn cục
- ☐ Bộ nhớ hằng
- ☐ Bộ nhớ chia sẻ
- ☐ Bộ nhớ thanh ghi

# Bộ nhớ toàn cục (GMEM)

- Khi host dùng hàm `cudaMalloc` để cấp phát vùng nhớ ở device, vùng nhớ này sẽ nằm ở **bộ nhớ toàn cục (GMEM – global memory)** của device
- GMEM là nơi host giao tiếp (chép dữ liệu sang và lấy kết quả về) với device
- GMEM nằm ở DRAM và là bộ nhớ có dung lượng lớn nhất ở device
  - ▣ Truy vấn: `totalGlobalMem` trong struct [`cudaDeviceProp`](#)
  - ▣ Vd, GPU ở server của Khoa có ? GB GMEM
- Nhưng ở device, GMEM là bộ nhớ có tốc độ truy xuất chậm

# Device



Bộ nhớ toàn cục

# Bộ nhớ toàn cục (GMEM)

- Khi host dùng hàm `cudaMalloc` để cấp phát vùng nhớ ở device, vùng nhớ này sẽ nằm ở **bộ nhớ toàn cục (GMEM – global memory)** của device
- GMEM là nơi host giao tiếp (chép dữ liệu sang và lấy kết quả về) với device
- GMEM nằm ở DRAM và là bộ nhớ có dung lượng lớn nhất ở device
  - ▣ Truy vấn: `totalGlobalMem` trong struct [`cudaDeviceProp`](#)
  - ▣ Vd, GPU ở server của Khoa có ? GB GMEM
- Nhưng ở device, GMEM là bộ nhớ có tốc độ truy xuất chậm  
→ *nên tìm cách hạn chế số lần các thread truy xuất GMEM (đây là mục đích của việc sử dụng các loại bộ nhớ khác)*



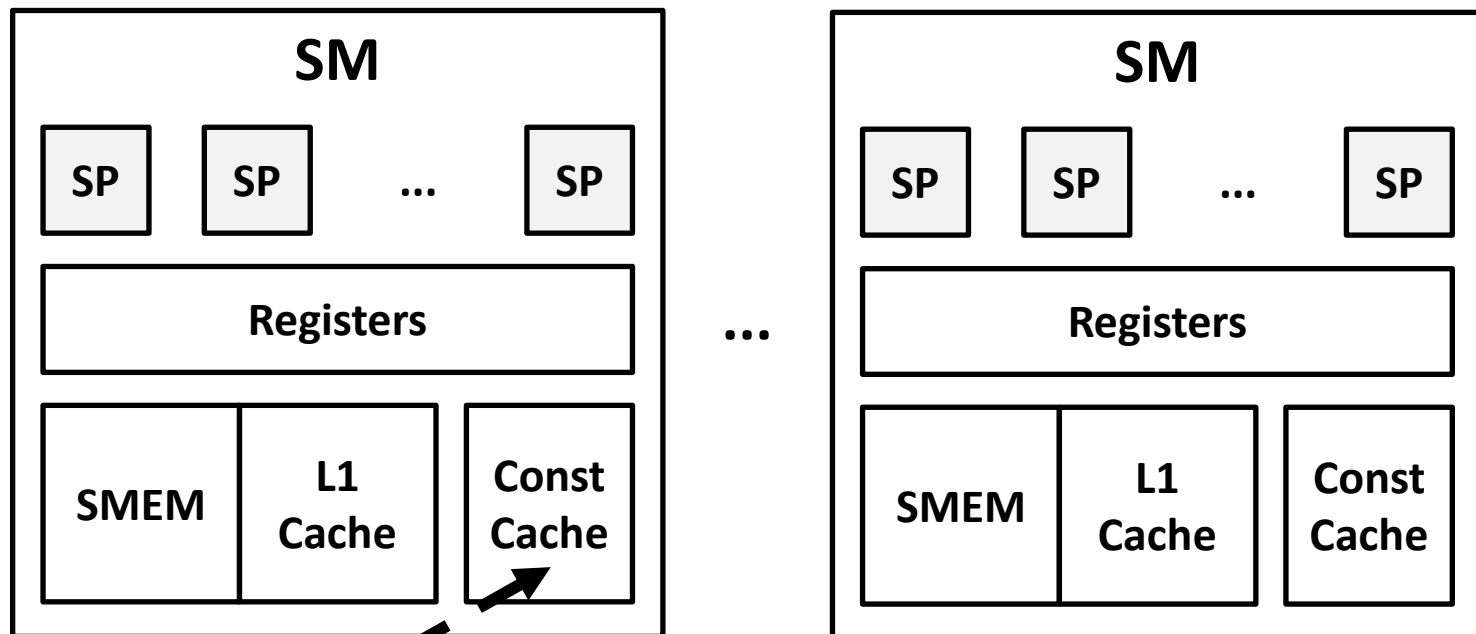
# Bộ nhớ toàn cục (GMEM)

