

Các loại bộ nhớ trong CUDA (p2)

Trần Trung Kiên – Phạm Trọng Nghĩa

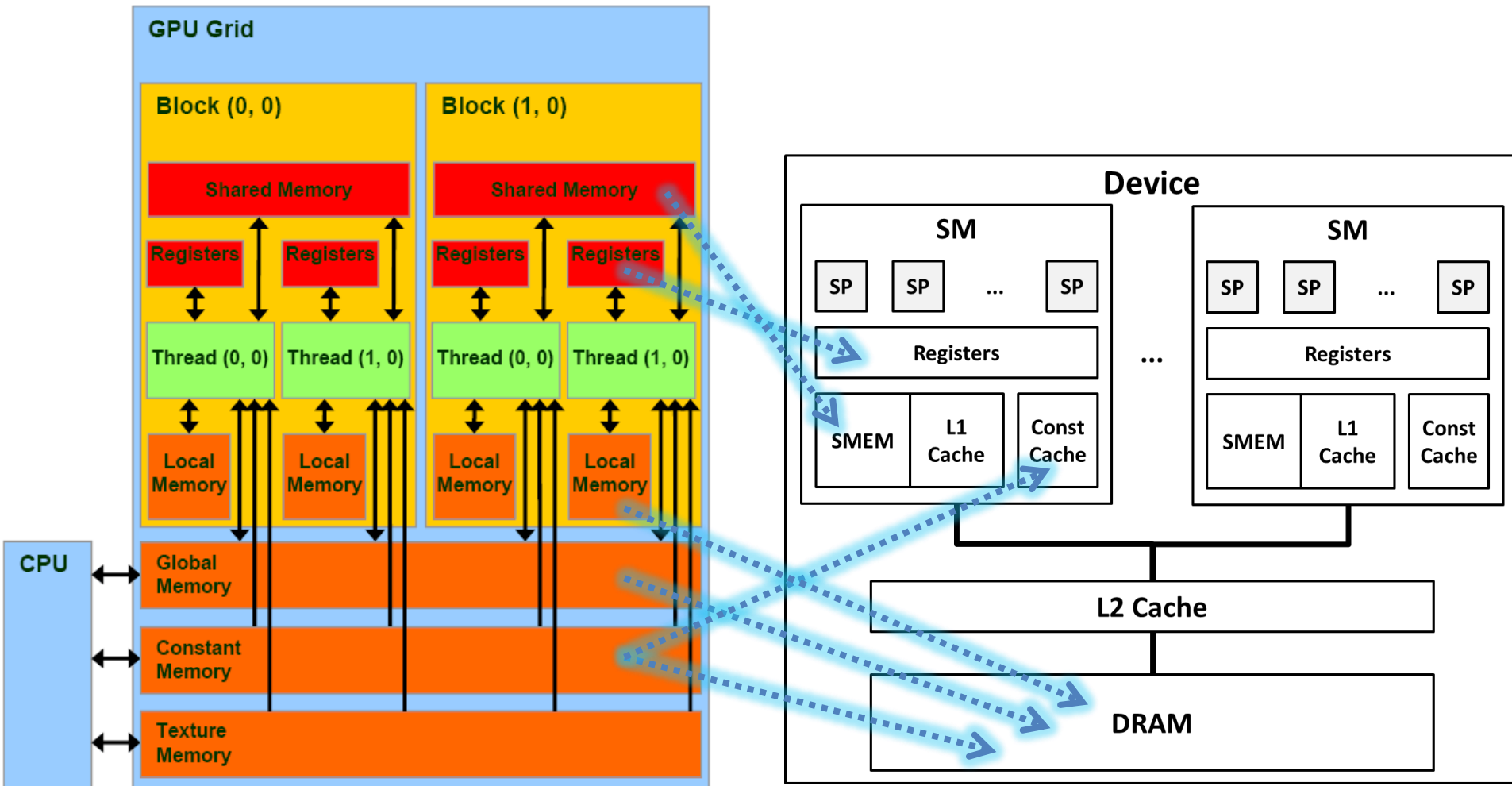
ttkien@fit.hcmus.edu.vn

ptnghia@fit.hcmus.edu.vn



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

Ôn lại buổi trước



- Tận dụng các bộ nhớ có tốc độ cao ở ngay trong các SM để lưu dữ liệu, 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)

Buổi này

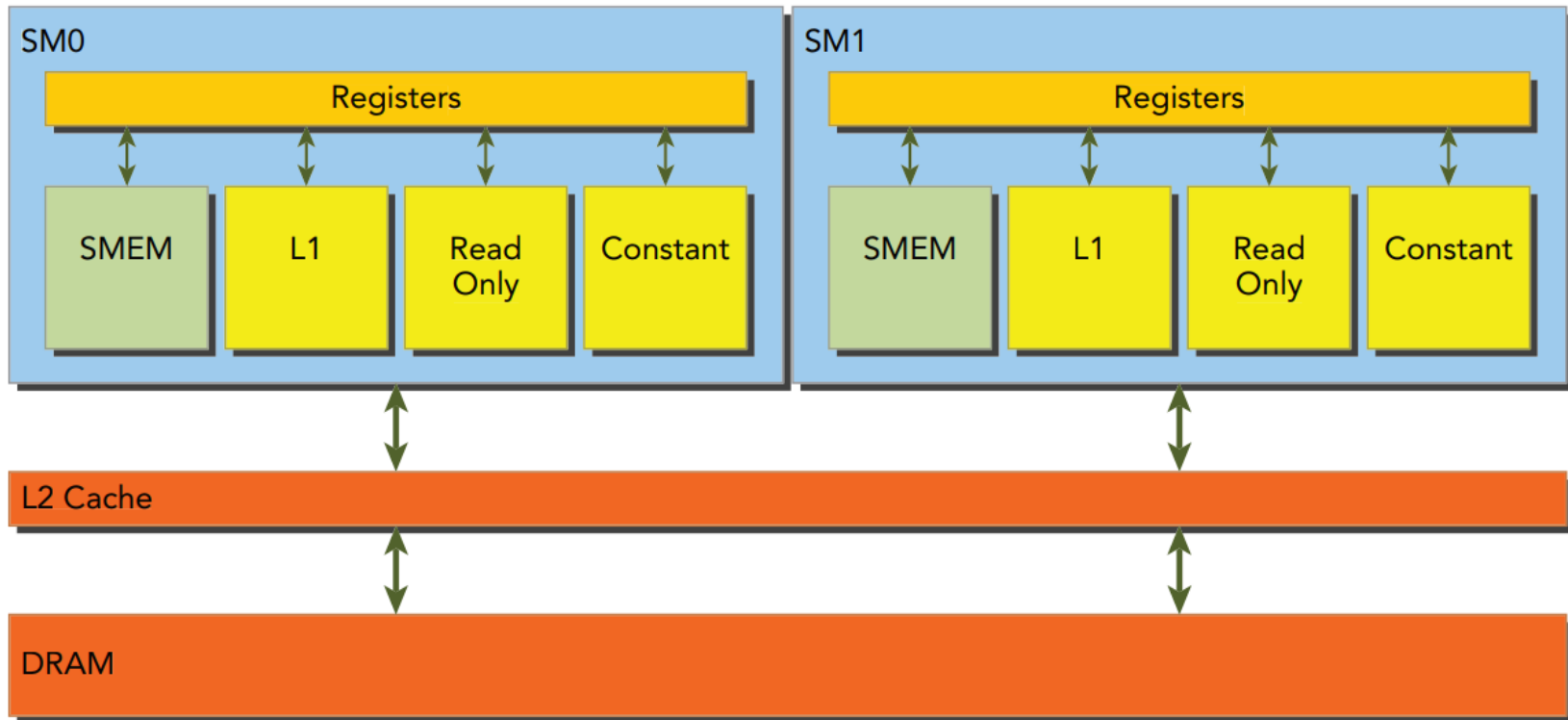
- ☐ Truy xuất GMEM hiệu quả
- ☐ Truy xuất SMEM hiệu quả

Truy xuất GMEM

Các truy xuất được xử lý theo warp

- ☐ Mỗi warp cung cấp 32 địa chỉ (ít hơn nếu có các thread không kích hoạt) và số byte cần truy xuất từ mỗi địa chỉ
- ☐ Phần cứng nhìn vào thông tin này và chuyển thành một hoặc một vài **truy xuất khối dữ liệu 32 byte** (32 byte kề nhau và phải được căn lề)

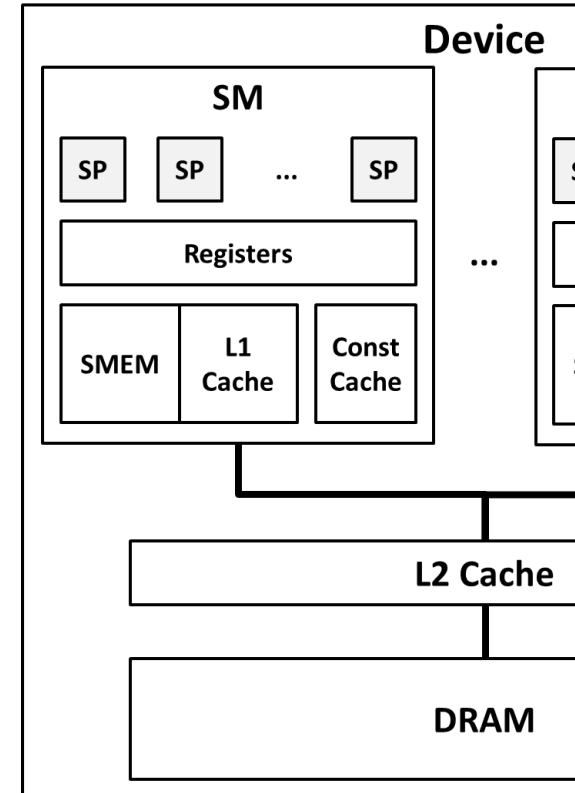
Đọc GMEM



- ☐ Các truy xuất bộ nhớ toàn cục được thực hiện qua cache.
- ☐ Dùng 128-byte (L1), hoặc 32-byte (không dùng L1)

