Execution on GPUs

Suggest topics for remaining lectures on Ed!
Final project gearup tomorrow, 8:30pm, CIT 265 + zoom

source

Review

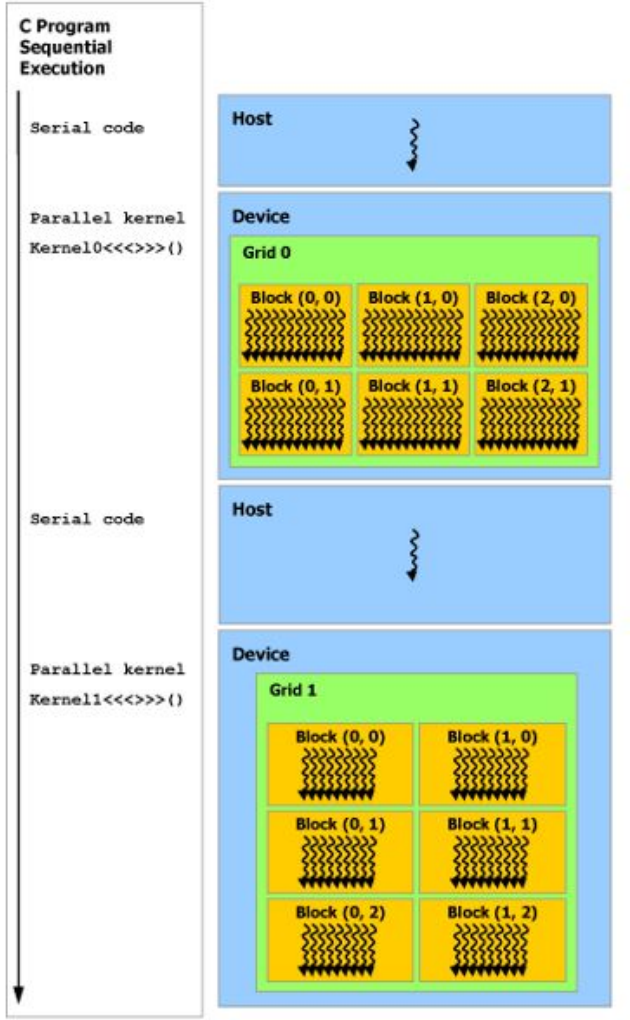
SPMD model allows us to write one program and spawn it as many **threads**

SIMT processor (**Streaming Multiprocessor** on nVidia GPUs) allows us to execute a **warp** of threads in lockstep; processor schedules warps

Threads are grouped into **blocks** (+sometimes clusters within blocks); full workload is a **grid**

CUDA is an API for programming nVidia GPUs at the thread level

Note: this is not a graphics course, and not a GPU programming course
We're studying the details of highly parallel architectures



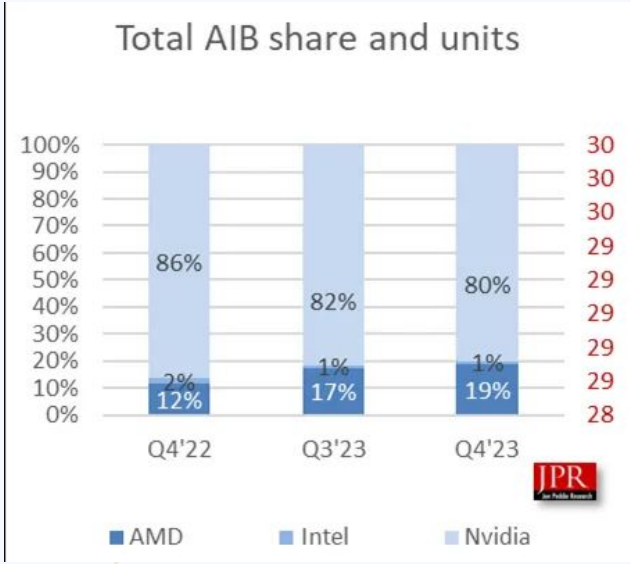
A note about nVidia

They're not paying me (but given their stocks, they could afford it…)

Using their terminology/referring to their guide for convenience and because…

I respect H&P for defining their own, non-nVidia vocabulary, but at this point it's a losing battle

