Parallel Programming

Memory architecture in CUDA

Phạm Trọng Nghĩa
ptnghia@fit.hcmus.edu.vn

Review:

- How to write a CUDA kernel function

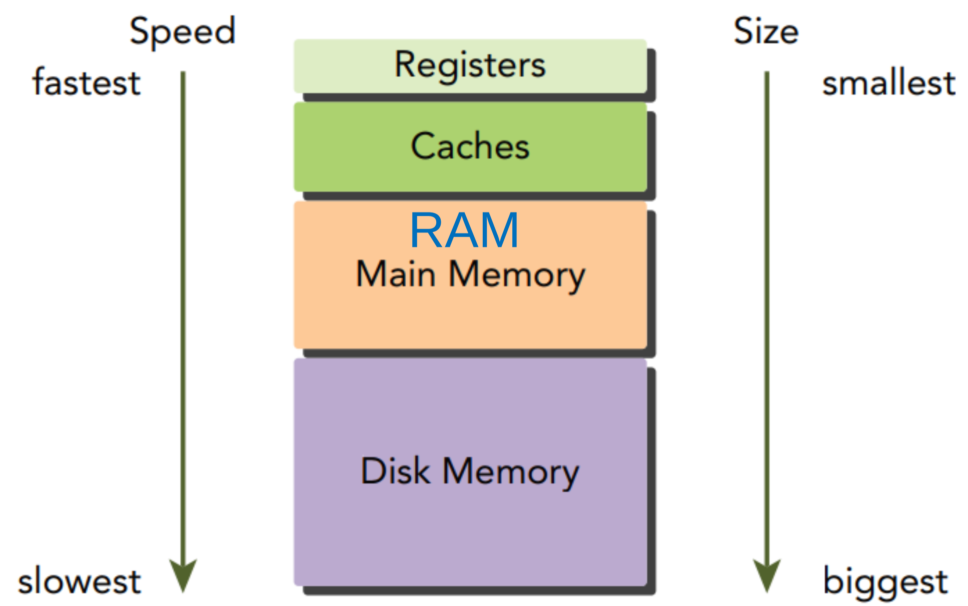
tổ chức

- How to configure and coordinate its execution by a massive number of threads.
- The compute architecture of current GPU hardware
- How threads are scheduled to execute on this hardware.

This lecture:

- So far we only use global memory
  - **Off-chip** DRAM, have long access latency and finite access bandwidth.
- GPUs provide a number of additional **on-chip** memory resources for accessing data that can remove the majority of traffic to and from the global memory
- We will study the use of different memory types to boost the execution performance of CUDA kernels.
  - How one can organize and position data for efficient access by a massive number of threads.

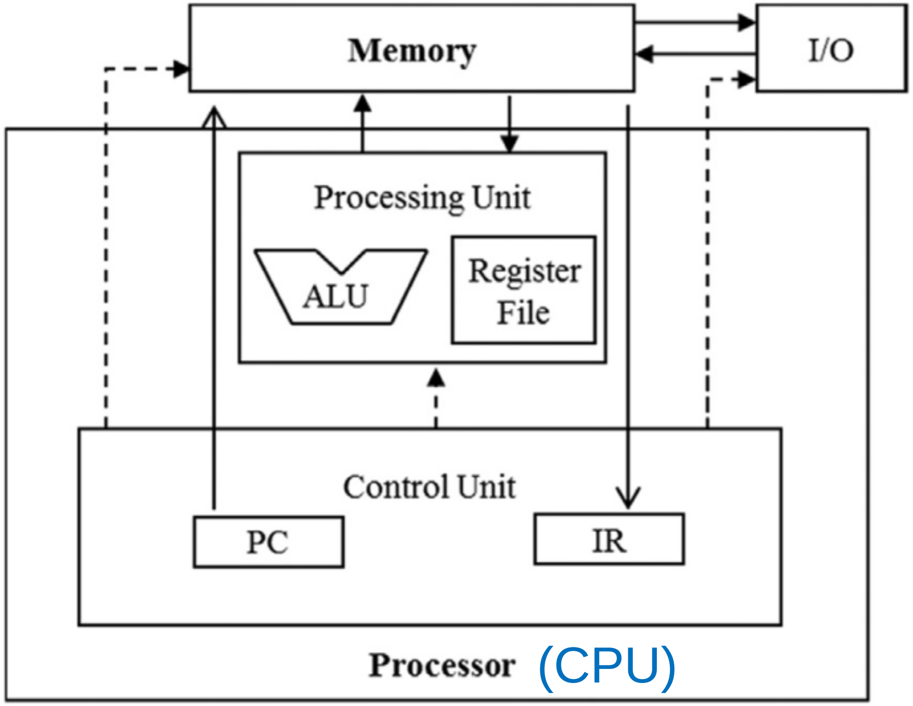
(CPU) Memory hierarchy


Speed
fastest
slowest
Registers
Caches
RAM
Main Memory
Disk Memory
Size
smallest
biggest

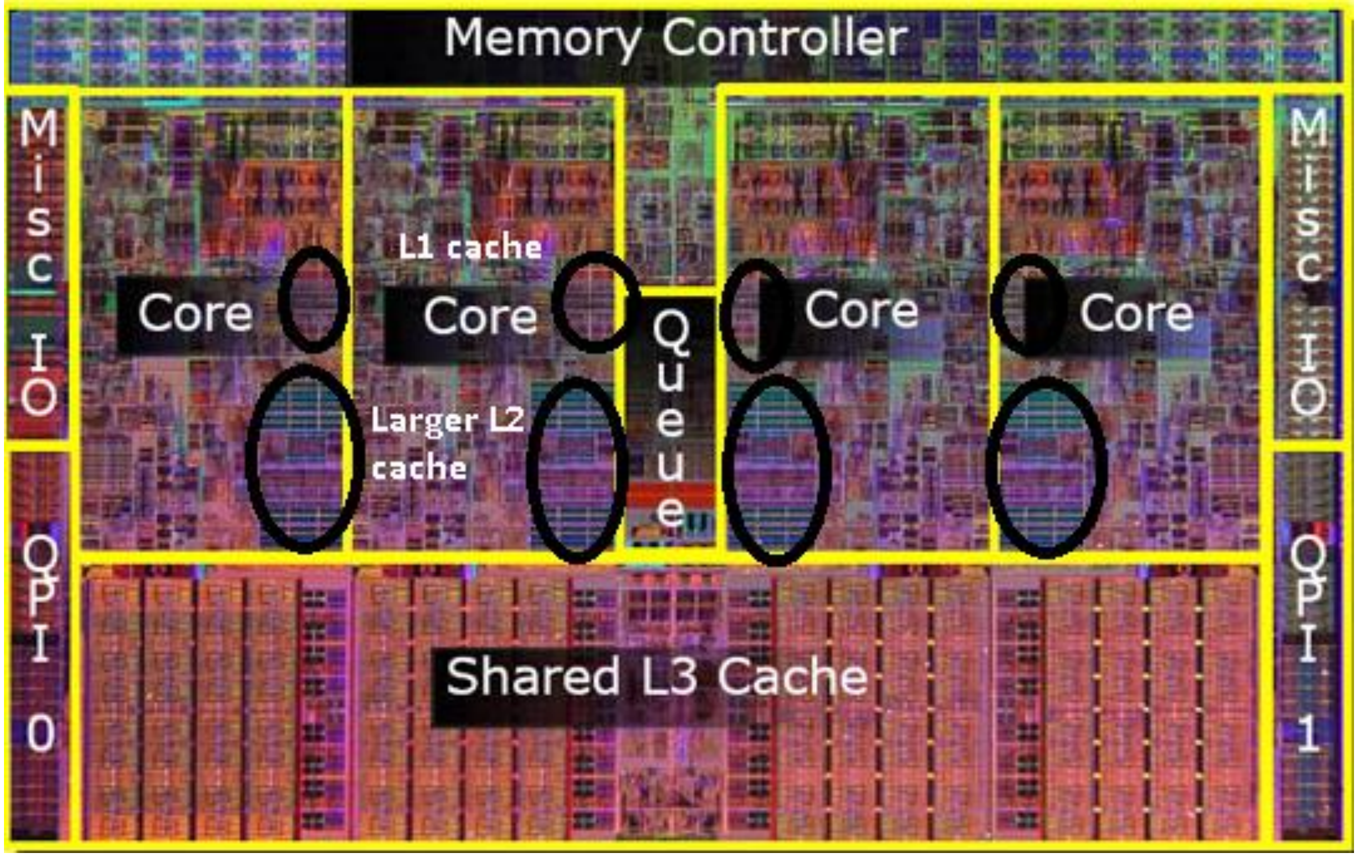
- Going down from top:
  - Lower cost per bit
  - Higher capacity
  - Higher latency
  - Less frequently accessed by the processor

Memory & Registers in Von-Neumann Model

- Registers
  - Fast: 1 cycle; no memory access required
  - Few: hundreds for CPU, O(10k) for GPU SM
- Memory (RAM)
  - Slow: hundreds of cycles
  - Huge: GB or more