Đọc GMEM

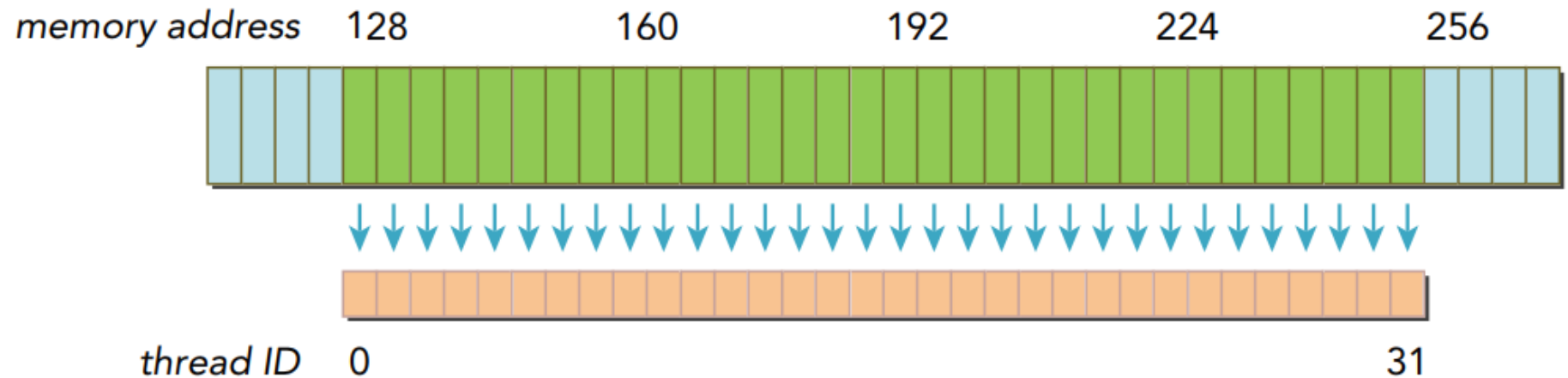
- Đọc được cache ở L1
 - ▣ Khi biên dịch, truyền:
-Xptxas -d1cm=**ca**
 - ▣ Nếu không có dữ liệu ở L1 thì xuống L2, nếu không có ở L2 thì xuống DRAM
- Đọc **không** được cache ở L1
 - ▣ Khi biên dịch, truyền:
-Xptxas -d1cm=**cg**
 - ▣ Xuống thẳng L2, nếu không có dữ liệu ở L2 thì xuống DRAM



Đọc GMEM

- Truy xuất bộ nhớ toàn cục cần lưu ý hai vấn đề:
 - Aligned memory accesses:
 - Địa chỉ đầu tiên của truy xuất bộ nhớ chia hết cho 32 hoặc 128 (L1)
 - Coalesced memory accesses
 - 32 thread trong cùng một warp truy xuất đến một vùng nhớ liên tục.

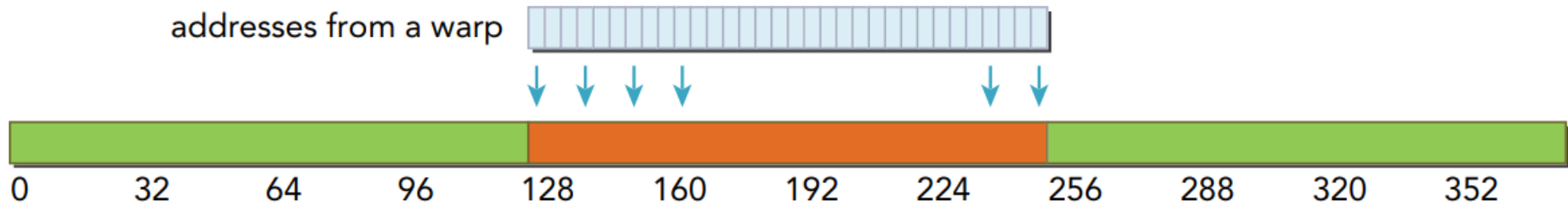
Aligned and Coalesced Access



- ☐ Aligned và Coalesced
- ☐ Tối ưu
- ☐ Một lần đọc 128-byte được thực hiện.
Không có lãng phí bộ nhớ

Aligned and Coalesced Access

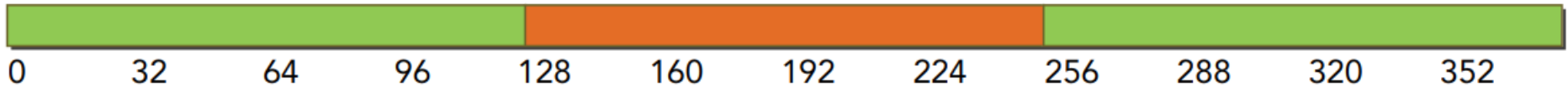
addresses from a warp



- ☐ Aligned và coalesced
- ☐ Tối ưu

Aligned and Coalesced Access

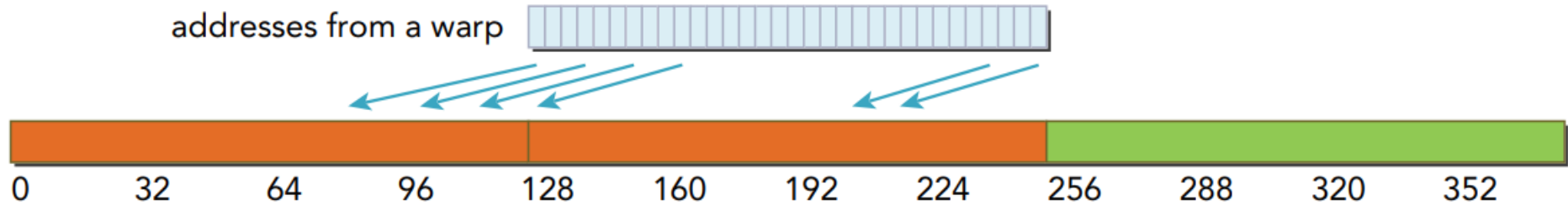
addresses from a warp



- ☐ Aligned?
- ☐ Số lượng 128-byte transaction?
- ☐ Bus utilization (%)?

Aligned and Coalesced Access

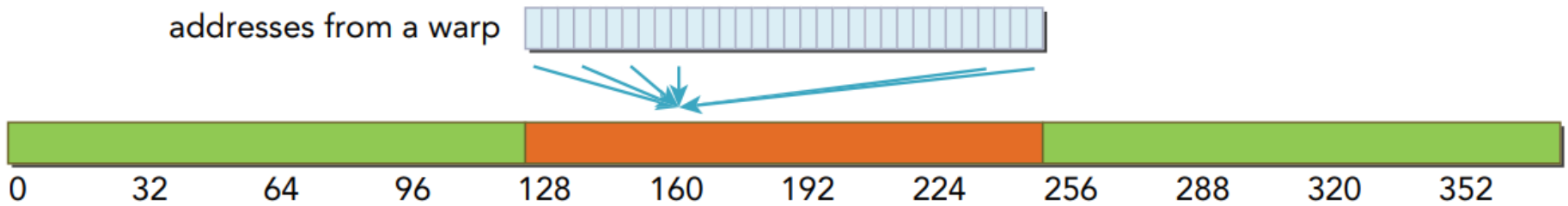
addresses from a warp



- ☐ Aligned?
- ☐ Số lượng 128-byte transaction?
- ☐ Bus utilization (%)?

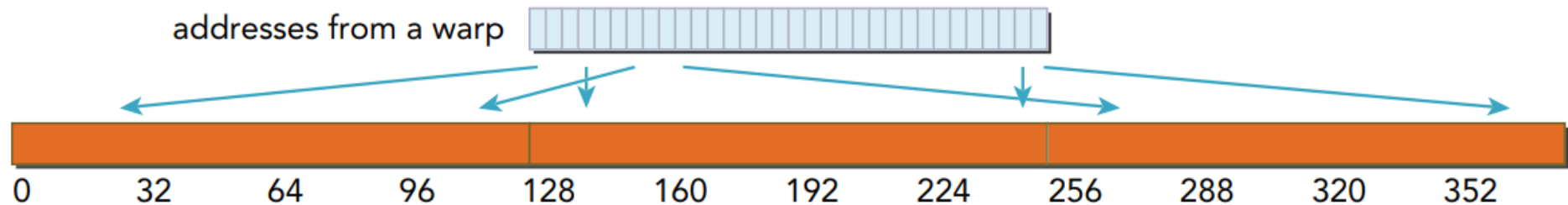
Aligned and Coalesced Access

addresses from a warp



- ☐ Aligned?
- ☐ Số lượng 128-byte transaction?
- ☐ Bus utilization (%)?

Aligned and Coalesced Access



- ☐ Aligned?
- ☐ Số lượng 128-byte transaction?
- ☐ Bus utilization (%)?