- Ta có thể cấp phát vùng nhớ ở GMEM bằng hàm `cudaMalloc`
  - Host có thể đọc/ghi vùng nhớ này bằng hàm `cudaMemcpy`
  - Con trỏ trỏ tới vùng nhớ này được host truyền vào tham số của hàm kernel
  - Trong hàm kernel, các thread đều có thể truy xuất đến vùng nhớ này thông qua con trỏ được truyền vào
  - Vùng nhớ này sẽ được giải phóng khi host gọi hàm `cudaFree`
- Hoặc cũng có thể khai báo tĩnh biến ở GMEM với từ khóa `__device__`
  - Vd, `__device__ float a[10];`
  - Câu lệnh khai báo phải được đặt ngoài tất cả các hàm
  - Host có thể **đọc/ghi** biến này bằng hàm `cudaMemcpyFrom/ToSymbol`.
    - Tại sao không dùng `cudaMemcpy`?
  - Trong hàm kernel, các thread đều có thể truy xuất đến biến này mà không cần dùng phương pháp truyền tham số vào hàm kernel
  - Biến này sẽ được tự động giải phóng khi chương trình chạy xong

# Bộ nhớ hằng (CMEM)

- Ngoài GMEM, host cũng có thể giao tiếp với device thông qua **bộ nhớ hằng (CMEM – constant memory)**
- Khi nào thì nên dùng CMEM?
  - ▣ Khi host muốn truyền cho device dữ liệu mà **không thay đổi** trong quá trình thực thi hàm kernel
  - ▣ Dữ liệu này cũng phải **nhỏ** vì CMEM chỉ có 64 KB
    - Truy vấn: totalConstMem trong struct [cudaDeviceProp](#)
  - ▣ Các thread trong warp cùng đọc **một dữ liệu chung**
    - CMEM cũng nằm ở DRAM giống GMEM, nhưng có bộ nhớ **Const Cache** ở các SM (8 KB / SM với hầu hết các CC)
    - Const Cache có độ trễ thấp, nhưng lại có băng thông thấp (4B / clock cycle / SM)
      - nếu các thread trong warp không cùng đọc một địa chỉ thì sẽ tốn nhiều lần đọc, nếu cùng đọc một địa chỉ thì chỉ tốn một lần đọc và dữ liệu đọc được sẽ được “broadcast” cho các thread trong warp