Memory
I/O
Processing Unit
ALU
Register File
Control Unit
PC
IR
Processor
(CPU)

(CPU) Memory hierarchy


Memory Controller
Misc IO
Core
L1 cache
Core
Queue
Core
Core
Misc IO
Larger L2 cache
QPI 0
Shared L3 Cache
QPI 1

CUDA Memory Model

- CUDA programming model exposes more of the memory hierarchy and gives you more explicit control

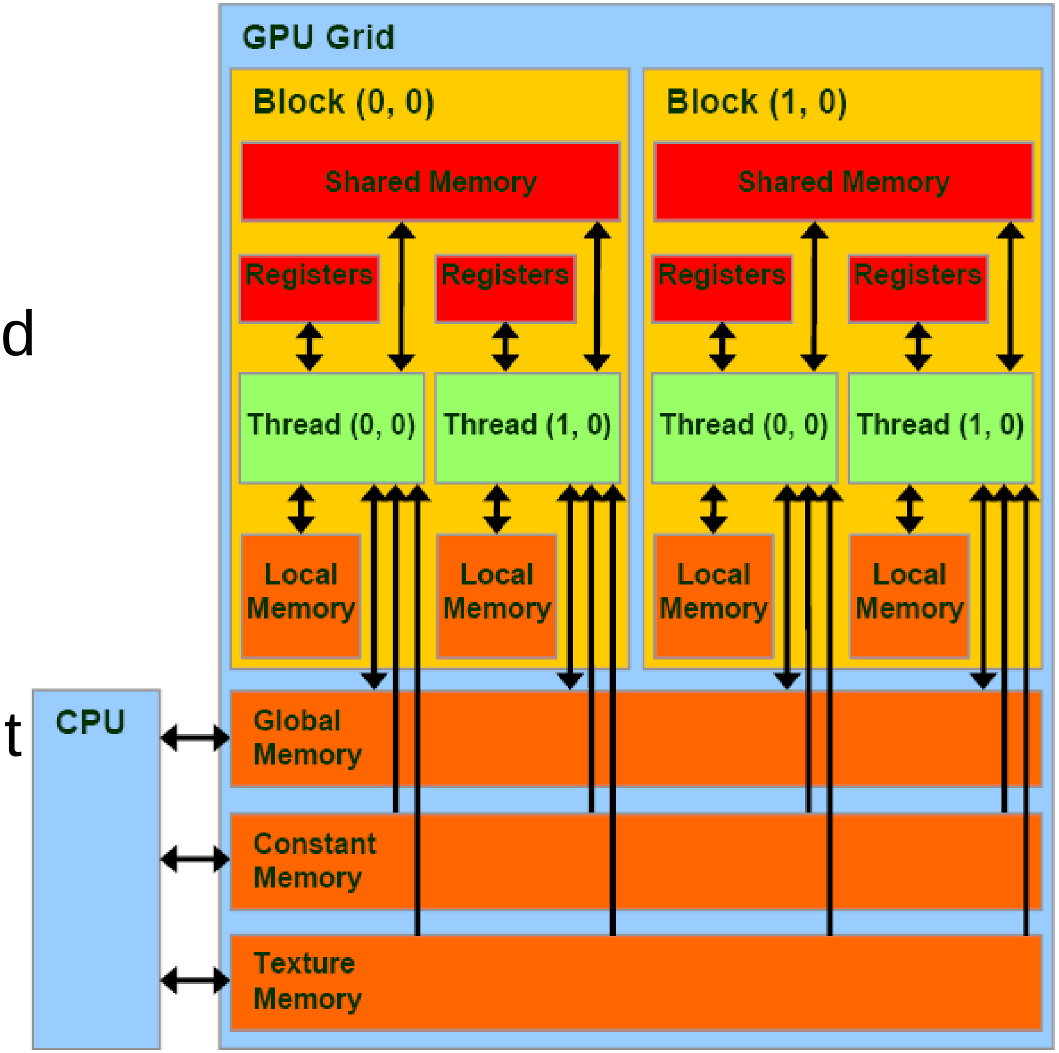
- Global memory $\xleftrightarrow{\text{read/write}}$ grid
- Constant memory đọc nhanh hơn global $\xrightarrow{\text{read}}$ grid
- Shared memory mỗi block có 1 cái $\xleftrightarrow{\text{read/write}}$ block
- Register memory, local memory $\xleftrightarrow{\text{read/write}}$ thread

CUDA Memory Model

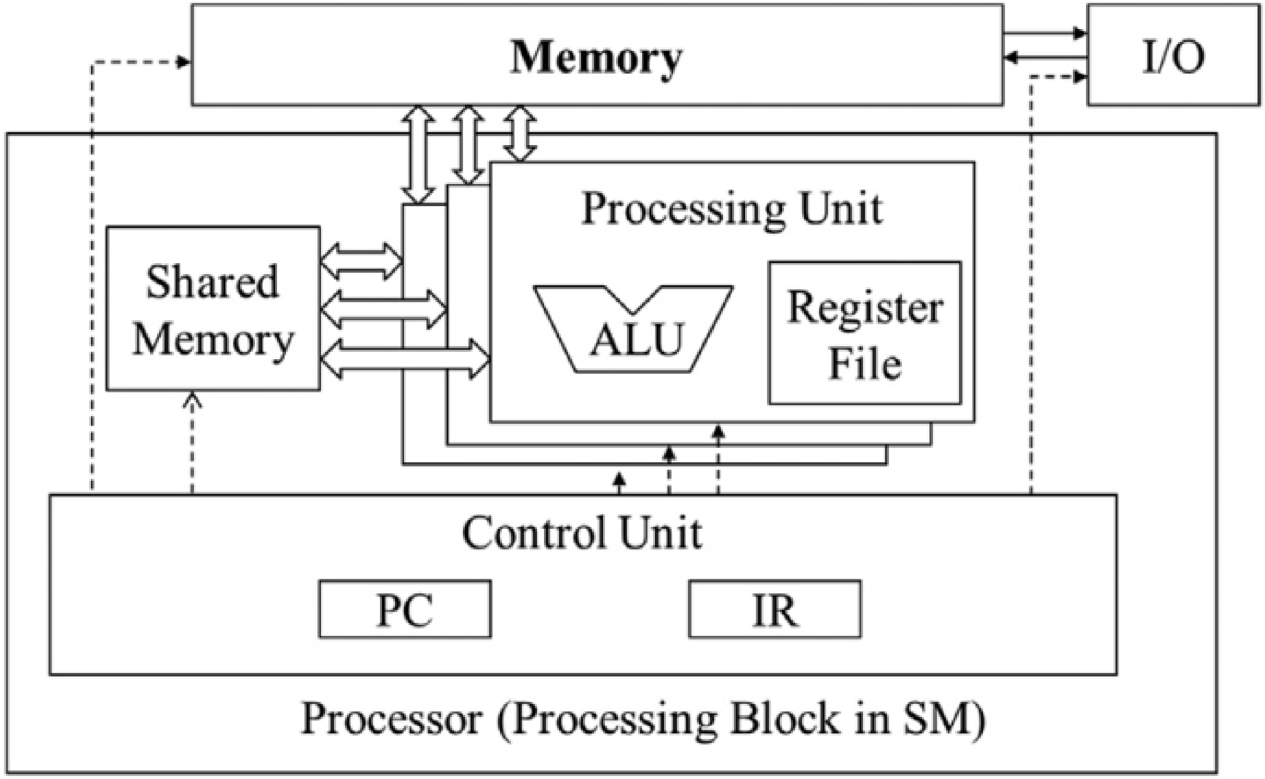
Each thread can:

- Read/write per-thread registers (~1 cycle)
- Read/write per-block shared memory (~5 cycles)
- Read/write per-grid global memory (~500 cycles)
- Read only per-grid constant memory (~5 cycles with caching)

Bên trong device


GPU Grid
Block (0, 0)
Block (1, 0)
Shared Memory
Shared Memory
Registers
Registers
Registers
Registers
Thread (0, 0)
Thread (1, 0)
Thread (0, 0)
Thread (1, 0)
Local Memory
Local Memory
Local Memory
Local Memory
CPU
Global Memory
Constant Memory
Texture Memory