(We can certainly talk about whether this amount of dominance is an overall good for society/the market/the consumer, with the caveat that I'm not an economist and my takes are casual)



source

nVidia PTX ISA

PTX = parallel thread execution

Full summary in P&H fig. B.4.3

Suffixes may define operand data type, memory space

Arithmetic: add, sub, mul, etc

Special function: sqrt, sin, cos, etc

Logical: and, or, xor, etc

Memory: ld, st, tex (texture lookup), atom

Control: branch, call, ret, sync, exit

CSR multiplication in CUDA

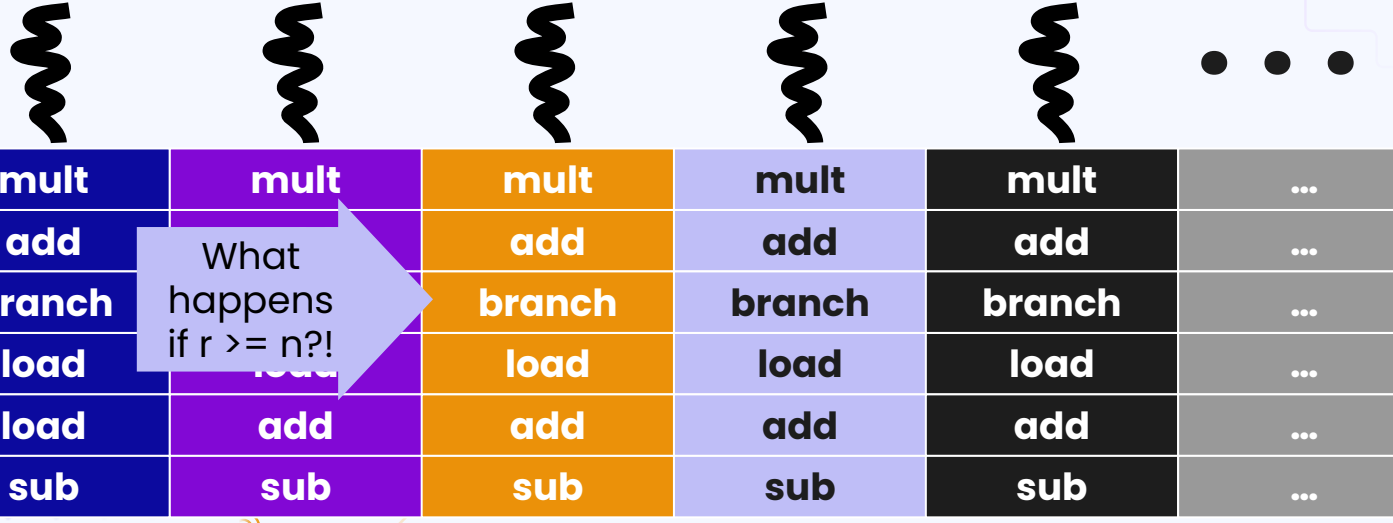
```
__host__
// set up matrix in CSR format here: ...
// 8 blocks, 256 threads per block to do multiplication
csrMult<<<8, 256>>>(2048, Rp, C, V, x, y);

// GPU side
__device__
void csrMult(int n, int* Rp, int* C, float* V, float* x, float* y) {
    int r = blockIdx.x * blockDim.x + threadIdx.x;
    if (r < n) {
        int rBeg = Rp[r];
        int rSize = Rp[r + 1] - Rp[r];
        y[r] = multRow(rSize, C + rBeg, V + rBeg, x);
    }
}
```

Instead of loop, use these provided variables as per-thread "coordinates" for data access