# Device



L2 Cache

DRAM

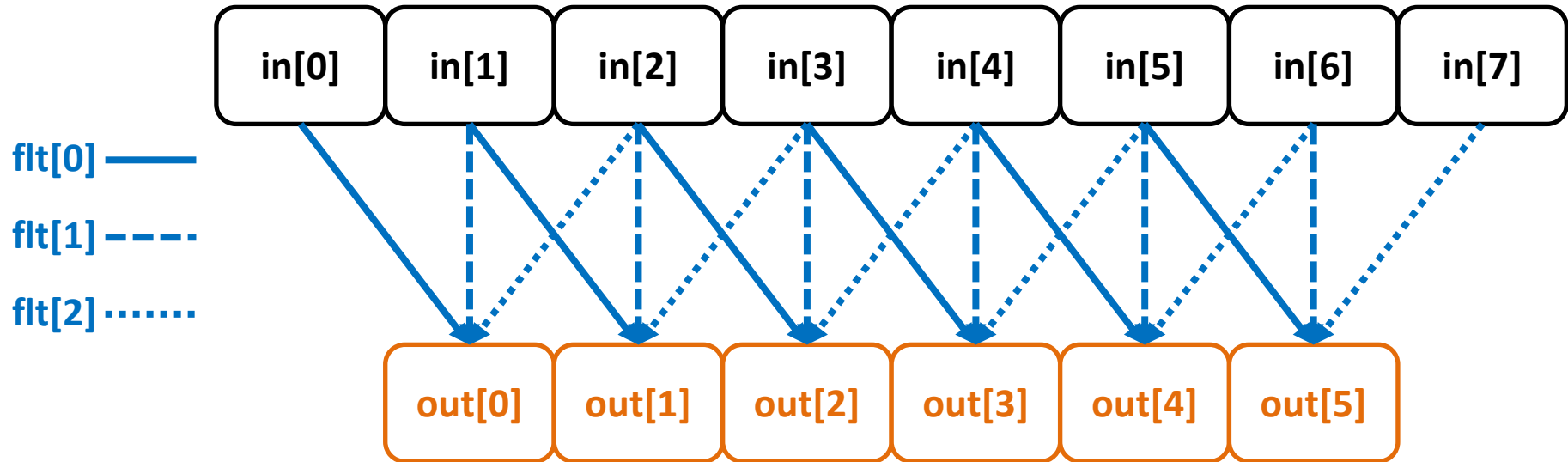
Bộ nhớ hằng

Bộ nhớ toàn cục

# Bộ nhớ hằng (CMEM)

- ☐ Ở device, các tham số của hàm kernel được lưu ở CMEM
- ☐ Các khai báo biến ở CMEM: tương tự như cách khai báo tĩnh biến ở GMEM, nhưng thay từ khóa `__device__` bằng `__constant__`
  - ☐ Vd, `__constant__ float a[10];`
  - ☐ Câu lệnh khai báo phải được đặt ngoài tất cả các hàm
  - ☐ Host có thể **đọc/ghi** biến này bằng hàm `cudaMemcpyFrom/ToSymbol`
  - ☐ Trong hàm kernel, các thread đều có thể đọc (không ghi) biến này mà không cần dùng phương pháp truyền tham số vào hàm kernel
  - ☐ Biến này sẽ được tự động giải phóng khi chương trình chạy xong

# Ví dụ: tính tích chập một chiều



$$\text{out}[0] = \text{in}[0] * \text{flt}[0] + \text{in}[1] * \text{flt}[1] + \text{in}[2] * \text{flt}[2]$$

$$\text{out}[1] = \text{in}[1] * \text{flt}[0] + \text{in}[2] * \text{flt}[1] + \text{in}[3] * \text{flt}[2]$$

$$\text{out}[2] = \text{in}[2] * \text{flt}[0] + \text{in}[3] * \text{flt}[1] + \text{in}[4] * \text{flt}[2]$$

...

`ni = 8`

`nf = 3`

`no = ?`

```

#define NF 100
#define NI (1<<24)
#define NO (NI - NF + 1)
__constant__ float dflt[NF];
...
int main(int argc, char *argv[])
{
    // Set up data for input and filter
    float *in, *flt;
    ...
    // Allocate device memories
    float *d_in, *d_out;
    cudaMalloc(&d_in, NI * sizeof(float));
    cudaMalloc(&d_out, NO * sizeof(float));
    // Copy data from host memories to device memories
    cudaMemcpy(d_in, in, NI * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dflt, flt, NF * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpyToSymbol(dflt, flt, NF * sizeof(float));
    // Launch the kernel
    ...
    // Copy results from device memory to host memory
    cudaMemcpy(out, d_out, NO * sizeof(float), cudaMemcpyDeviceToHost);
    // Free device memories
    cudaFree(d_in);
    cudaFree(d_out);
    ...
}

```

```

#define NF 100
#define NI (1<<24)
#define NO (NI - NF + 1)
__constant__ float d_flt[NF];
...
int main(int argc, char *argv[])
{
    ...
    // Launch the kernel
    dim3 blockSize(512);
    dim3 gridSize((NO - 1) / blockSize.x + 1);
    convOnDevice<<<gridSize, blockSize>>>(d_in, d_out);
    ...
}

```

```

__global__ void convOnDevice(float *d_in, float *d_out)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < NO)
    {
        float s = 0;
        for (int j = 0; j < NF; j++)
        {
            s += d_flt[j] * d_in[ ? ];
        }
        d_out[i] = s;
    }
}

```

```

#define NF 100
#define NI (1<<24)
#define NO (NI - NF + 1)
__constant__ float d_flt[NF];
...
int main(int argc, char *argv[])
{
    ...
    // Launch the kernel
    dim3 blockSize(512);
    dim3 gridSize((NO - 1) / blockSize.x + 1);
    convOnDevice<<<gridSize, blockSize>>>(d_in, d_out);
    ...
}

```

```

__global__ void convOnDevice(float *d_in, float *d_out)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < NO)
    {
        float s = 0;
        for (int j = 0; j < NF; j++)
        {
            s += d_flt[j] * d_in[i + j];
        }
        d_out[i] = s;
    }
}