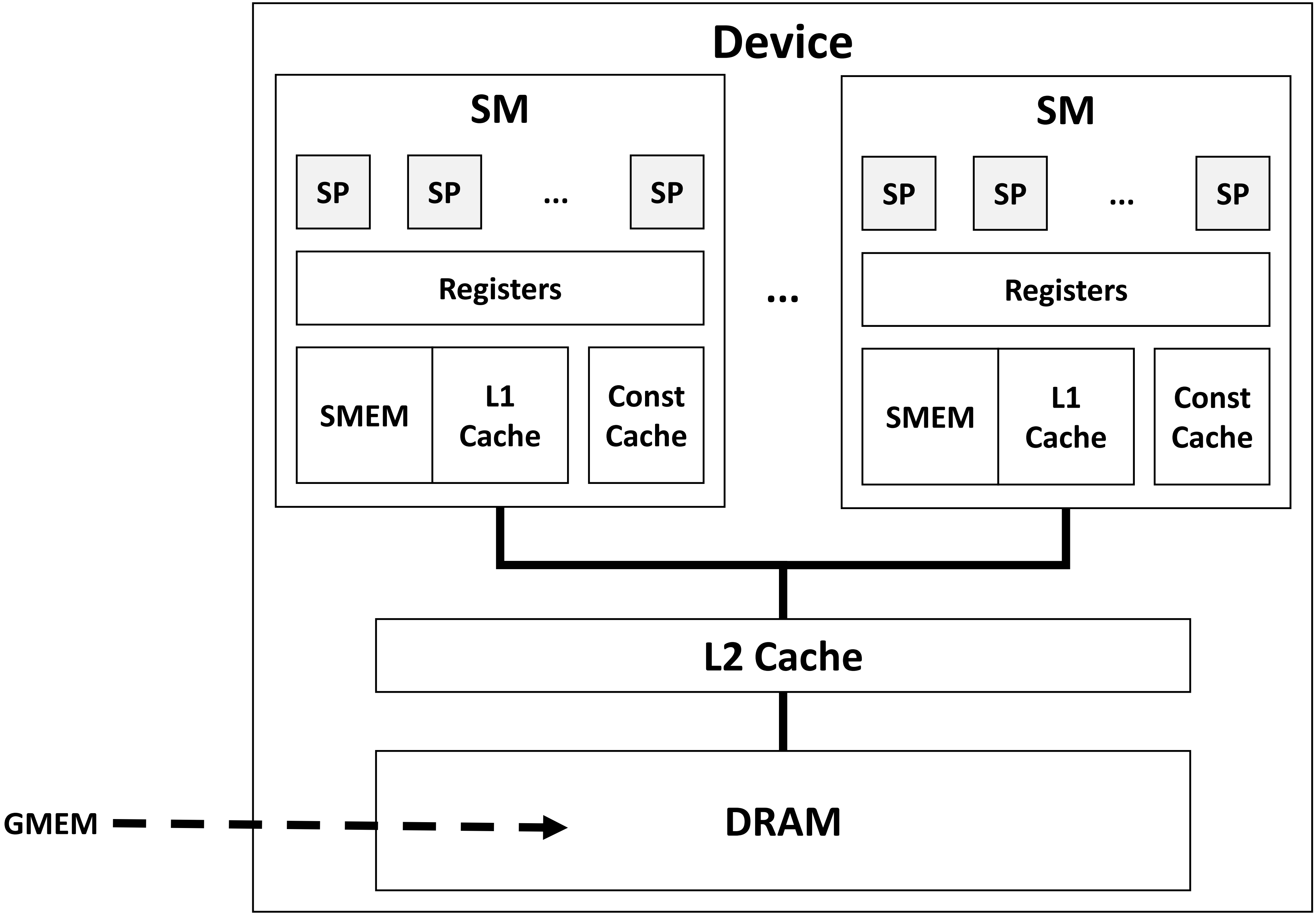
Hardware View of CUDA Memories


Memory
I/O
Processing Unit
Shared Memory
ALU
Register File
Control Unit
PC
IR
Processor (Processing Block in SM)

1 SM

Global memory (GMEM)

- When host calls `cudaMalloc` to allocate a memory region in device, this region will lie in **global memory (GMEM)** of device
- GMEM is where host communicates (ship input data to and get output data from) with device
- GMEM lies in DRAM and is the biggest memory of device
  - Query: `totalGlobalMem` in struct `cudaDeviceProp`
  - E.g., Colab GPU has ? GB GMEM
- But in device, GMEM is the slowest memory to access



Device

SM

SP SP … SP

Registers

SMEM

L1 Cache

Const Cache

…

SM

SP SP … SP

Registers

SMEM

L1 Cache

Const Cache

L2 Cache

DRAM

GMEM

Global memory (GMEM)

- When host calls `cudaMalloc` to allocate a memory region in device, this region will lie in **global memory (GMEM)** of device
- GMEM is where host communicates (ship input data to and get output data from) with device
- GMEM lies in DRAM and is the biggest memory of device
  - Query: `totalGlobalMem` in struct `cudaDeviceProp`
  - E.g., Colab GPU has ? GB GMEM
- But in device, GMEM is the slowest memory to access

→ we should reduce # GMEM accesses from threads

(this is the goal of using other types of memories)

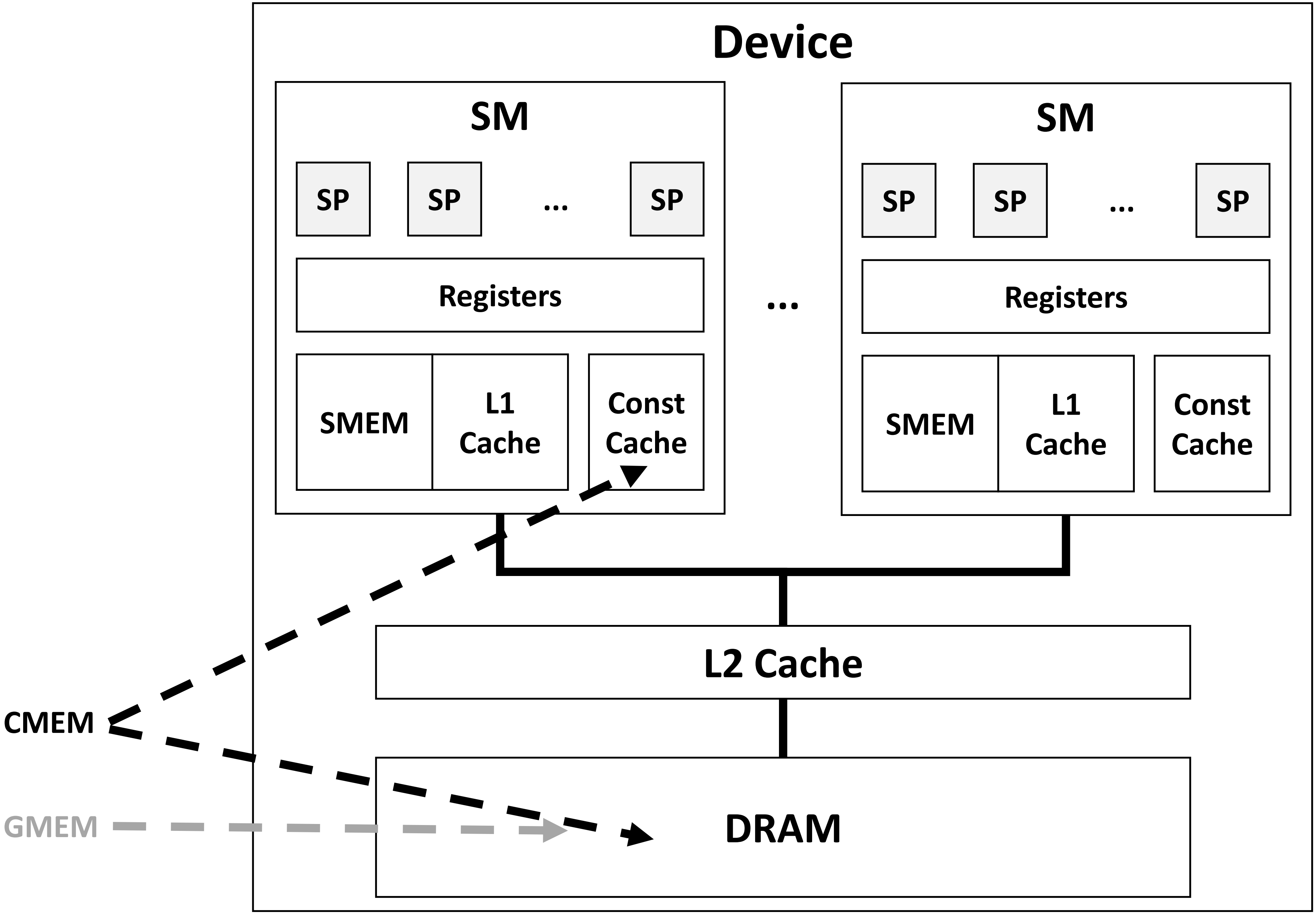
Global memory (GMEM)

- We can allocate a memory region in GMEM by `cudaMalloc`
  - Host can read/write this region by `cudaMemcpy`
  - The pointer pointing to this region is passed as an argument to a kernel by host
  - In the kernel, all threads can access this region through the passed pointer
  - This region will be freed when host calls `cudaFree`
- Or: we can declare a static variable in GMEM with keyword `__device__`
  - E.g., `__device__ float a[10];`
  - This declaration must be put outside all functions tầm vực toàn cục
  - Host can read/write this variable by `cudaMemcpyFrom/ToSymbol`
  - In a kernel, all threads can access this variable (we don't need to pass it as an argument to the kernel)
  - This variable will be freed automatically when the program finishes

Constant memory (CMEM)

- In addition to GMEM, host can also communicate with device through **constant memory (CMEM)**
- When should we use CMEM? host : read + write
  device : only read
  - When host wants to send device data which is constant during kernel execution
  - This data also needs to be small because CMEM is only 64 KB
    - Query: `totalConstMem` in struct `cudaDeviceProp`
  - Threads in the same warp read the same data
    - CMEM lies in DRAM (similar to GMEM), but has **Const Caches** in SMs (8 KB / SM with most CCs[1])

[1] CC: Compute Capability



Device

SM

SP

SP

...