Uncached Loads

- ☐ Không dùng Cache L1.
- ☐ Memory segment: 32 bytes

Đọc GMEM

- Warp requests 32 aligned, consecutive 4-byte words
- Addresses fall within 4 segments
 - Warp needs 128 bytes
 - 128 bytes move across the bus on a miss
 - Bus utilization: 100%



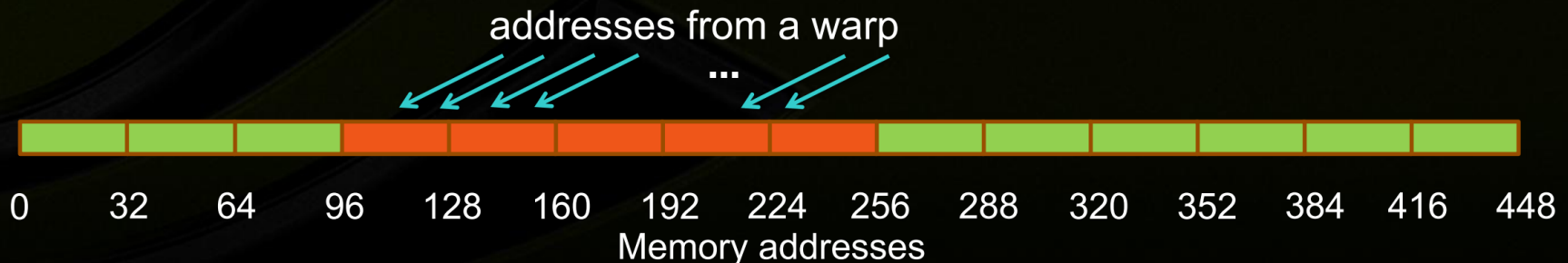
Đọc GMEM

- Warp requests 32 aligned, permuted 4-byte words
- Addresses fall within 4 segments
 - Warp needs 128 bytes
 - 128 bytes move across the bus on a miss
 - Bus utilization: 100%



Đọc GMEM

- Warp requests 32 misaligned, consecutive 4-byte words
- Addresses fall within at most 5 segments
 - Warp needs 128 bytes
 - 160 bytes move across the bus on misses
 - Bus utilization: at least 80%
 - Some misaligned patterns will fall within 4 segments, so 100% utilization



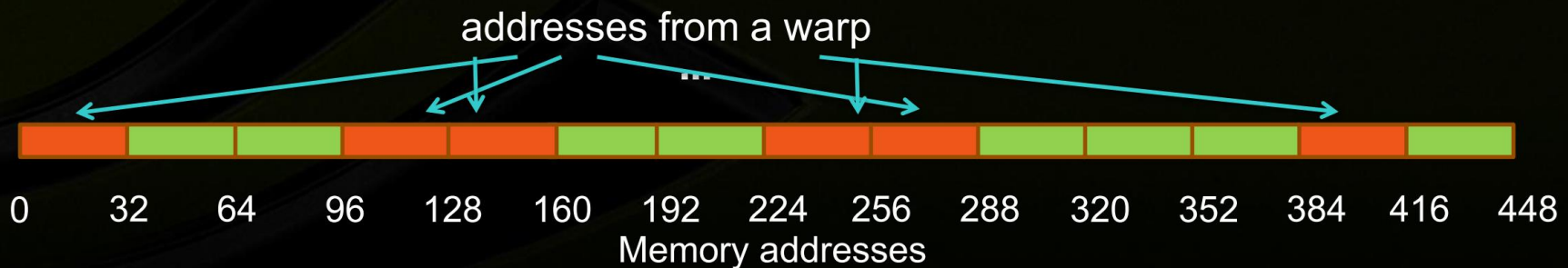
Đọc GMEM

- All threads in a warp request the same 4-byte word
- Addresses fall within a single segment
 - Warp needs 4 bytes
 - 32 bytes move across the bus on a miss
 - Bus utilization: 12.5%



Đọc GMEM

- Warp requests 32 scattered 4-byte words
- Addresses fall within N segments
 - Warp needs 128 bytes
 - $N*32$ bytes move across the bus on a miss
 - Bus utilization: $128 / (N*32)$

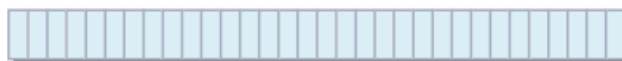


Ghi GMEM

- ☐ Giống như đọc nhưng không được cache ở L1 (thông tin trước khi có CC 7.x)
- ☐ Memory segment: 32 bytes.
- ☐ Một memory transaction: 1, 2 hoặc 4 segment
- ☐ Nếu trong warp có nhiều thread cùng ghi vào một địa chỉ thì
 - ☐ chỉ có một thread sẽ ghi được
 - ☐ nhưng đó là thread nào thì không biết được

Ghi GMEM

addresses from a warp



one 4-segment transaction

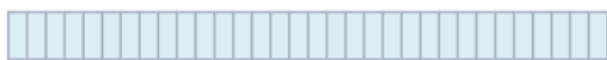


0 32 64 96 128 160 192 224 256 288 320 352

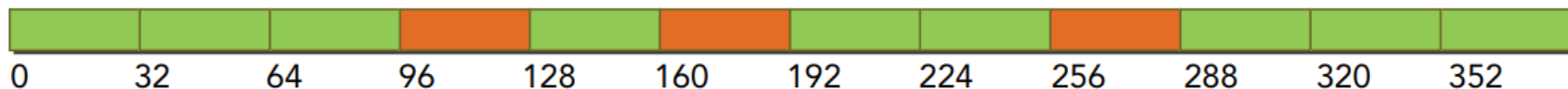
memory address

Ghi GMEM

addresses from a warp

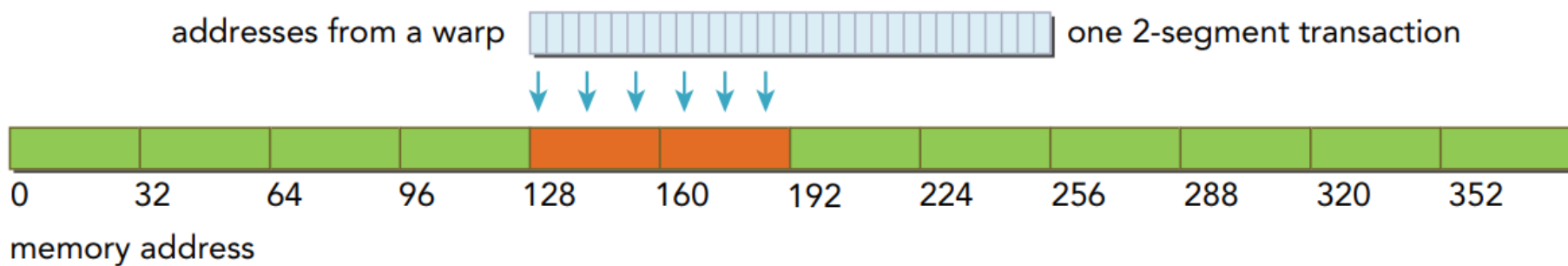


three 1-segment transactions

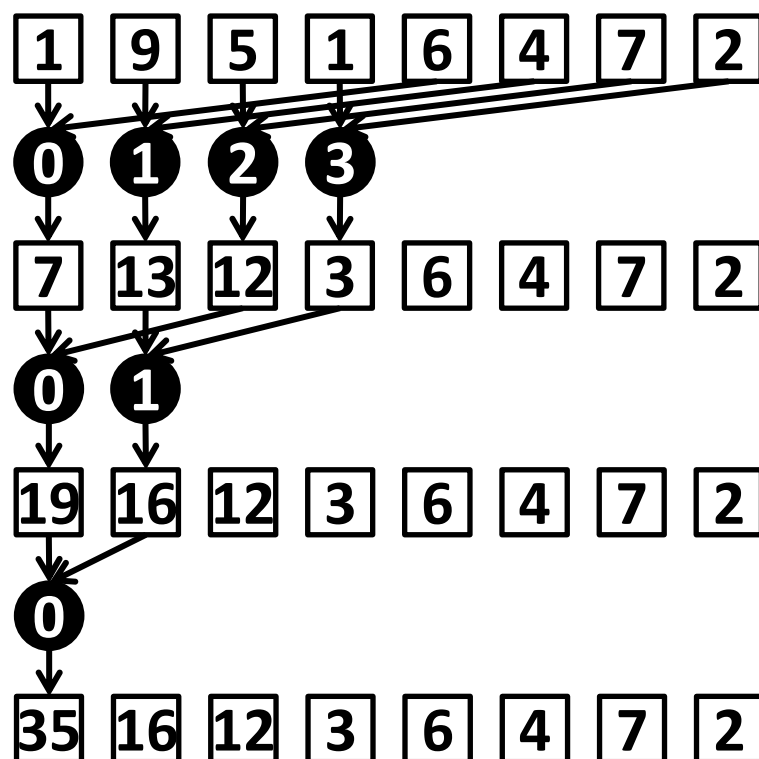
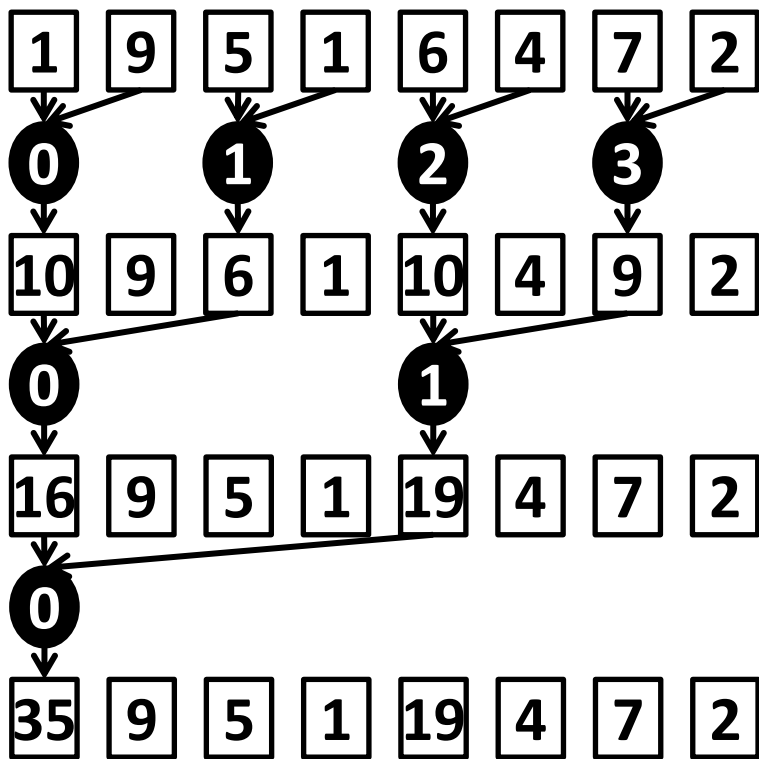


memory address

Ghi GMEM



Ví dụ 1: tại sao hàm kernel 3 lại nhanh hơn kernel 2?