```



# Thí nghiệm

- ☐ Kích thước mảng đầu vào:  $2^{14}$
- ☐ Phát sinh ngẫu nhiên giá trị số thực trong  $[0, 1]$  cho mảng đầu vào và bộ lọc
- ☐ GPU: GeForce GTX 560 Ti (CC 2.1)
- ☐ So sánh thời gian chạy của hàm kernel (block size 512) khi lưu bộ lọc ở CMEM với khi lưu bộ lọc ở GMEM
  - ☐ CMEM: 17.513 ms
  - ☐ GMEM: 25.099 ms

# Bộ nhớ thanh ghi (RMEM)

Ngoài CMEM với cơ chế cache, ta cũng có thể làm giảm số lần truy xuất DRAM bằng **bộ nhớ thanh ghi (RMEM – Registers)**

```
__global__ void convOnDevice(float *d_in, float *d_out)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < NO)
    {
        float s = 0;
        for (int j = 0; j < NF; j++)
        {
            s += d_flt[j] * d_in[i + j];
        }
        d_out[i] = s;
    }
}
```

Thời gian chạy: 17.513

# Bộ nhớ thanh ghi (RMEM)

Ngoài CMEM với cơ chế cache, ta cũng có thể làm giảm số lần truy xuất DRAM bằng **bộ nhớ thanh ghi (RMEM – Registers)**

```
__global__ void convOnDevice(float *d_in, float *d_out)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < NO)
    {
        float s = 0;
        d_out[i] = 0;
        for (int j = 0; j < NF; j++)
        {
            s += d_flt[j] * d_in[i + j];
            d_out[i] += d_flt[j] * d_in[i + j];
        }
        d_out[i] = s;
    }
}
```

Thời gian chạy: ~~17.513~~ 47.107

# Bộ nhớ thanh ghi (RMEM)

Ngoài CMEM với cơ chế cache, ta cũng có thể làm giảm số lần truy xuất DRAM bằng **bộ nhớ thanh ghi (RMEM – Registers)**

```
__global__ void convOnDevice(float *d_in, float *d_out, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < NO)
    {
        float s = 0;
        d_out[i] = 0;
        for (int j = 0; j < NF; j++)
        {
            s += dflt[j] * d_in[i + j];
            d_out[i] += dflt[j] * d_in[i + j];
        }
        d_out[i] = s;
    }
}
```

- Mỗi thread sẽ có một phiên bản riêng của biến s và được lưu ở RMEM của thread đó
- RMEM là bộ nhớ có tốc độ truy xuất nhanh nhất ở device
- RMEM của thread sẽ được giải phóng khi thread thực thi xong hàm kernel
- Host không thể “thấy” và đọc/ghi RMEM

Ghi kết quả **nhiều lần** xuống RMEM

Ghi kết quả cuối cùng **một lần** từ RMEM xuống GMEM

# Bộ nhớ thanh ghi (RMEM)

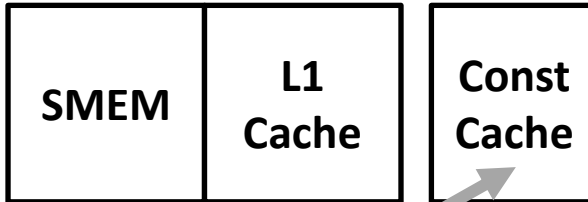
- ☐ Định nghĩa:
  - ☐ Một biến được khai báo không có từ khoá khác đi kèm (`__shared__`, `constant__`,...)
  - ☐ Mảng, nếu phần tử tham chiếu được xác định lúc biên dịch.
- ☐ Scope: thread
- ☐ Lifetime: with the kernel
- ☐ Nhanh nhất, nhưng kích thước hạn chế.

# Bộ nhớ cục bộ (LMEM)

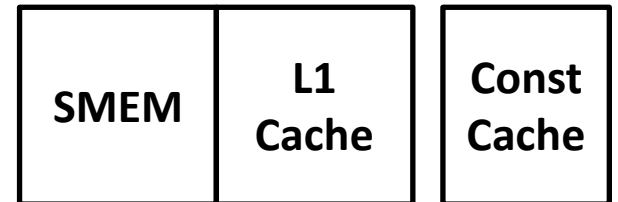
- Tuy có tốc độ nhanh nhất, nhưng RMEM có dung lượng khá hạn chế
  - ▣ Ở hầu hết các CC: 64K thanh ghi / SM, tối đa 255 thanh ghi / thread
- Nếu mỗi thread có lượng dữ liệu lớn hơn dung lượng RMEM cho phép thì sao?
  - ▣ “Tràn” RMEM, dữ liệu bị tràn sẽ được đẩy xuống **bộ nhớ cục bộ (LMEM – local memory)**
  - ▣ LMEM nằm ở DRAM, nhưng có cơ chế cache
    - CUDA document: “Cached in L1 and L2 by default on devices of compute capability 2.x and 3.x; devices of compute capability 5.x cache locals only in L2”
  - ▣ Giống RMEM, LMEM là dành riêng cho mỗi thread và sẽ được giải phóng khi thread thực thi xong

# Device

SM



SM



...