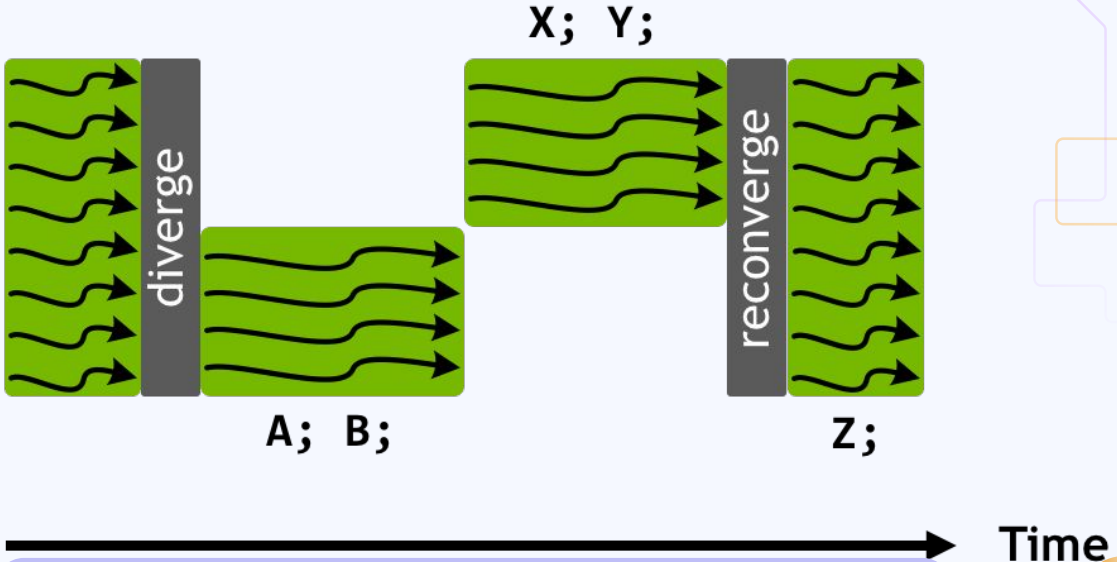
Kernel execution

```
int r = blockIdx.x * blockDim.x + threadIdx.x;
if (r < n) {
    int rBeg = Rp[r];
    int rSize = Rp[r + 1] - Rp[r];
    y[r] = multRow(rSize, C + rBeg, V + rBeg, x);
}
```



Active threads

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

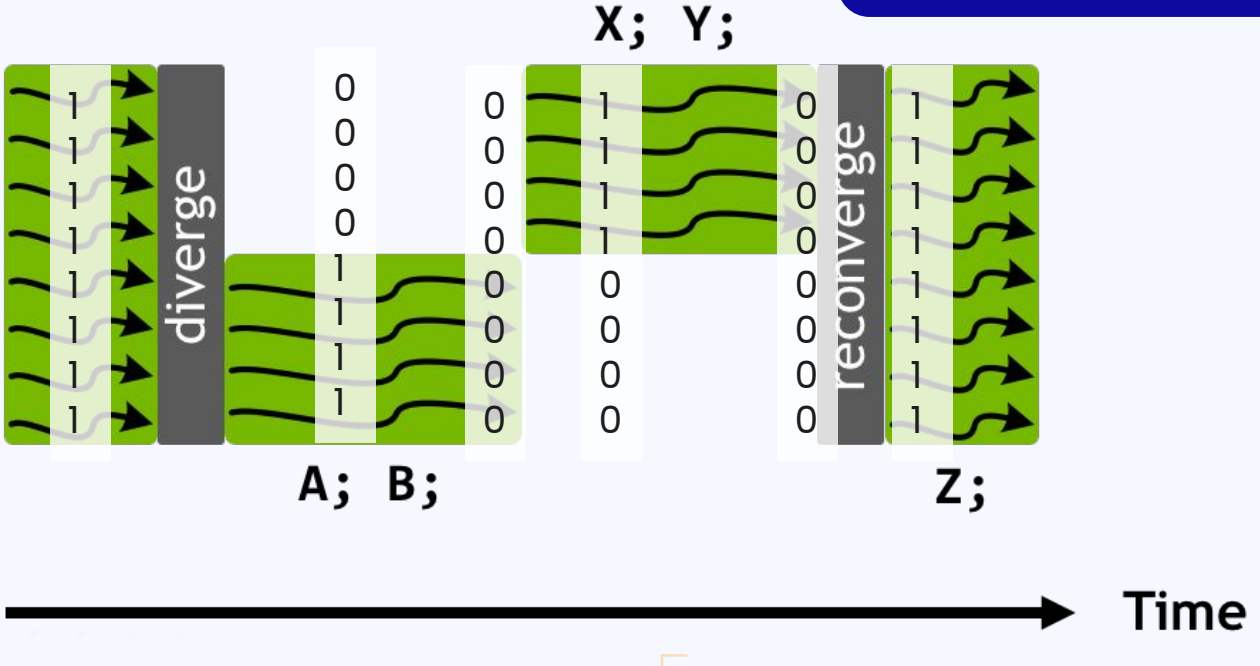


How to keep track of which threads are active?
How to keep track of when to reconverge?