SP

Registers

SMEM

L1 Cache

Const Cache

...

SM

SP

SP

...

SP

Registers

SMEM

L1 Cache

Const Cache

L2 Cache

DRAM

CMEM

GMEM

Constant memory (CMEM)

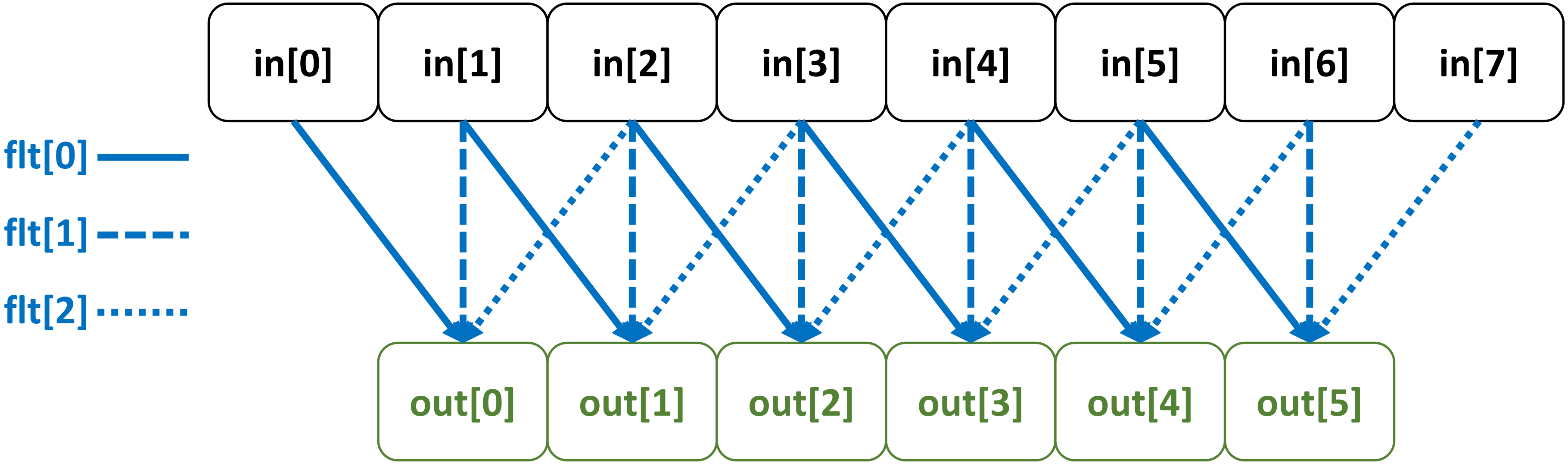
- In addition to GMEM, host can also communicate with device through **constant memory (CMEM)**
- When should we use CMEM?
  - When host wants to send device data which is constant during kernel execution
  - This data also needs to be small because CMEM is only 64 KB
    - Query: `totalConstMem` in struct `cudaDeviceProp`
  - Threads in the same warp read the same data
    - CMEM lies in DRAM (similar to GMEM), but has **Const Caches** in SMs (8 KB / SM with most CCs)
    - Const Cache has low latency, but low bandwidth (4B / clock cycle / SM)

      → if threads in a warp don't read the same memory address, the warp will need to read many times, otherwise it will need to read one time and the read data will be broadcasted to all threads in the warp

Constant memory (CMEM)

- In device, kernel arguments are stored in CMEM
- Declare a variable in CMEM: similar to declaring a static variable in GMEM, but replace keyword `__device__` by `__constant__`
  - E.g., `__constant__ float a[10];`
  - This declaration must be put outside all functions
  - Host can read/write this variable by `cudaMemcpyFrom/ToSymbol`
  - In a kernel, all threads can read (not write) this variable (we don’t need to pass it as an argument to the kernel)
  - This variable will be freed automatically when the program finishes

Example: 1D convolution


flt[0]
flt[1]
flt[2]
in[0]
in[1]
in[2]
in[3]
in[4]
in[5]
in[6]
in[7]
out[0]
out[1]
out[2]
out[3]
out[4]
out[5]

out[0] = in[0] * flt[0] + in[1] * flt[1] + in[2] * flt[2]
out[1] = in[1] * flt[0] + in[2] * flt[1] + in[3] * flt[2]
out[2] = in[2] * flt[0] + in[3] * flt[1] + in[4] * flt[2]
…

ni = 8
nf = 3
no = ? ni - nf + 1

```
#define NF 100
#define NI (1<<24)
#define NO (NI - NF + 1)
__constant__ float d_flt[NF];
…
int main(int argc, char *argv[]) {
        // Allocate memories for input, filter, output; set up data for input, filter
        float *in, *flt, *out;
        …
        // 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(d_flt, flt, NF * sizeof(float), cudaMemcpyHostToDevice);~~
        cudaMemcpyToSymbol(d_flt, 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);
        …
}
```

```
__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;
        }
}
```

số lượng filter

```
#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);
        …
}
```

```
#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;
        }
}
```

Register memory (RMEM)

In addition to CMEM with cache, we can also reduce DRAM accesses by **register memory (RMEM)** biến khai báo trong kernel nằm trên register

```
__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;
		}
}
```

Running time: 17.513

Register memory (RMEM)

In addition to CMEM with cache, we can also reduce DRAM accesses by **register memory (RMEM)**

```
__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;~~
        }
}
```

Running time: ~~17.513~~ 47.107

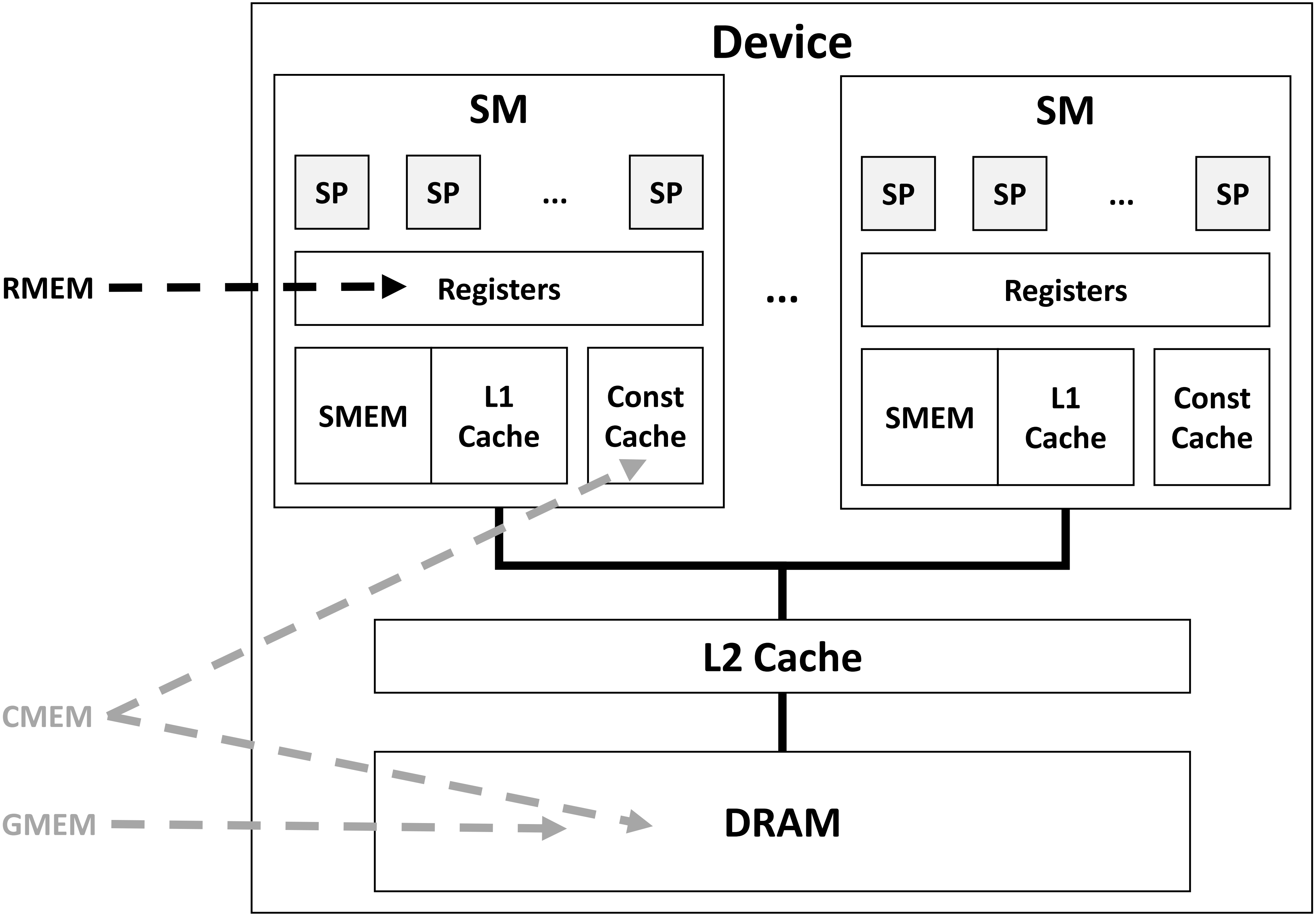
Register memory (RMEM)