L2 Cache

DRAM

Bộ nhớ thanh ghi

Bộ nhớ cục bộ

Bộ nhớ hằng

Bộ nhớ toàn cục

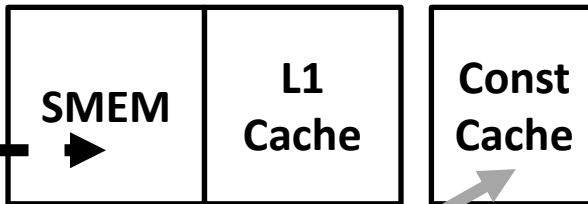
# Bộ nhớ chia sẻ (SMEM)

- ☐ Ngoài CMEM và RMEM, ta cũng có thể làm giảm số lần truy xuất DRAM bằng **bộ nhớ chia sẻ (SMEM – shared memory)**
- ☐ Mỗi block sẽ có một SMEM riêng và sẽ được giải phóng khi block thực thi xong
- ☐ Host không thể đọc/ghi SMEM
- ☐ SMEM nằm ở trên các SM, cùng cấp với L1 Cache và Const Cache → có tốc độ truy xuất nhanh hơn nhiều so với DRAM (mặc dù không bằng RMEM)
- ☐ Ở hầu hết các CC, mỗi SM có 48 hoặc 96 KB SMEM và 48 hoặc 96 KB này được phân ra cho các block chứa trong SM
- ☐ Là “bộ nhớ cache” mà người lập trình có thể kiểm soát được

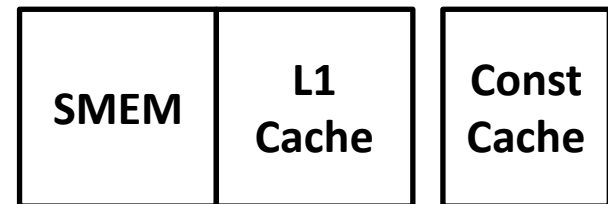


# Device

SM



SM



...