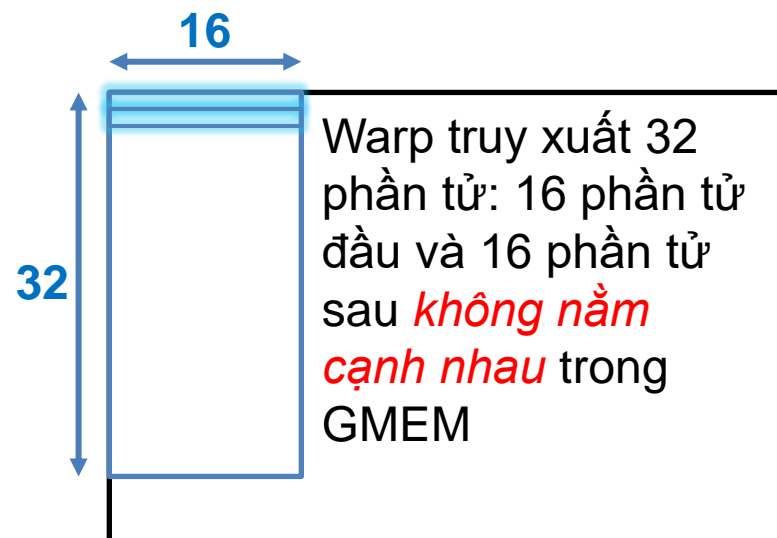
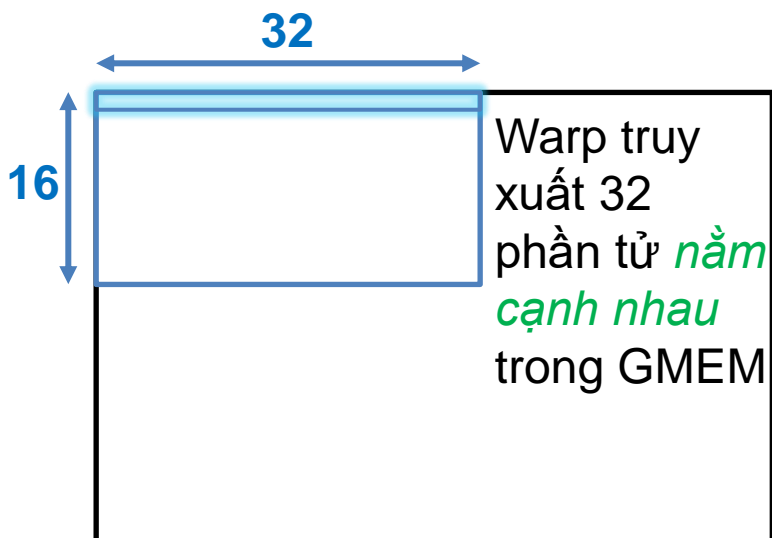


Ví dụ 2: kích thước block trong bài toán cộng 2 ma trận

| Function | Block size | Grid size | Time (ms) |
|------------------|------------|-----------|--------------|
| addMatOnHost | | | 257.141 |
| addMatOnDevice2D | 32 x 32 | 257 x 257 | 14.676 |
| | 16 x 32 | 513 x 257 | 12.267 |
| | 32 x 16 | 257 x 513 | 9.911 |
| | 16 x 16 | 513 x 513 | 10.904 |

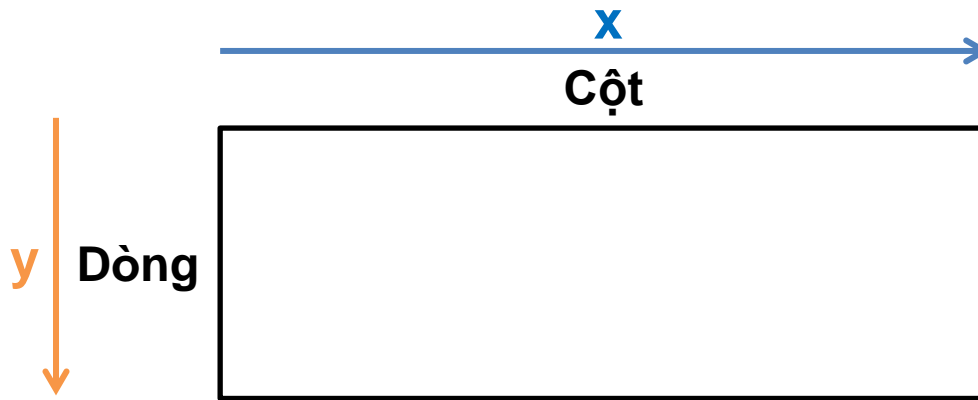
Block 16x32 vs 32x16:

- Có cùng occupancy?
- Tại sao 32x16 lại nhanh hơn 16x32?

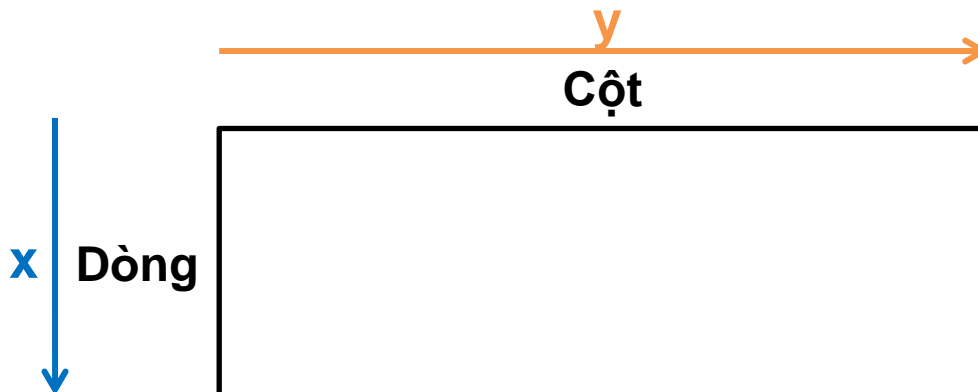


Ví dụ 3: chiều x và y ở dữ liệu 2 chiều

```
int row = blockIdx.y * blockDim.y + threadIdx.y;  
int col = blockIdx.x * blockDim.x + threadIdx.x;
```



Đảo chiều x với chiều y (đảo công thức tính row và col) thì có bị gì không?



Truy xuất kiểu dữ liệu struct

- ☐ Truy xuất GMEM hiệu quả nhất khi: các thread trong warp truy xuất đến các phần tử nằm kề nhau trong GMEM và địa chỉ của phần tử đầu tiên được căn lề
- ☐ Điều này chỉ đúng khi phần tử có kích thước thuộc loại cơ bản: 1, 2, 4, 8, 16 byte (các kiểu dữ liệu được định nghĩa sẵn đều có kích thước thuộc loại này)
- ☐ Nếu phần tử có kích thước khác (vd, tự định nghĩa một struct) thì trình biên dịch sẽ chuyển câu lệnh truy xuất phần tử thành các câu lệnh truy xuất theo các kích thước cơ bản (phải được căn lề)

Hệ quả là ...