In addition to CMEM with cache, we can also reduce DRAM accesses by **register memory (RMEM)**

```
__global__ void convOnDevice(float *d_in, fl
{
        int i = blockIdx.x * blockDim.x + thr

        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;~~
        }
}
```

- Each thread will have its own version of variable s stored in its own RMEM
- RMEM is the fastest memory in device



Device

SM

SP SP … SP

Registers

SMEM

L1 Cache

Const Cache

…

SM

SP SP … SP

Registers

SMEM

L1 Cache

Const Cache

L2 Cache

DRAM

RMEM

CMEM

GMEM

Register memory (RMEM)

In addition to CMEM with cache, we can also reduce DRAM accesses by **register memory (RMEM)**

```
__global__ void convOnDevice(float *d_ir
{
            int i = blockIdx.x * blockDim.x +

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

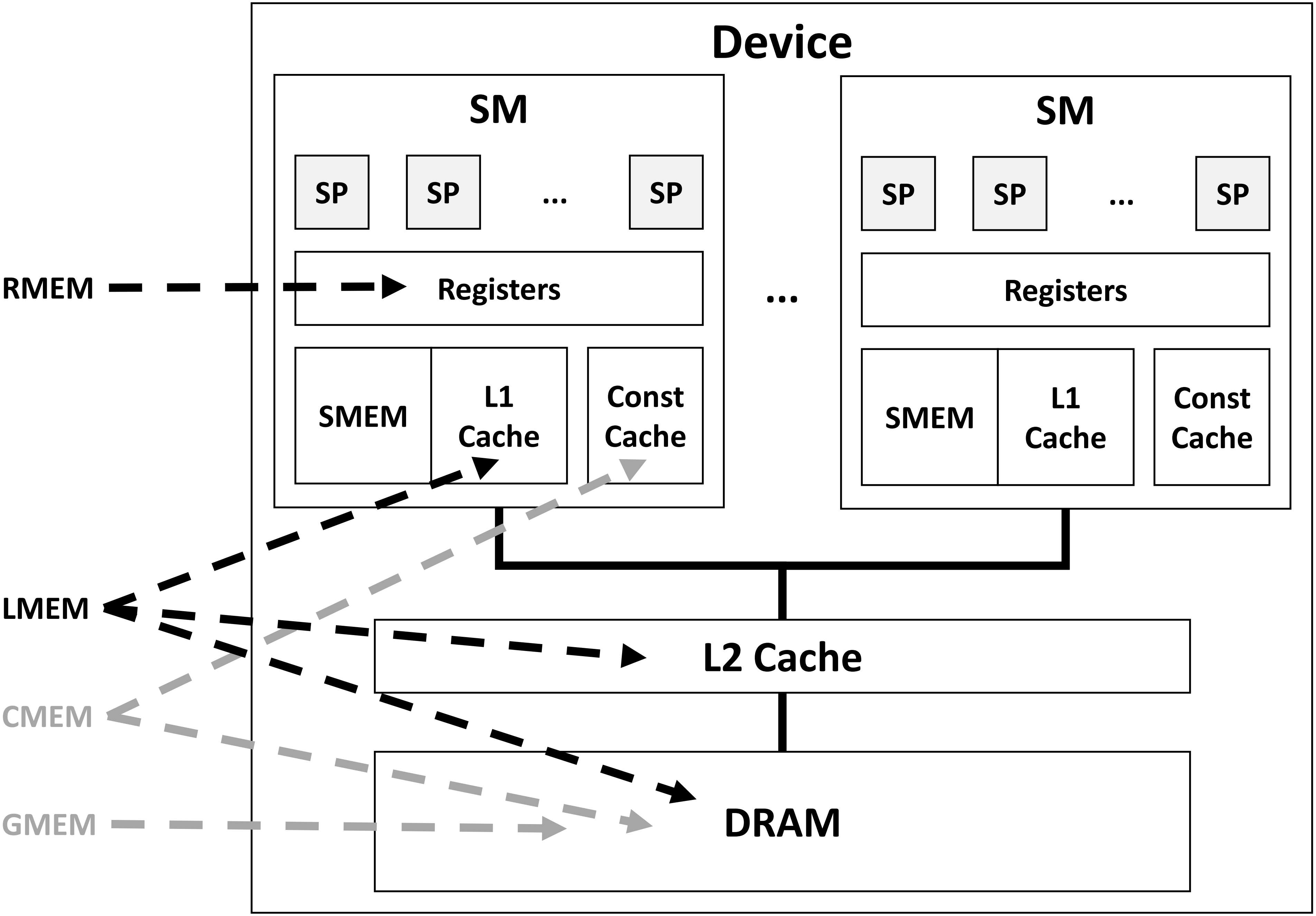
- Each thread will have its own version of variable s stored in its own RMEM
- RMEM is the fastest memory in device
- RMEM of a thread will be freed when it finishes
- Host cannot “see” and read/write RMEM

Write results **many times** to RMEM

Write the final result **one time** from RMEM to GMEM

Local memory (LMEM)

- Although fastest, RMEM size is limited
  - In most CCs: 64K 32-bit registers / SM, at most 255 32-bit registers / thread
- What if each thread has data size > RMEM size?
  - Data “spills” out of RMEM onto **local memory (LMEM)**
  - LMEM lies in DRAM, but has cache


Device
SM
SP
SP
...
SP
Registers
SMEM
L1 Cache
Const Cache
...
SM
SP
SP
...
SP
Registers
SMEM
L1 Cache
Const Cache
RMEM
LMEM
CMEM
GMEM
L2 Cache
DRAM

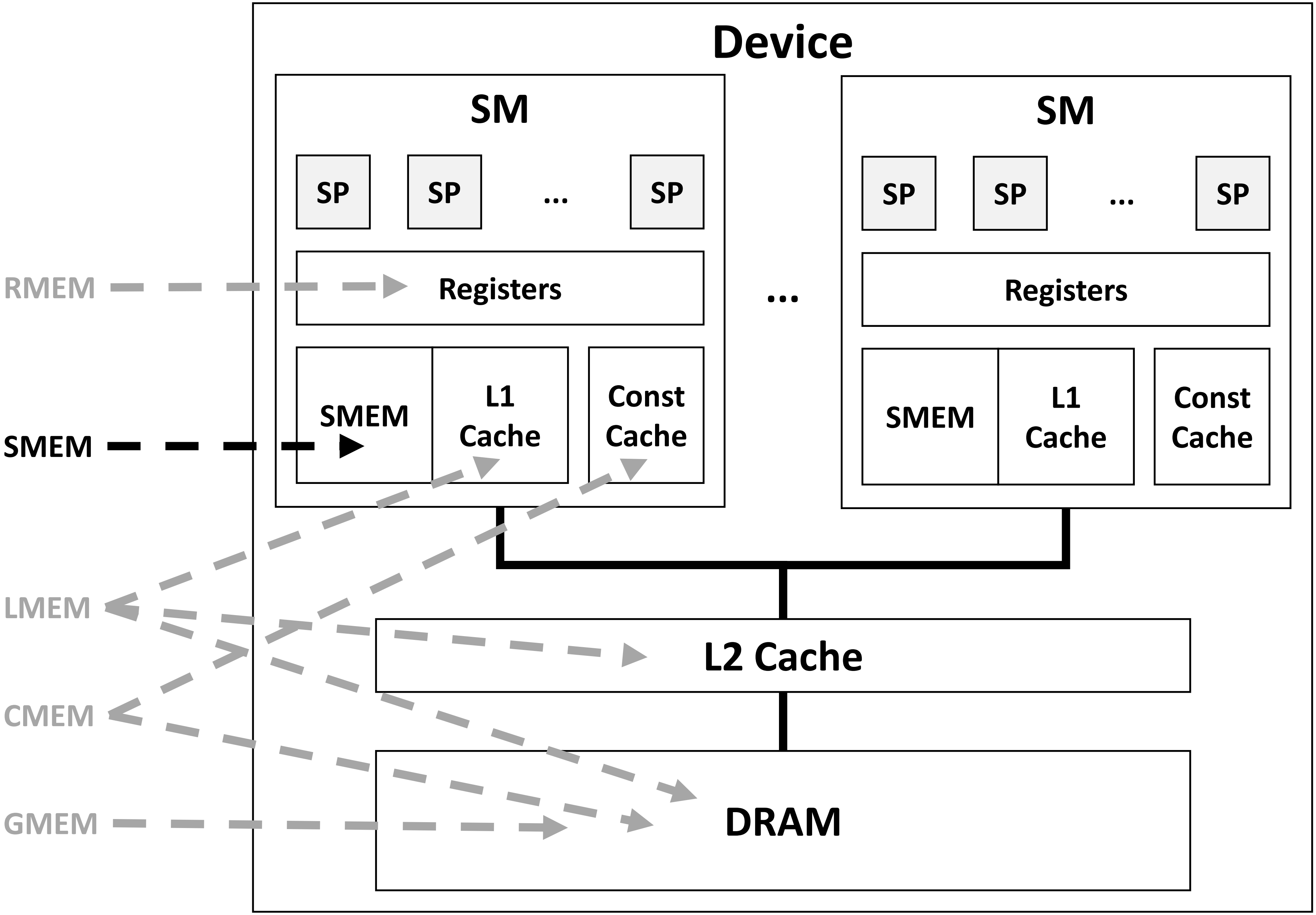
Local memory (LMEM)

- Although fastest, RMEM size is limited
  - In most CC: 64K registers / SM, at most 255 registers / thread
- What if each thread has data size > RMEM size?
  - Data “spills” out of RMEM onto **local memory (LMEM)**
  - LMEM lies in DRAM, but has cache
  - Similar to RMEM, LMEM is private for a thread and will be freed when it finishes

chỉ dành riêng cho 1 thread

Shared memory (SMEM)

- In addition to CMEM and RMEM, we can reduce DRAM accesses by **shared memory (SMEM)**
- A block has its own SMEM and will be freed when the block finishes
- SMEM resides in SM, as the same level with L1 Cache and Const Cache → can be accessed much faster than DRAM (although not as fast as RMEM)


Device
SM
SP
SP
...
SP
Registers
SMEM
L1 Cache
Const Cache
...
SM
SP
SP
...
SP
Registers
SMEM
L1 Cache
Const Cache
RMEM
SMEM
LMEM
CMEM
GMEM
L2 Cache
DRAM

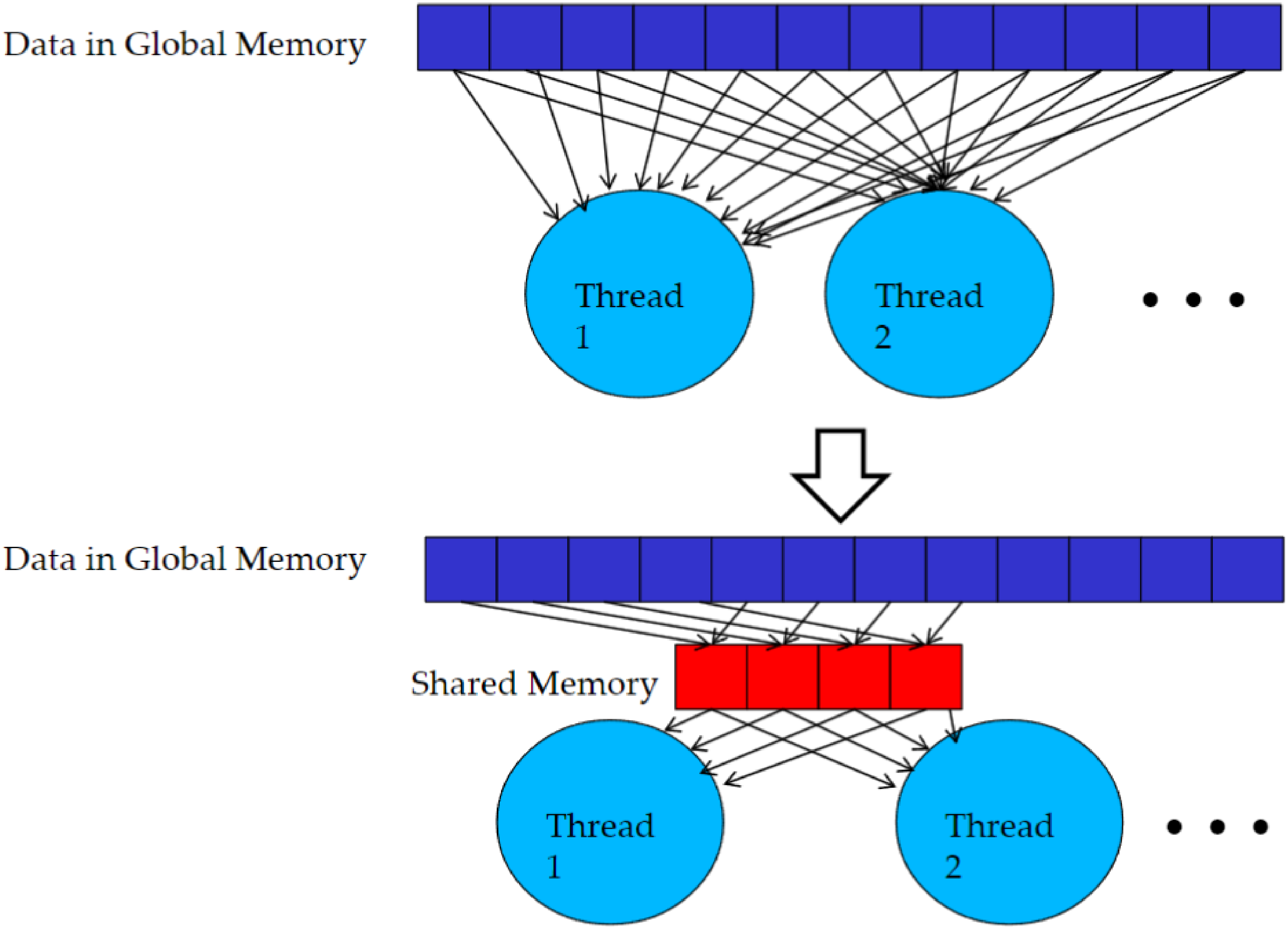
Shared memory (SMEM)

- In addition to CMEM and RMEM, we can reduce DRAM accesses by **shared memory (SMEM)**
- A block has their own SMEM and will be freed when the block finishes