image source

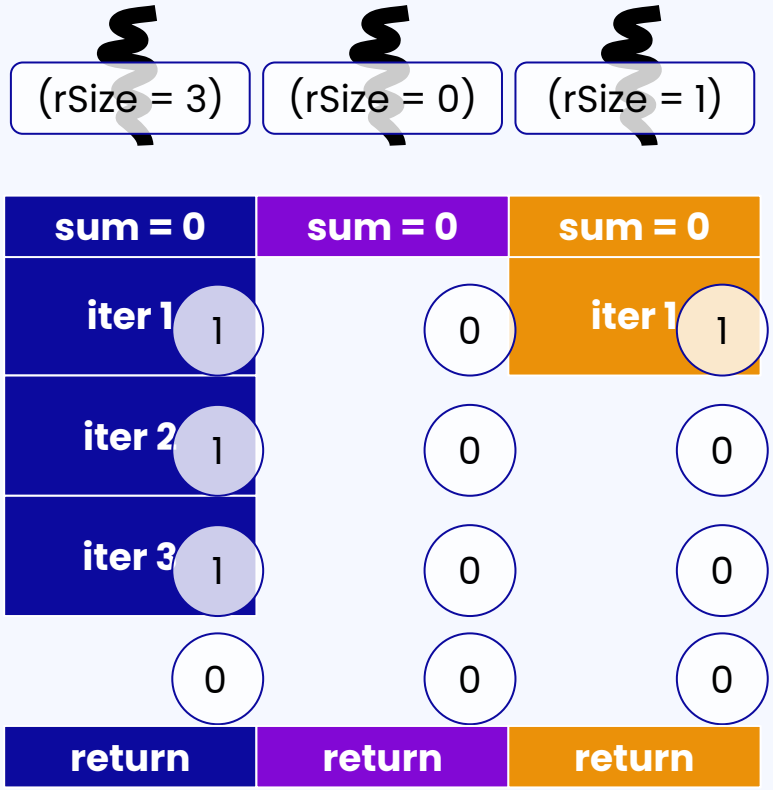
Execution mask

Use single PC and keep track of which threads are using that PC



Execution mask & loops

```
float multRow(int rSize, int* Cr,
              float* Vr, float* x) {
    float sum = 0;
    for (int i = 0; i < rSize; i++) {
        sum += Vr[i] * x[Cr[i]];
    }
    return sum;
}
```

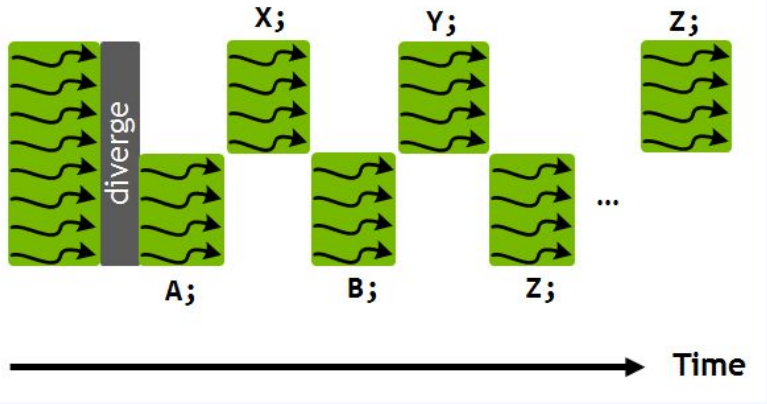


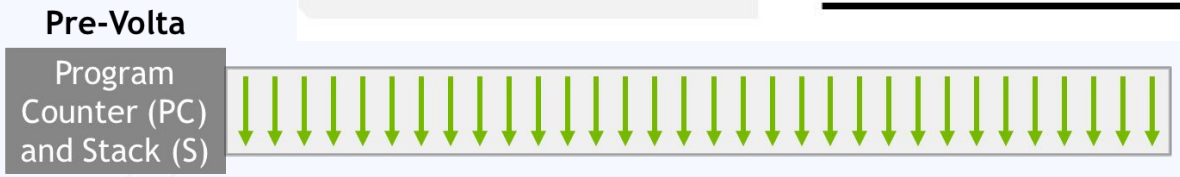
Where might execution mask usage become complicated?

nVidia Volta

Allows switching between execution paths
...how?