L2 Cache

DRAM

Bộ nhớ thanh ghi

Bộ nhớ chia sẻ

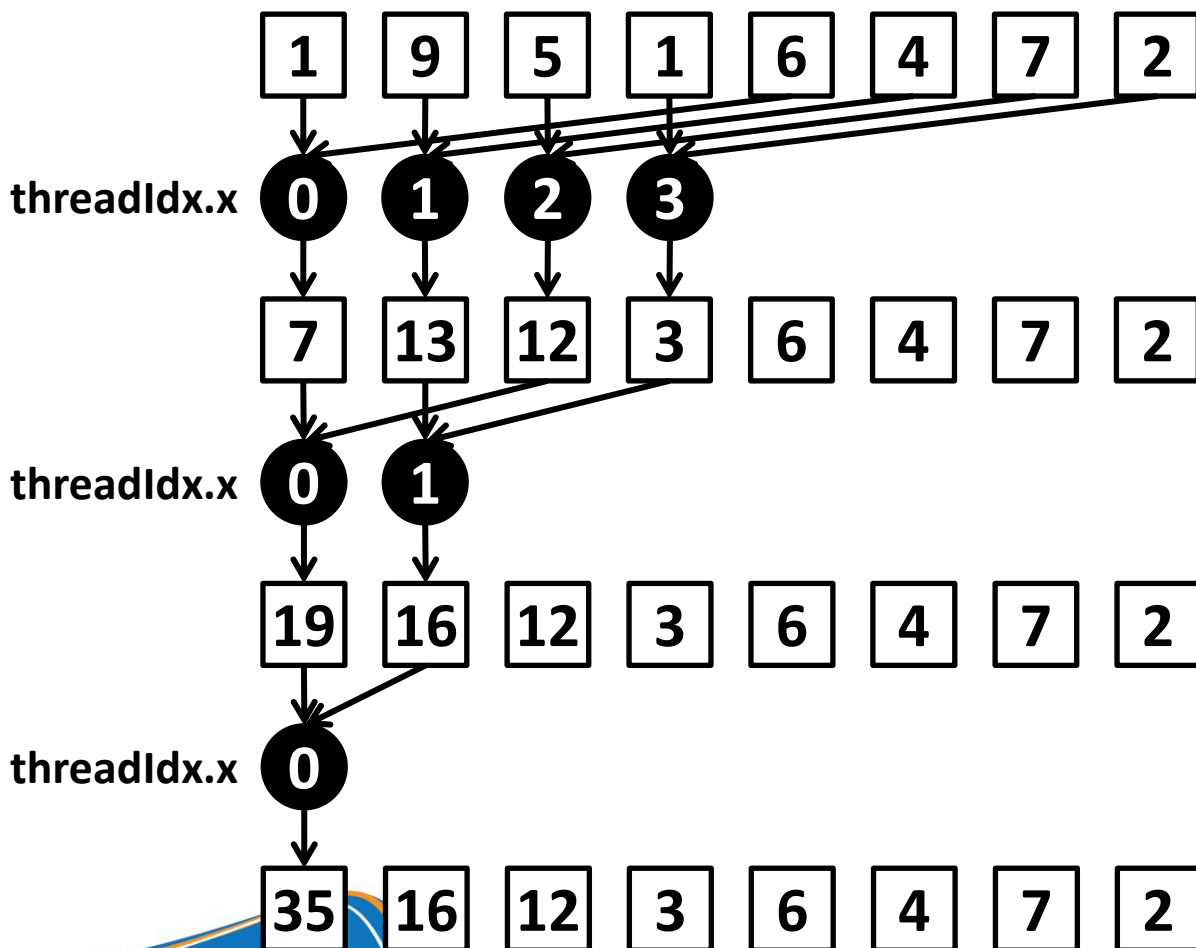
Bộ nhớ cục bộ

Bộ nhớ hằng

Bộ nhớ toàn cục

# Ví dụ 1: bài toán “reduction”

Xét 1 block gồm 4 thread



**Ý tưởng.** Thay vì ở mỗi bước block đều phải đọc ghi DRAM, block sẽ:

- Đọc một lần dữ liệu của block từ GMEM vào SMEM
- Ở mỗi bước, block đọc ghi với dữ liệu trên SMEM
- Cuối cùng, block ghi kết quả từ SMEM xuống GMEM

```

__global__ void reduceOnDevice4(int *in, int *out, int n)
{
    // Each block loads data from GMEM to SMEM
    __shared__ int blkData[2 * 256];
    int numElemsBeforeBlk = blockIdx.x * blockDim.x * 2;
    blkData[?] = in[?];
    blkData[?] = in[?];
    __syncthreads();

    // Each block does reduction with data on SMEM
    for (int stride = blockDim.x; stride > 0; stride /= 2)
    {
        if (threadIdx.x < stride)
        {
            blkData[?] += blkData[?];
        }
        __syncthreads(); // Synchronize within block
    }

    // Each block writes result from SMEM to GMEM
    if (threadIdx.x == 0)
        out[blockIdx.x] = blkData[?];
}

```

Giả sử:

- $2 * \text{blockDim.x} = 2^k$
- N chia hết cho  $2 * \text{blockDim.x}$

Giả sử  $\text{blockDim.x} = 256$

```

__global__ void reduceOnDevice4(int *in, int *out, int n)
{
    // Each block loads data from GMEM to SMEM
    __shared__ int blkData[2 * 256];
    int numElemsBeforeBlk = blockIdx.x * blockDim.x * 2;
    blkData[threadIdx.x] = in[numElemsBeforeBlk + threadIdx.x];
    blkData[blockDim.x + threadIdx.x] = in[numElemsBeforeBlk + blockDim.x + threadIdx.x];
    __syncthreads();

    // Each block does reduction with data on SMEM
    for (int stride = blockDim.x; stride > 0; stride /= 2)
    {
        if (threadIdx.x < stride)
        {
            blkData[threadIdx.x] += blkData[threadIdx.x + stride];
        }
        __syncthreads(); // Synchronize within threadblock
    }

    // Each block writes result from SMEM to GMEM
    if (threadIdx.x == 0)
        out[blockIdx.x] = blkData[0];
}

```

Giả sử:

- $2 * \text{blockDim.x} = 2^k$
- N chia hết cho  $2 * \text{blockDim.x}$

Giả sử  $\text{blockDim.x} = 256$

```

__global__ void reduceOnDevice4(int *in, int *out, int n)
{
    // Each block loads data from GMEM to SMEM
    __shared__ int blkData[2 * 256];
    int numElemsBeforeBlk = blockIdx.x * blockDim.x * 2;
    blkData[threadIdx.x] = in[numElemsBeforeBlk + threadIdx.x];
    blkData[blockDim.x + threadIdx.x] = in[numElemsBeforeBlk + blockDim.x + threadIdx.x];
    __syncthreads();

    // Each block does reduction with data on SMEM
    for (int stride = blockDim.x; stride > 0; stride /= 2)
    {
        if (threadIdx.x < stride)
        {
            blkData[threadIdx.x] += blkData[threadIdx.x + stride];
        }
        __syncthreads(); // Synchronize within threadblock
    }

    // Each block writes result from SMEM to GMEM
    if (threadIdx.x == 0)
        out[blockIdx.x] = blkData[0];
}

```

Giả sử:

- $2 * blockDim.x = 2^k$
- N chia hết cho  $2 * blockDim.x$

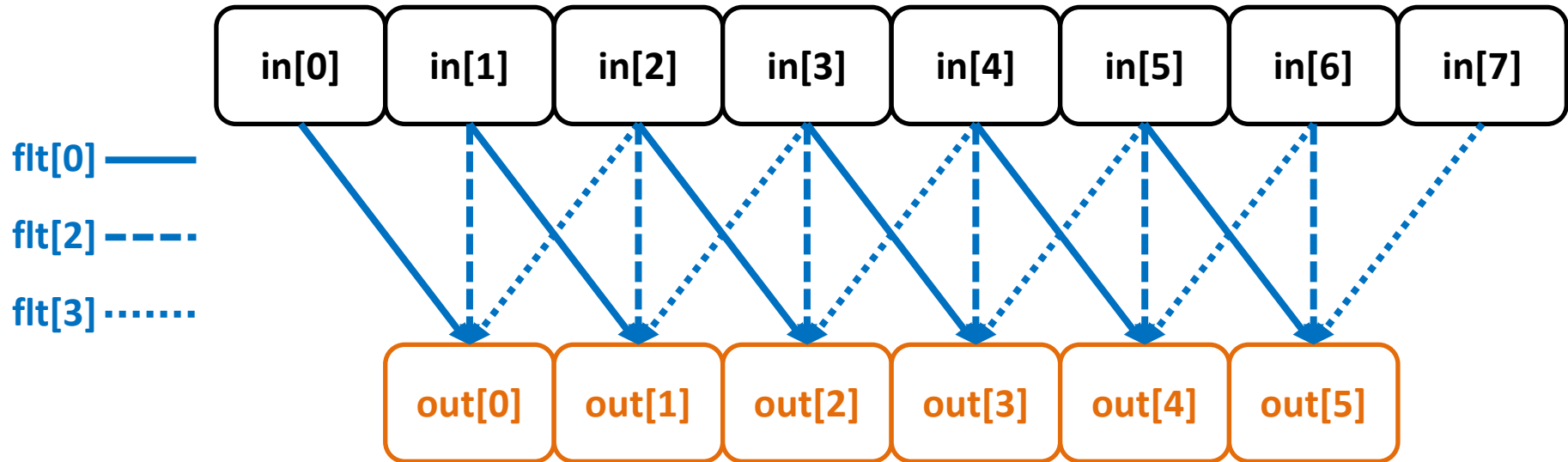
Giả sử  $blockDim.x = 256$

Bỏ đi được không?

# Thí nghiệm

Function	Kernel time (ms)
reduceOnDevice1	6.937
reduceOnDevice2	4.968
reduceOnDevice3	4.250
reduceOnDevice4	3.029

# Ví dụ 2: bài toán tính tích chập một chiều



Ta thấy mỗi phần tử `in[i]` được dùng chung cho 3 thread cạnh nhau  
 → Mỗi block đọc dữ liệu của mình từ GMEM vào SMEM (mỗi phần tử ở GMEM được đọc **một lần**); sau đó, dữ liệu ở SMEM này được **dùng lại nhiều lần** cho các thread trong block