Truy xuất kiểu dữ liệu struct

```
struct Point
{
    float x;
    float y;
    float z;
};

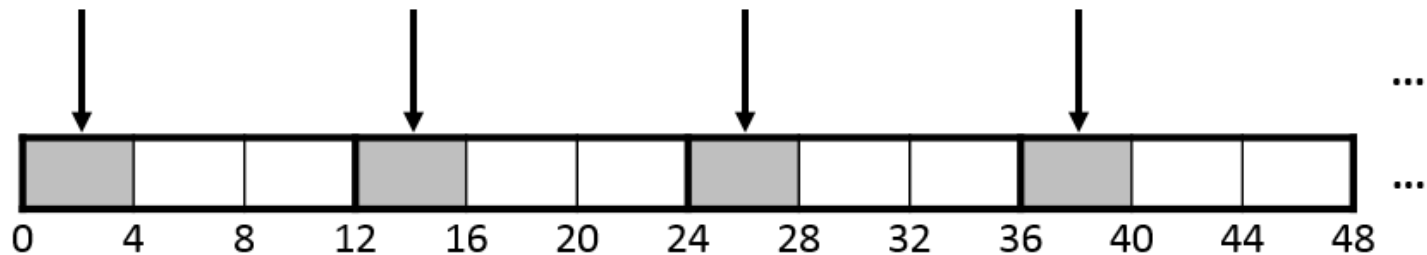
...
Point *d_data;
cudaMalloc(&d_data, ...);

...
__global__ void kernel(Point *d_data, ...)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    Point p = d_data[i];
    ...
}
```

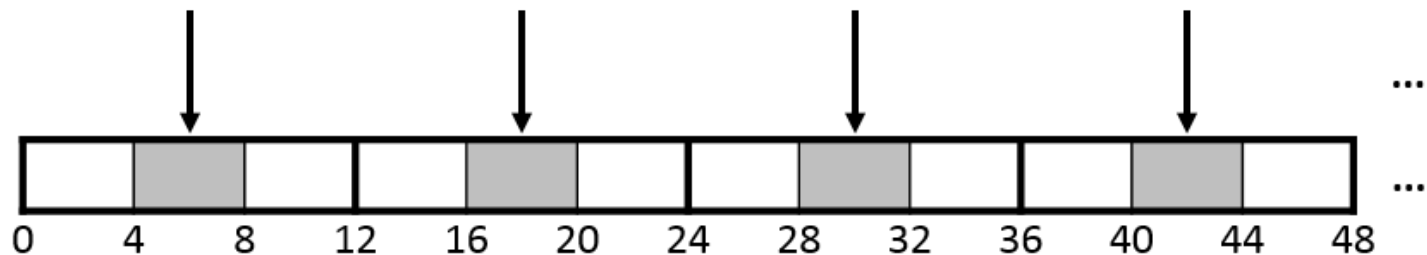
Truy xuất phần tử **12** byte → sẽ bị chuyển thành 3 câu lệnh truy xuất phần tử 4 byte

Chuyển thành 2 câu lệnh: một truy xuất 8 byte, một truy xuất 4 byte?

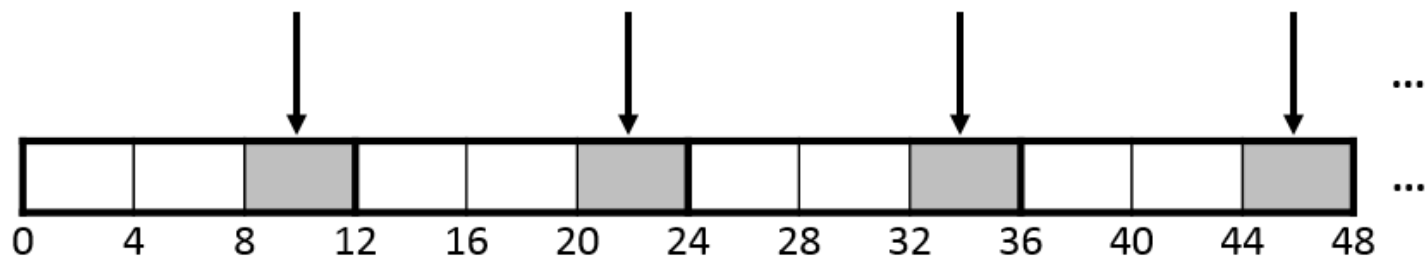
Các phần tử được đọc bởi các tiểu trình trong warp với câu lệnh đọc 4 byte thứ nhất



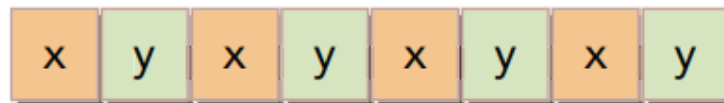
Các phần tử được đọc bởi các tiểu trình trong warp với câu lệnh đọc 4 byte thứ hai



Các phần tử được đọc bởi các tiểu trình trong warp với câu lệnh đọc 4 byte thứ ba



AoS memory layout



thread ID

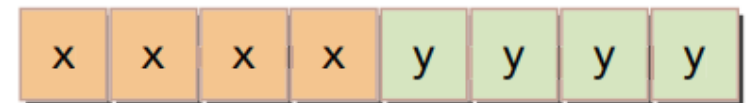
t0

t1

t2

t3

SoA memory layout



t0

t1

t2

t3

```
struct SoA
```

```
{
```

```
    float *xArr;
```

```
    float *yArr;
```

```
    float *zArr;
```

```
};
```

```
...
```

```
SoA d_data;
```

```
cudaMalloc(&d_data.xArr, ...);
```

```
cudaMalloc(&d_data.yArr, ...);
```

```
cudaMalloc(&d_data.zArr, ...);
```

```
...
```

```
__global__ void kernel(SoA d_data, ...)
```

```
{
```

```
    int i = blockIdx.x * blockDim.x + threadIdx.x;
```

```
    float x = d_data.xArr[i];
```

```
    float y = d_data.yArr[i];
```

```
    float z = d_data.zArr[i];
```

```
...}
```

Truy xuất GMEM (ở DRAM) hiệu quả

- Tận dụng tối đa các byte dữ liệu được vận chuyển giữa DRAM và SM bằng cách cố gắng để **các thread trong warp** truy xuất đến **các phần tử kề nhau trong bộ nhớ** và **địa chỉ của phần tử đầu tiên được căn lề**
 - Lưu ý: phần tử được truy xuất phải có kích thước thuộc loại cơ bản (1, 2, 4, 8, 16 byte)
- Ngoài ra, SM cần có đủ các câu-lệnh-truy-xuất-GMEM độc lập để che độ trễ
 - Tăng số lượng warp độc lập trong SM
 - Tăng số lượng câu-lệnh-truy-xuất-GMEM độc lập trong mỗi warp

Buổi này

- ☐ Truy xuất GMEM hiệu quả
- ☐ Truy xuất SMEM hiệu quả

Ví dụ: chuyển vị ma trận

| | | | |
|---|---|----|----|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |

matrix

| | | |
|---|---|----|
| 0 | 4 | 8 |
| 1 | 5 | 9 |
| 2 | 6 | 10 |
| 3 | 7 | 11 |

transposed

Ví dụ: chuyển vị ma trận

- ☐ Cho ma trận `iMatrix`. Tính ma trận chuyển vị `oMatrix`: `oMatrix[r][c] = iMatrix[c][r]`
- ☐ Đơn giản hóa:
 - ☐ Ma trận vuông có kích thước $w \times w$
 - ☐ Block vuông có kích thước 32×32 , và w chia hết cho 32

Host code

```
void transpose(float *out, float *in, const int nx, const
int ny)
{
    for (int iy = 0; iy < ny; ++iy) {
        for (int ix = 0; ix < nx; ++ix) {
            out[ix*ny+iy] = in[?];
        }
    }
}
```

Host code

```
void transpose(float *out, float *in, const int nx, const
int ny)
{
    for (int iy = 0; iy < ny; ++iy) {
        for (int ix = 0; ix < nx; ++ix) {
            out[ix*ny+iy] = in[iy*nx+ix];
        }
    }
}
```

data layout of original matrix

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|----|----|

data layout of transposed matrix

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|----|---|---|----|
| 0 | 4 | 8 | 1 | 5 | 9 | 2 | 6 | 10 | 3 | 7 | 11 |
|---|---|---|---|---|---|---|---|----|---|---|----|

Hàm kernel 1

```
__global__ void transpose1(int *iMatrix, int *oMatrix, int w)
{
    int r = blockIdx.y * blockDim.y + threadIdx.y;
    int c = blockIdx.x * blockDim.x + threadIdx.x;

    oMatrix[r * w + c] = iMatrix[
];
}
```