- SMEM resides in SM, as the same level with L1 Cache and Const Cache → can be accessed much faster than DRAM (although not as fast as RMEM)
- In most CCs, each SM has 48-96 KB physical SMEM and this 48-96 KB is divided for blocks residing in SM
- SMEM is the “cache memory” programmers can control
- Host cannot read/write SMEM

Shared memory (SMEM)

- In addition to CMEM and RMEM, we can reduce DRAM accesses by **shared memory (SMEM)**
- A block has their own SMEM and will be freed when the block finishes
- SMEM resides in SM, as the same level with L1 Cache and Const Cache → can be accessed much faster than DRAM (although not as fast as RMEM)
- In most CCs, each SM has 48-96 KB physical SMEM and this 48-96 KB is divided for blocks residing in SM
- SMEM is the “cache memory” programmers can control
- Host cannot read/write SMEM

Shared memory - Work flow


Data in Global Memory
Thread 1
Thread 2
• • •
Data in Global Memory
Shared Memory
Thread 1
Thread 2
• • •

https://wiki.illinois.edu/wiki/display/ECE408

Shared memory - Work flow

- Global memory read/write is **slow**
- To avoid Global Memory bottleneck, tile the input data to take advantage of Shared Memory:
  - Partition data into subsets that fit into the (smaller but faster) shared memory
  - Handle each data subset with one block by:
    - Loading the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
    - Performing the computation on shared memory. Each thread can efficiently access any data element
    - Copying results from shared memory to global memory

Shared memory – Static Allocate

phải là một hằng số cụ thể

```
#define SMEM_SIZE 512
__global__ void kernel_SMEM(int* in, int* out, ...) {
  // SMEM static shared memory declaration
  __shared__ int s_Data[SMEM_SIZE]; //Known at compile time.

  // Copy Data from gmem to smem using multiple thread
  // to exploit memory-level parallelism
  //...

  // Do computation on smem (s_Data) instead of gmem(in)
  //...

  // Write the result back to gmem
  //...
}
```

Shared memory – Static Allocate

```
#define SMEM_SIZE 512
__global__ void kernel_SMEM(int* in, int* out, ...) {
  // SMEM static shared memory declaration
  __shared__ int s_Data[SMEM_SIZE]; //With constant number.

  i = blockDim.x * blockIdx.x + threadIdx.x;
  // Copy Data from gmem to smem
  s_Data[threadIdx.x] = in[i];
  //...

  // Do computation on s_Data instead of in
  //...

  // Write back result from s_Data to out if need
  out[i] = s_Data[threadIdx.x];
  //...
}
```

Shared memory - Dynamic allocation

```
__global__ void kernel_SMEM(int* in, int* out, ...) {
  // SMEM static shared memory declaration
  extern __shared__ int s_Data[];

 // Copy Data from gmem to smem using multiple thread
  // to exploit memory-level parallelism
  //...