# Ví dụ 2: bài toán tính tích chập một chiều

Code: BT3 😊



# Tổng kết

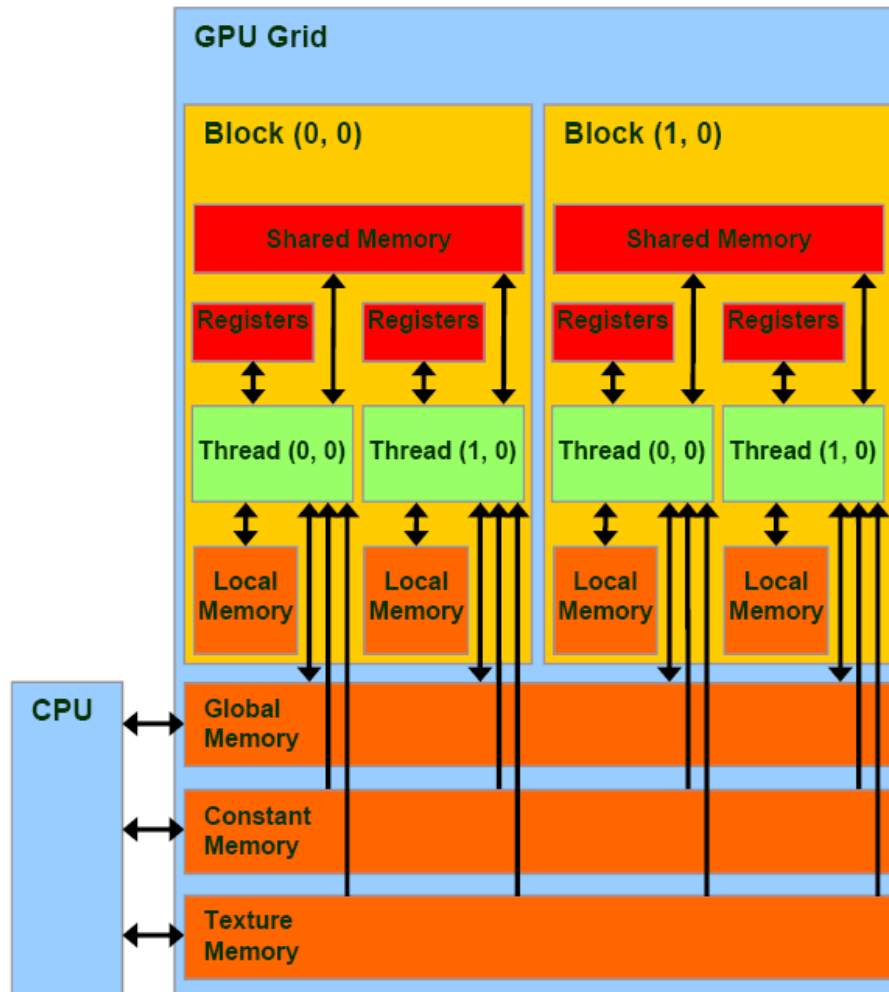
QUALIFIER	VARIABLE NAME	MEMORY	SCOPE	LIFESPAN
	<code>float var</code>	Register	Thread	Thread
	<code>float var[100]</code>	Local	Thread	Thread
<code>__shared__</code>	<code>float var †</code>	Shared	Block	Block
<code>__device__</code>	<code>float var †</code>	Global	Global	Application
<code>__constant__</code>	<code>float var †</code>	Constant	Global	Application

☐ † Có thể là biến scalar hay array

# Tổng kết

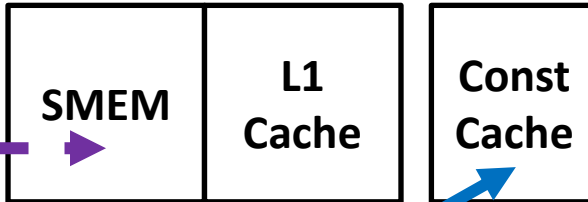
Tận dụng các bộ nhớ có tốc độ cao để giảm số lần truy xuất xuống DRAM

**Giá phải trả:** có thể sẽ làm occupancy giảm xuống (Vd, nếu SM có 48 KB SMEM và block dùng đến 40 KB SMEM thì trong SM chỉ có thể chứa được một block)

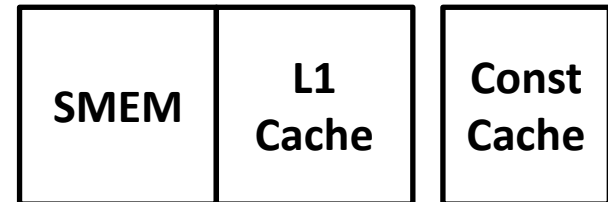


# Device

SM



SM



...

L2 Cache

DRAM

Bộ nhớ thanh ghi

Bộ nhớ chia sẻ

Bộ nhớ cục bộ

Bộ nhớ hằng

Bộ nhớ toàn cục