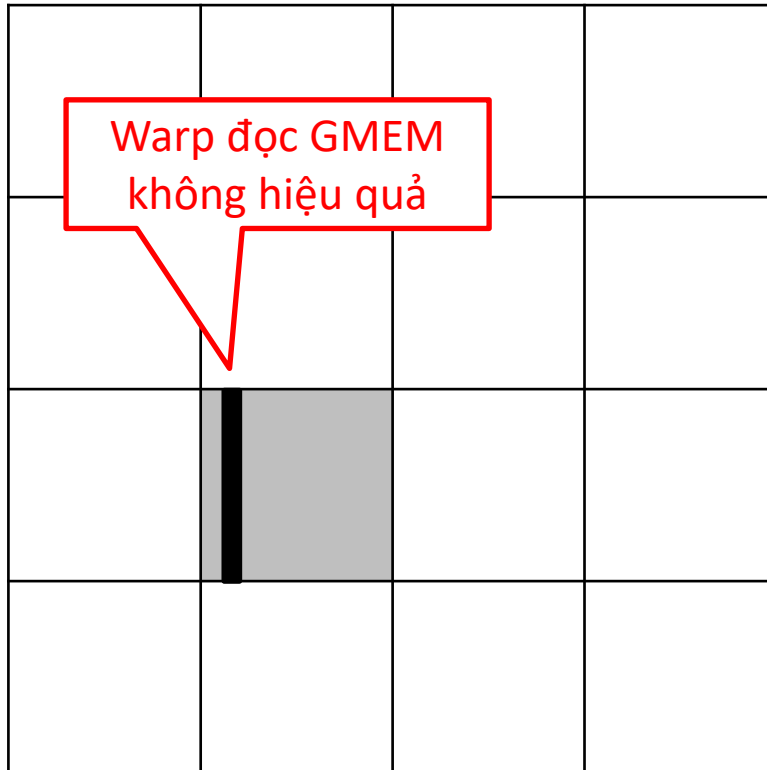
Hàm kernel 1

```
__global__ void transpose1(int *iMatrix, int *oMatrix, int w)
{
    int r = blockIdx.y * blockDim.y + threadIdx.y;
    int c = blockIdx.x * blockDim.x + threadIdx.x;

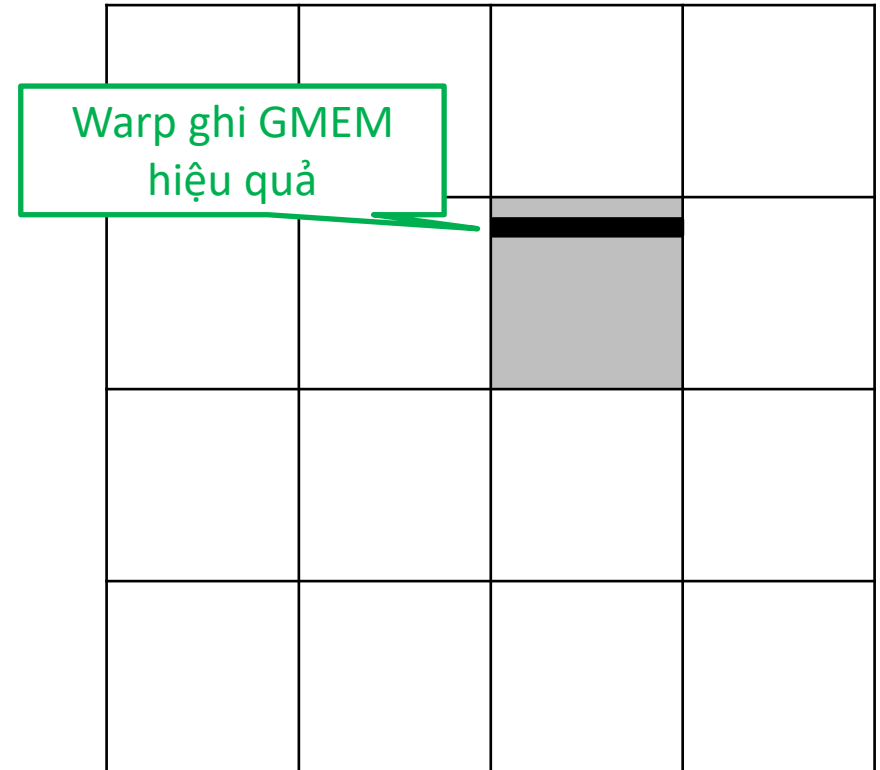
    oMatrix[r * w + c] = iMatrix[c * w + r];
}
```

Hàm kernel 1

iMatrix



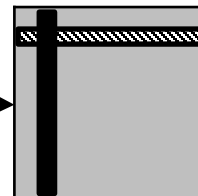
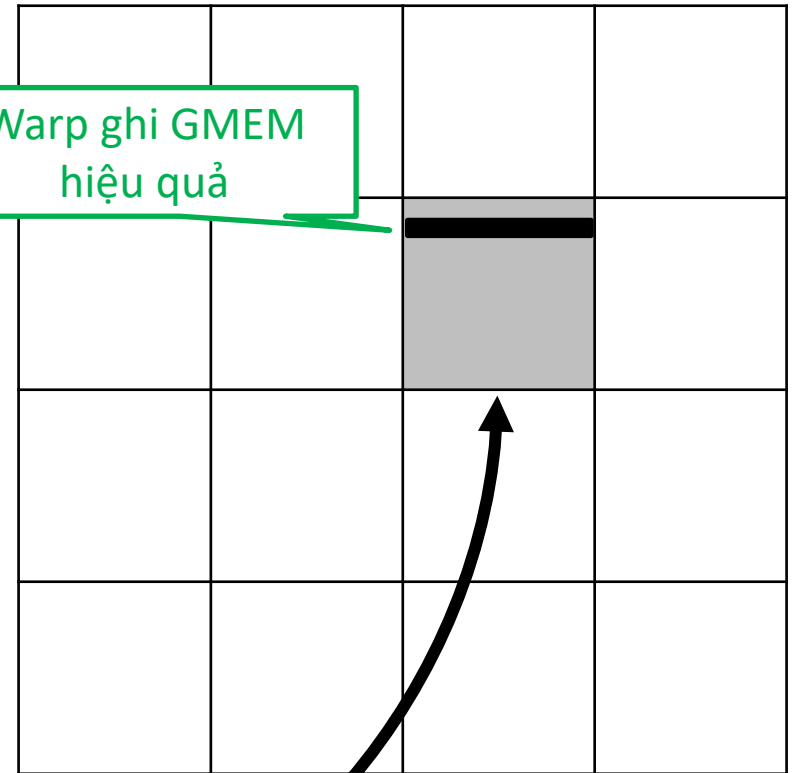
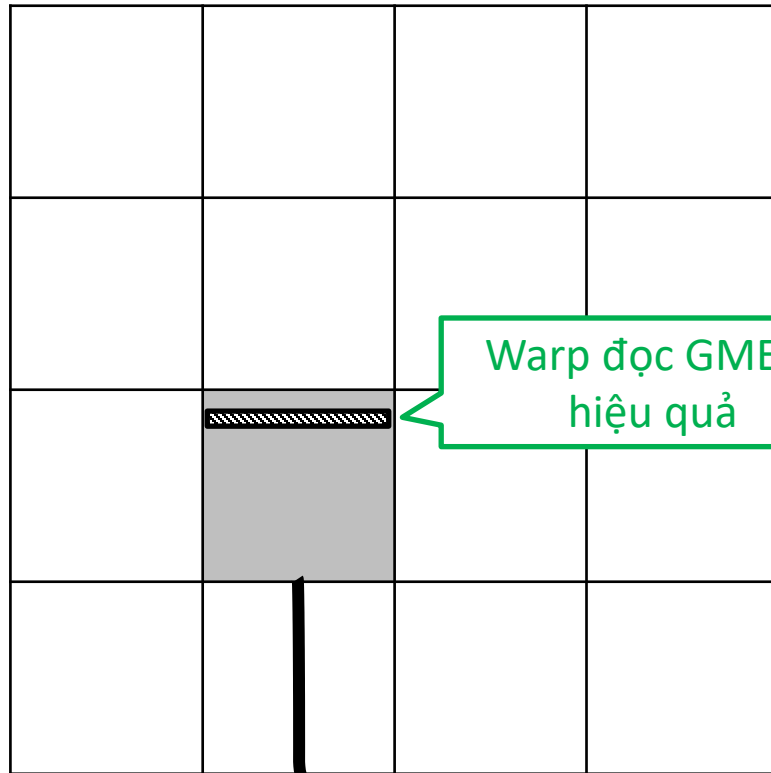
oMatrix



Cải tiến?

Hàm kernel 2

oMatrix



SMEM

Hàm kernel 2

```
__global__ void transpose2(int *iMatrix, int *oMatrix, int w)
{
    __shared__ int s_blkData[32][32];

    // Each block load data efficiently from GMEM to SMEM
    int iR =
    int iC =
    s_blkData[          ][          ] = iMatrix[iR * w + iC];
    __syncthreads();

    // Each block write data efficiently from SMEM to GMEM
    int oR = blockIdx.y * blockDim.y + threadIdx.y;
    int oC = blockIdx.x * blockDim.x + threadIdx.x;
    oMatrix[oR * w + oC] = s_blkData[          ][          ];
}
```

Hàm kernel 2

```
__global__ void transpose2(int *iMatrix, int *oMatrix, int w)
{
    __shared__ int s_blkData[32][32];

    // Each block load data efficiently from GMEM to SMEM
    int iR = blockIdx.x * blockDim.x + threadIdx.y;
    int iC = blockIdx.y * blockDim.y + threadIdx.x;
    s_blkData[threadIdx.y][threadIdx.x] = iMatrix[iR * w + iC];
    __syncthreads();

    // Each block write data efficiently from SMEM to GMEM
    int oR = blockIdx.y * blockDim.y + threadIdx.y;
    int oC = blockIdx.x * blockDim.x + threadIdx.x;
    oMatrix[oR * w + oC] = s_blkData[threadIdx.x][threadIdx.y];
}
```

Thí nghiệm

- ☐ Phát sinh ngẫu nhiên ma trận số nguyên $iMatrix$ có kích thước $2^{13} \times 2^{13}$
- ☐ GPU CC 2.0
- ☐ Thời gian chạy
 - ☐ Hàm kernel 1: 19.699713 ms
 - ☐ Hàm kernel 2: 21.157248 ms (???)