  // Do computation on smem (s_Data) instead of gmem(in)
  //...

  // Write the result back to gmem
  //...
}
```

- Specifying the desired size in bytes as a third argument inside the triple angled brackets

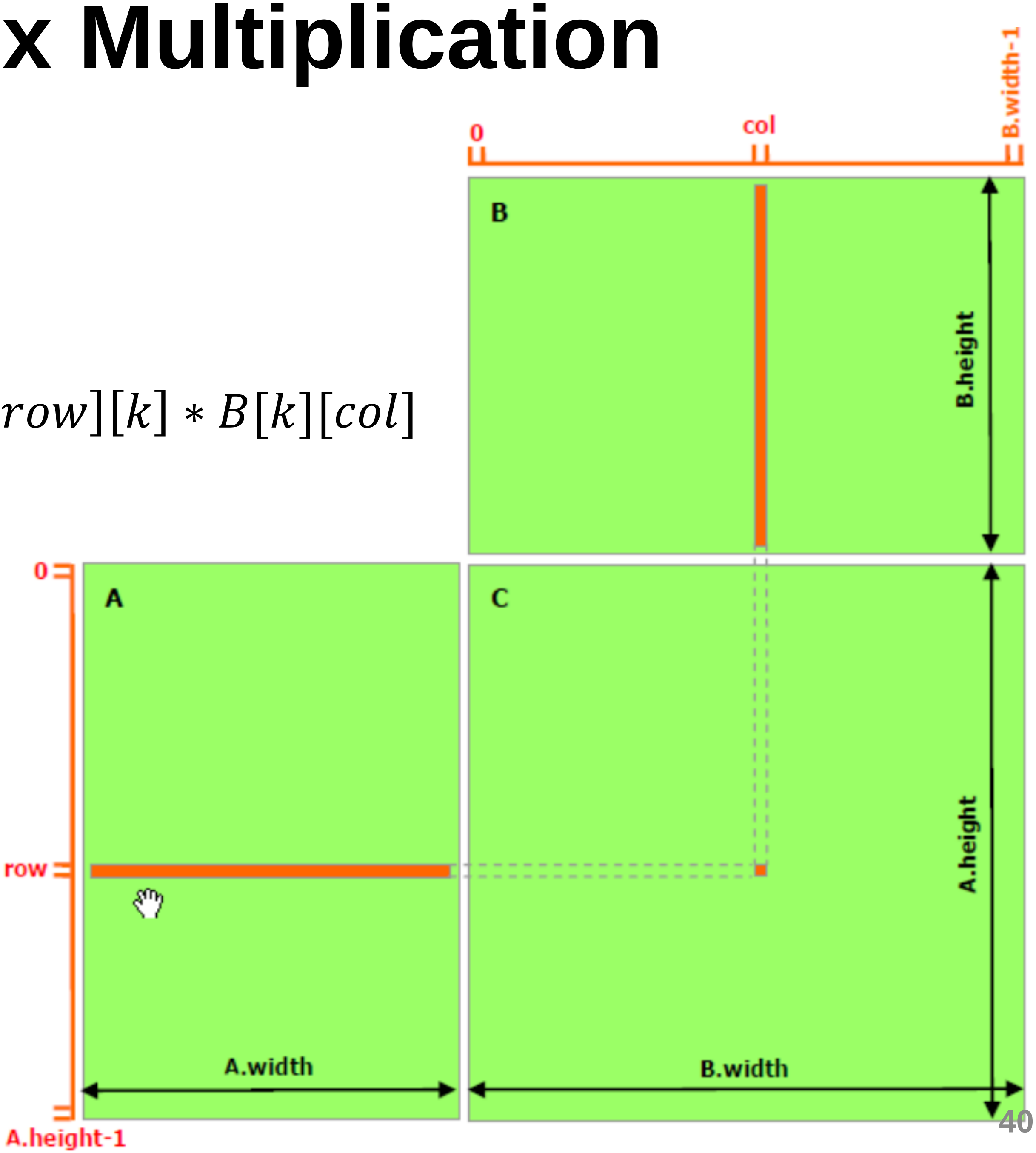
```
smemSize = n * sizeof(int)
kernel_SMEM<<<gridSize,blockSize, smemSize>>>(...)
```

Kích thước shared memory

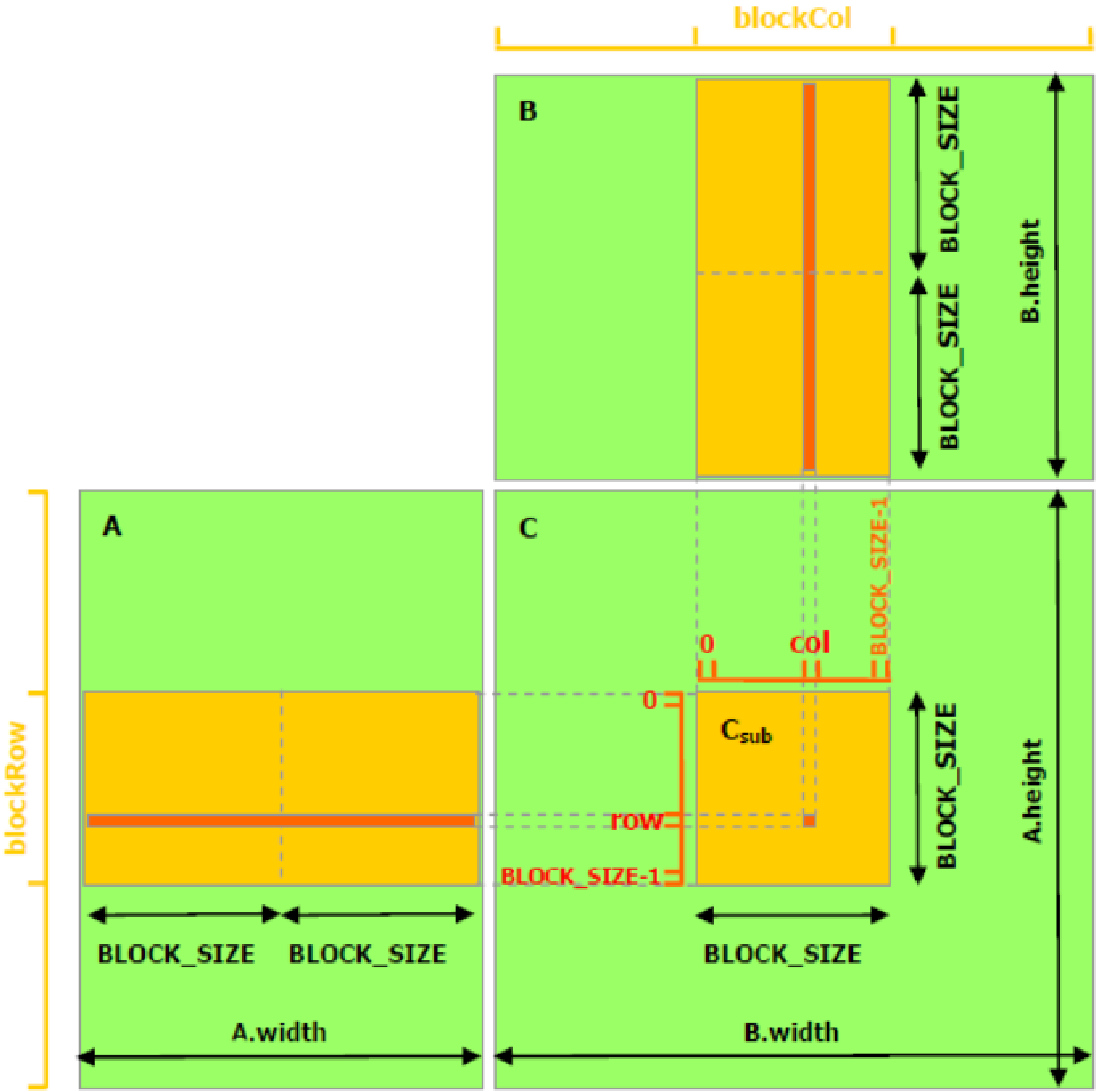
Basic Matrix Multiplication

Your homework

$$C[row][col] = \sum_{k=1}^{A.width} A[row][k] * B[k][col]$$


0
col
B.width-1
B
B.height
0
A
C
row
A.height
A.width
B.width
A.height-1

Tiled Matrix Multiplication


blockCol
B
BLOCK_SIZE
BLOCK_SIZE
B.height
A
C
blockRow
BLOCK_SIZE-1
0
col
0
row
BLOCK_SIZE-1
Csub
BLOCK_SIZE
A.height
BLOCK_SIZE
BLOCK_SIZE
BLOCK_SIZE
A.width
B.width

Tiled Matrix Multiplication

- Data of A and B are in GMEM, and every time the data of A/B is read, the threads must read from GMEM.
- An element in the A/B matrix is read by different threads in the same block.
- We can reduce the number of GMEM accesses by:
  - First, each block will read the A/B data that the block needs from GMEM and store it in the block's SMEM. Each element is read once
  - Then, when A/B data is needed, threads in the block can read the data in SMEM at high speed.

Tiled Matrix Multiplication

- **Problem**: A and B too large to store in SMEM.
- **Solution**:
  - Partition the Data that block need into subset. E.g.: A1,A2,… B1,B2,…
  - Each phase:
    - Block will load $\boldsymbol{A_i}$, $\boldsymbol{B_i}$ from GMEM to SMEM.
    - Calculate partial Matrix Multiplication, add to previous paritial result.
    - Repeat for all Subset of A and B.