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```





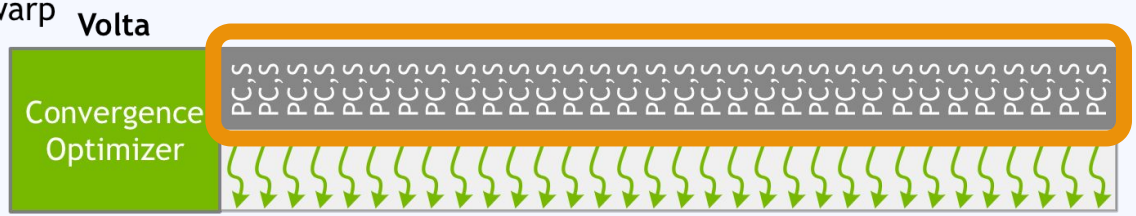


image source

Downside of independent thread scheduling: space to store thread state, convergence optimizer hardware

Upsides?

GPU memory model

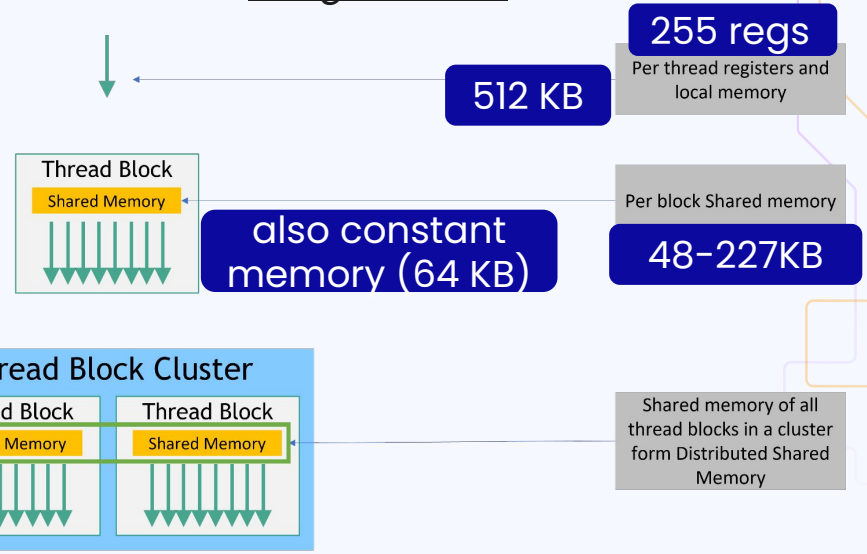
Local memory for th
state

Shared memory to communicate across threads

Resides in device memory, not on chip… why?

How?

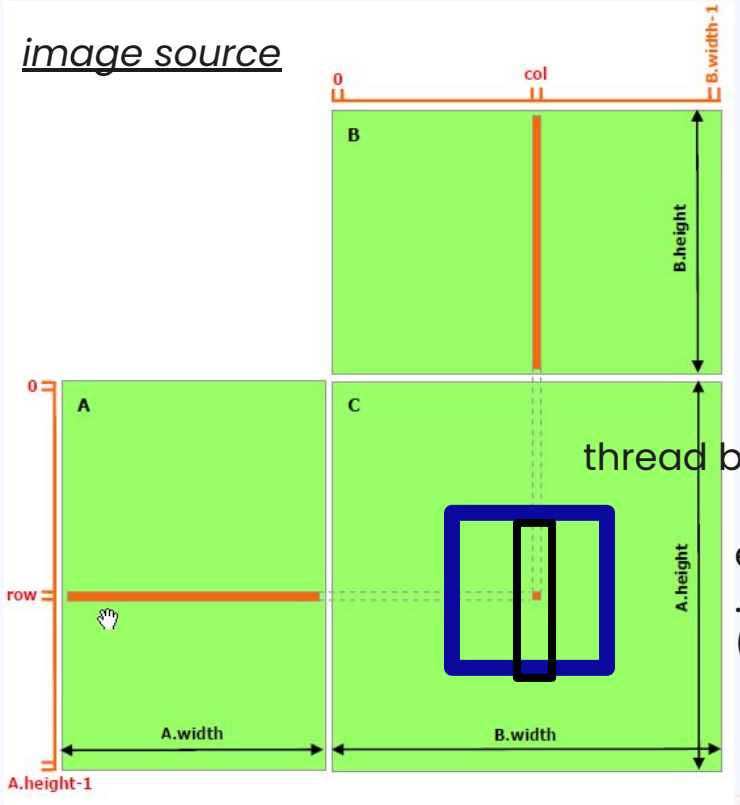
image source




```
__shared__ bool isDone;
…
if (threadIdx.x == 0)
     if (x) isDone = true;
…
__syncthreads();
if (isDone) {
     …
}
```

GPU memory: matrix multiply

image source



What's wrong with this?

multiple columns/rows may not fit in shared memory
column doesn't play well with cache lines

thread block computes these elts

every thread here will access same column
...but all the rest will access different columns
(same for rows)

not optimally taking advantage of sharing memory within block!