Vấn đề “bank conflict” khi truy xuất SMEM

SMEM được tổ chức thành các **bank**; vd, ở hầu hết các CC hiện nay, mỗi bank có độ rộng 4 byte:

Mảng gồm các
phần tử 4 byte
ở bộ nhớ chia sẻ

| | | | | | | | | | |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|-----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |
| 0 | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 | 36 |

... được tổ chức
thành các bank

| | | | | | | | | |
|-----------|-----------|-----------|-----------|-----------|-----------|-----|------------|------------|
| 0 | 1 | 2 | 3 | 4 | 5 | ... | 30 | 31 |
| 32 | 33 | 34 | 35 | 36 | 37 | ... | 62 | 63 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| Bank 0 | Bank 1 | Bank 2 | Bank 3 | Bank 4 | Bank 5 | ... | Bank 30 | Bank 31 |

Vấn đề “bank conflict” khi truy xuất SMEM

SMEM được tổ chức thành các **bank**; vd, ở hầu hết các CC hiện nay, mỗi bank có độ rộng 4 byte:

Mảng gồm các
phần tử 4 byte
ở bộ nhớ chia sẻ

| | | | | | | | | | |
|---|---|---|----|----|----|----|----|----|-----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |
| 0 | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 | 36 |

... được tổ chức
thành các bank

| | | | | | | | | |
|-----------|-----------|-----------|-----------|-----------|-----------|-----|------------|------------|
| 0 | 1 | 2 | 3 | 4 | 5 | ... | 30 | 31 |
| 32 | 33 | 34 | 35 | 36 | 37 | ... | 62 | 63 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| Bank 0 | Bank 1 | Bank 2 | Bank 3 | Bank 4 | Bank 5 | ... | Bank 30 | Bank 31 |

Nếu các thread trong warp truy xuất các phần tử 4 byte thuộc các bank khác nhau thì các truy xuất này sẽ được thực thi song song

Vấn đề “bank conflict” khi truy xuất SMEM

SMEM được tổ chức thành các **bank**; vd, ở hầu hết các CC hiện nay, mỗi bank có độ rộng 4 byte:

Mảng gồm các
phần tử 4 byte
ở bộ nhớ chia sẻ

| | | | | | | | | | |
|---|---|---|----|----|----|----|----|----|-----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |
| 0 | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 | 36 |

... được tổ chức
thành các bank

| | | | | | | | | |
|-----------|-----------|-----------|-----------|-----------|-----------|-----|------------|------------|
| 0 | 1 | 2 | 3 | 4 | 5 | ... | 30 | 31 |
| 32 | 33 | 34 | 35 | 36 | 37 | ... | 62 | 63 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| Bank 0 | Bank 1 | Bank 2 | Bank 3 | Bank 4 | Bank 5 | ... | Bank 30 | Bank 31 |

Nếu các thread trong warp truy xuất các phần tử 4 byte thuộc các bank khác nhau thì các truy xuất này sẽ được thực thi song song

Vấn đề “bank conflict” khi truy xuất SMEM

SMEM được tổ chức thành các **bank**; vd, ở hầu hết các CC hiện nay, mỗi bank có độ rộng 4 byte:

Mảng gồm các
phần tử 4 byte
ở bộ nhớ chia sẻ

| | | | | | | | | | |
|---|---|---|----|----|----|----|----|----|-----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |
| 0 | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 | 36 |

... được tổ chức
thành các bank

| | | | | | | | | |
|-----------|-----------|-----------|-----------|-----------|-----------|-----|------------|------------|
| 0 | 1 | 2 | 3 | 4 | 5 | ... | 30 | 31 |
| 32 | 33 | 34 | 35 | 36 | 37 | ... | 62 | 63 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| Bank 0 | Bank 1 | Bank 2 | Bank 3 | Bank 4 | Bank 5 | ... | Bank 30 | Bank 31 |

3-way bank conflict

Nếu các thread trong warp truy xuất các phần tử 4 byte khác nhau thuộc cùng một bank thì sẽ xảy ra “bank conflict”; các truy xuất này sẽ được thực thi một cách tuần tự

Vấn đề “bank conflict” khi truy xuất SMEM

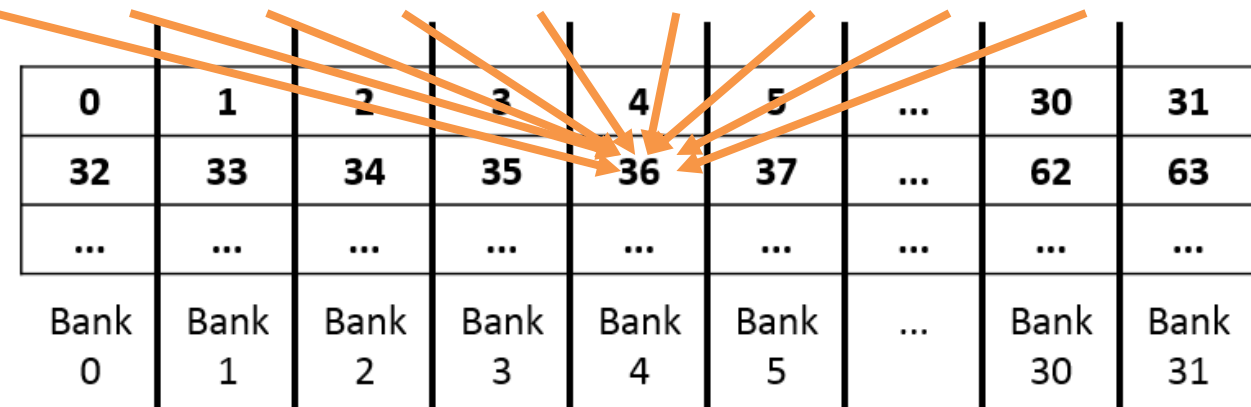
SMEM được tổ chức thành các **bank**; vd, ở hầu hết các CC hiện nay, mỗi bank có độ rộng 4 byte:

Mảng gồm các
phần tử 4 byte
ở bộ nhớ chia sẻ

| | | | | | | | | | |
|---|---|---|----|----|----|----|----|----|-----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |
| 0 | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 | 36 |

... được tổ chức
thành các bank

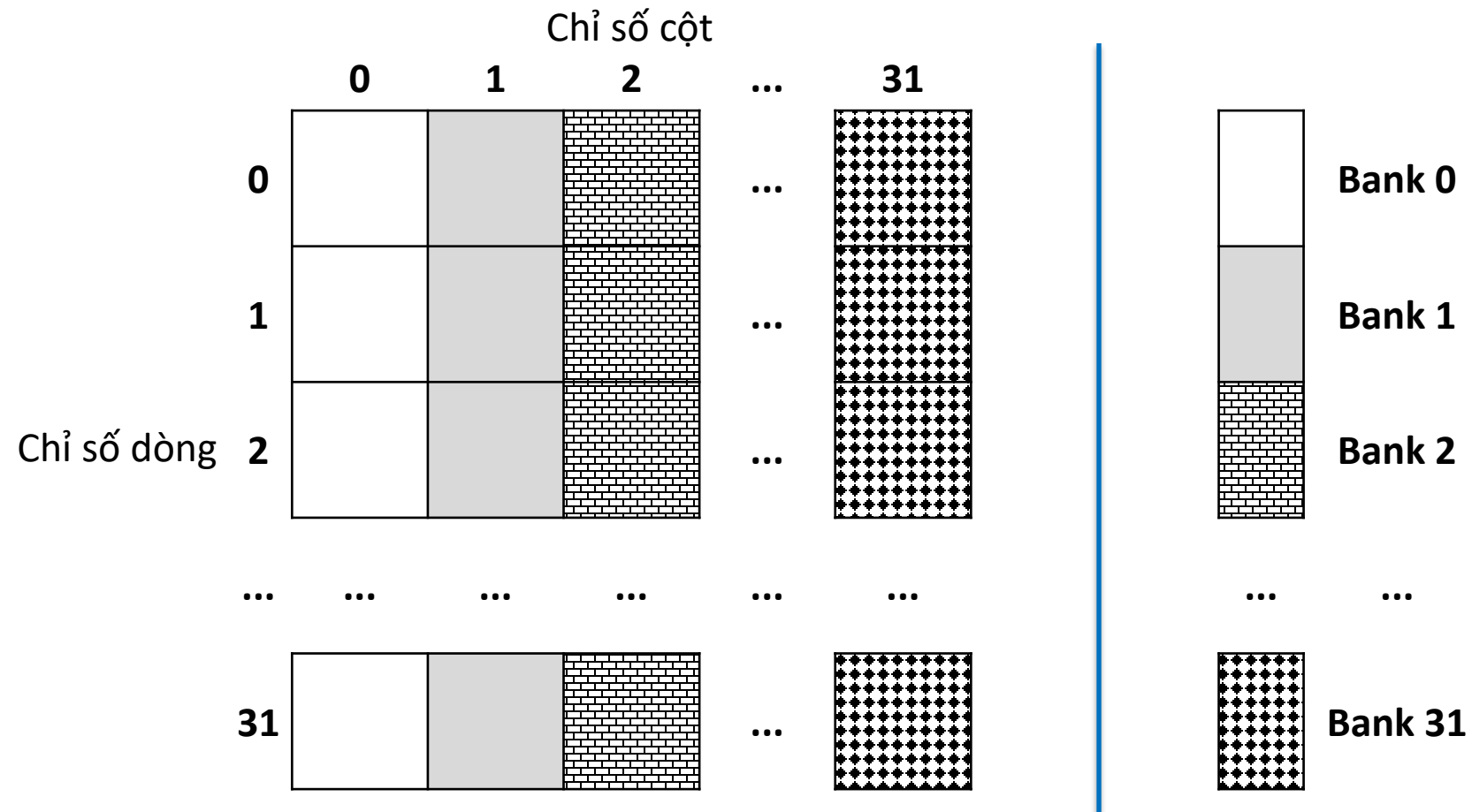
| | | | | | | | | |
|-----------|-----------|-----------|-----------|-----------|-----------|-----|------------|------------|
| 0 | 1 | 2 | 3 | 4 | 5 | ... | 30 | 31 |
| 32 | 33 | 34 | 35 | 36 | 37 | ... | 62 | 63 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| Bank 0 | Bank 1 | Bank 2 | Bank 3 | Bank 4 | Bank 5 | ... | Bank 30 | Bank 31 |



Nếu các thread trong warp cùng đọc một phần tử 4 byte thuộc một bank thì ta chỉ tốn một lần đọc; nếu ghi thì chỉ có một thread ghi được nhưng đó là thread nào thì không biết

Xét lại hàm kernel 2 ở ví dụ chuyển vị ma trận ...