Tiled Matrix Multiplication

- Your homework.

Summary

Utilize high speed memories to reduce DRAM accesses

Price: it can decrease occupancy (e.g., if SM has 48 KB SMEM and block consumes 40 KB SMEM then SM can only contain one block)

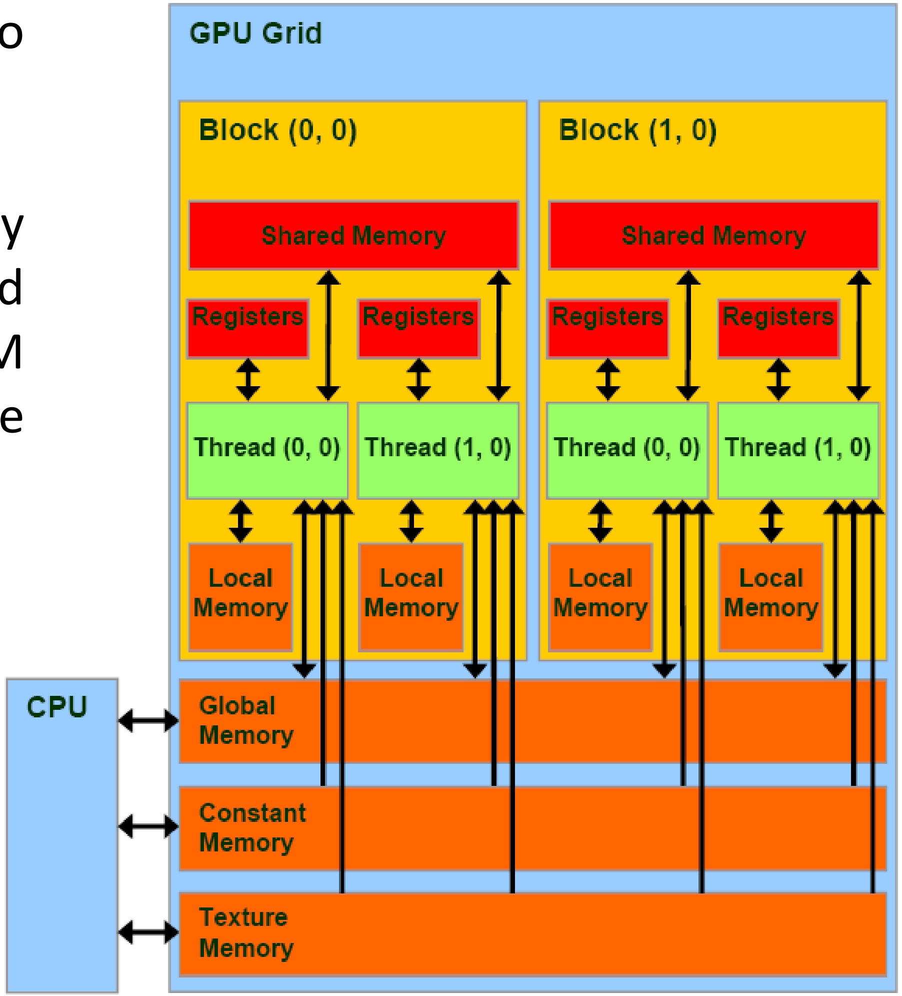

GPU Grid
Block (0, 0)
Block (1, 0)
Shared Memory
Shared Memory
Registers
Registers
Registers
Registers
Thread (0, 0)
Thread (1, 0)
Thread (0, 0)
Thread (1, 0)
Local Memory
Local Memory
Local Memory
Local Memory
CPU
Global Memory
Constant Memory
Texture Memory

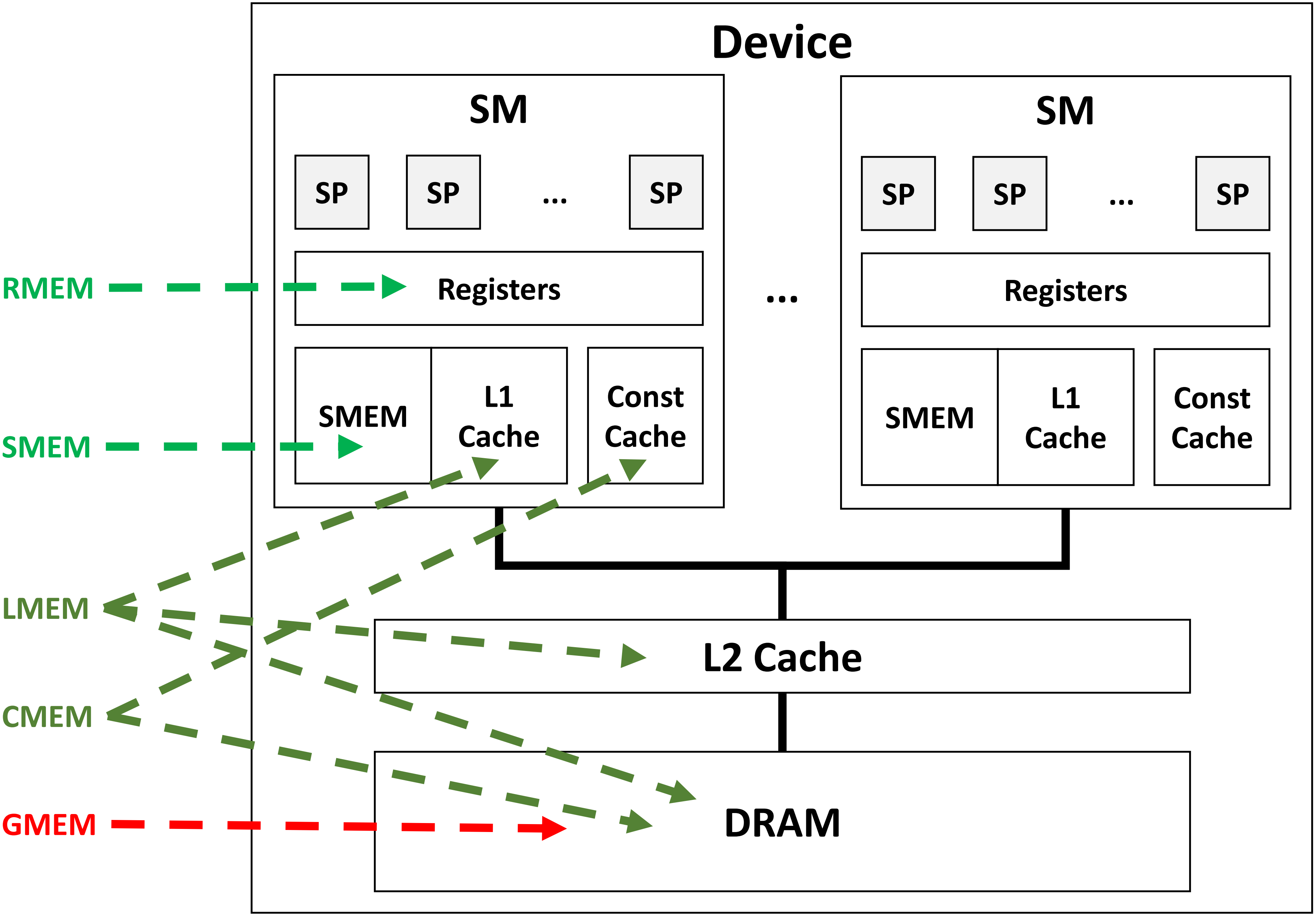
Image source: http://www.realworldtech.com/includes/images/articles/

Declaring CUDA Variables

thời gian tồn tại của vùng nhớ

Variable declaration	Memory	Scope	Lifetime
`int LocalVar;`	register	thread	thread
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;` (phải nằm toàn cục)	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

- **`__device__`**
  - Optional with **`__shared__`**, or **`__constant__`**
  - Not allowed by itself within functions
- Automatic variables with no qualifiers
  - In a register for primitive types and structures
  - In global memory for arrays with no fix size.



Device

SM

SP SP … SP

Registers

SMEM

L1 Cache

Const Cache

…

SM

SP SP … SP

Registers

SMEM

L1 Cache

Const Cache

L2 Cache

DRAM

RMEM

SMEM

LMEM

CMEM

GMEM

Reference

- [1] Wen-Mei, W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022
- [2] Cheng John, Max Grossman, and Ty McKercher. *Professional Cuda C Programming*. John Wiley & Sons, 2014
- [3] Illinois GPU course

https://wiki.illinois.edu/wiki/display/ECE408/ECE408+Home

THE END