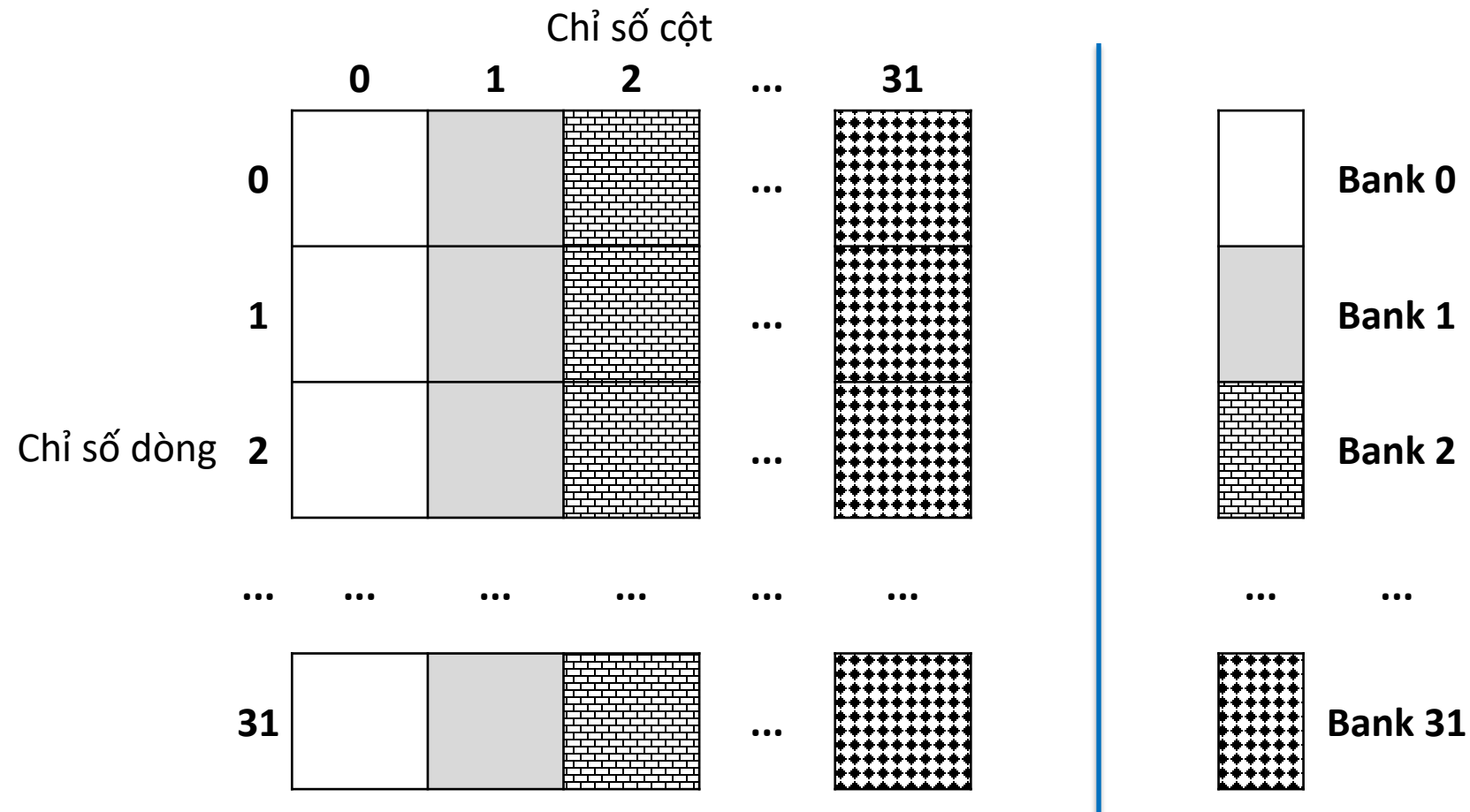
Vấn đề “bank conflict” ở hàm kernel 2



Warp ghi một dòng trên SMEM: “bank conflict”?

Warp đọc một cột trên SMEM: “bank conflict”?

Vấn đề “bank conflict” ở hàm kernel 2



Warp ghi một dòng trên SMEM: không xảy ra “bank conflict”

Warp đọc một cột trên SMEM: “bank conflict”

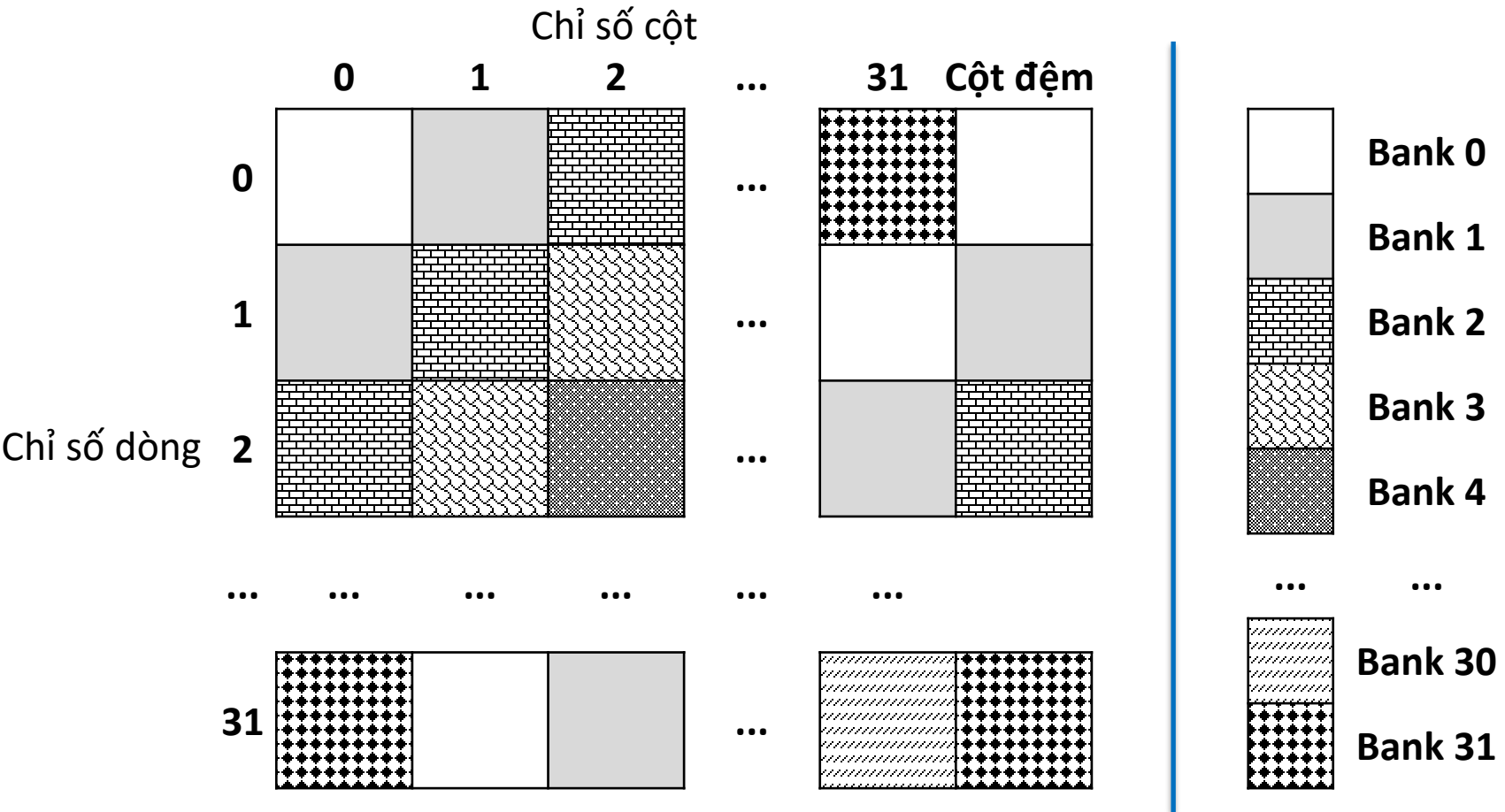
Hàm kernel 3 - giải quyết vấn đề “bank conflict” ở hàm kernel 2

```
__global__ void transpose3(int *iMatrix, int *oMatrix, int w)
{
    __shared__ int s_blkData[32][33];

    // Each block load data efficiently from GMEM to SMEM
    int iR = blockIdx.x * blockDim.x + threadIdx.y;
    int iC = blockIdx.y * blockDim.y + threadIdx.x;
    s_blkData[threadIdx.y][threadIdx.x] = iMatrix[iR * w + iC];
    __syncthreads();

    // Each block write data efficiently from SMEM to GMEM
    int oR = blockIdx.y * blockDim.y + threadIdx.y;
    int oC = blockIdx.x * blockDim.x + threadIdx.x;
    oMatrix[oR * w + oC] = s_blkData[threadIdx.x][threadIdx.y];
}
```

Hàm kernel 3 - giải quyết vấn đề “bank conflict” ở hàm kernel 2



Warp ghi một dòng trên SMEM: không xảy ra “bank conflict”

Warp đọc một cột trên SMEM: không xảy ra “bank conflict”

Thí nghiệm

- ☐ Phát sinh ngẫu nhiên ma trận số nguyên $iMatrix$ có kích thước $2^{13} \times 2^{13}$
- ☐ GPU CC 2.0
- ☐ Thời gian chạy
 - ☐ Hàm kernel 1: 19.699713 ms
 - ☐ Hàm kernel 2: 21.157248 ms
 - ☐ Hàm kernel 3: 11.426176 ms 😊