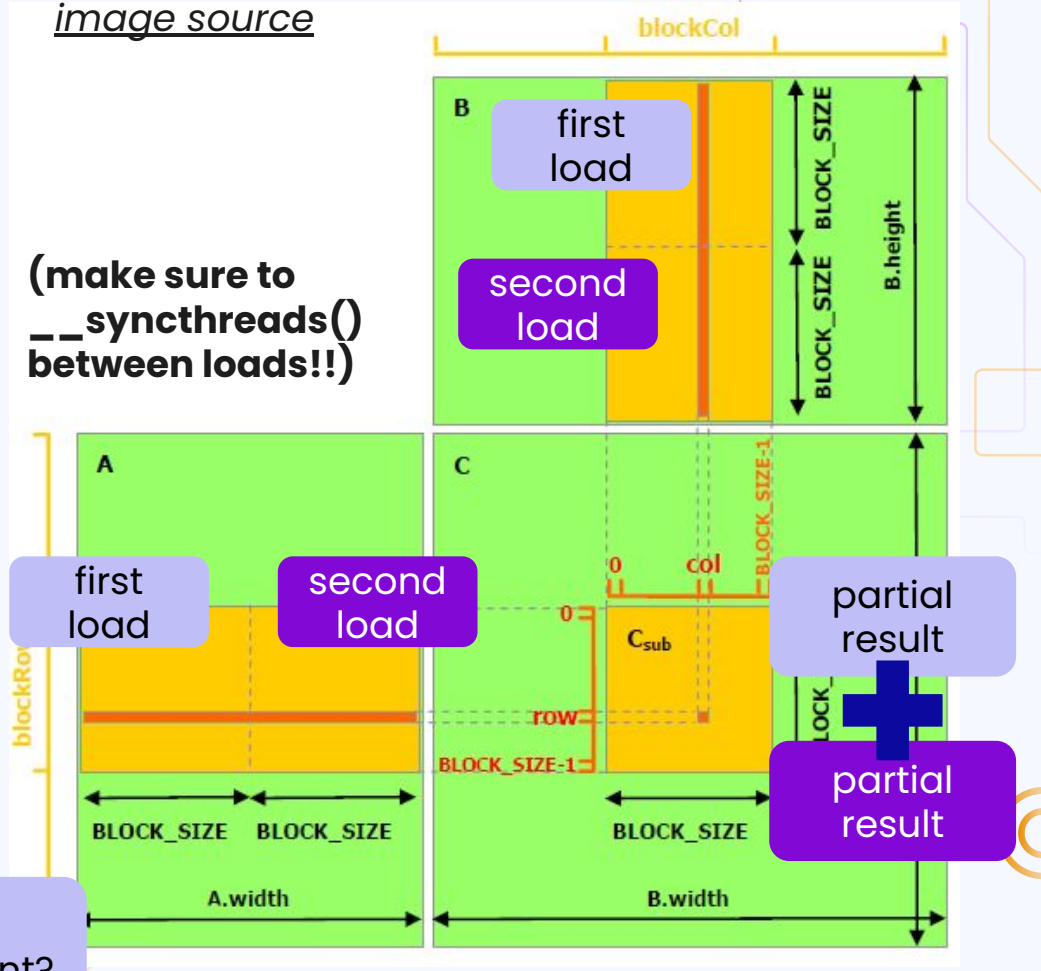
Tiling

(full code, including loading from CPU memory to device memory, in image source link)

```
int blockRow = blockIdx.y;
int blockCol = blockIdx.x;
int row = threadIdx.y;
int col = threadIdx.x;
...
// for-loop on m (number of tiles to load):
Matrix Asub = GetSubMatrix(A, blockRow, m);
Matrix Bsub = GetSubMatrix(B, m, blockCol);
__shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
__shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
As[row][col] = GetElement(Asub, row, col);
Bs[row][col] = GetElement(Bsub, row, col);
```

Each thread does two loads/stores here
Can hardware design make this more efficient?

image source



Coalesced memory access

Coalescing unit detects if accesses from same warp are in adjacent addresses and performs single, wide access (reduces uses of DRAM line)

Works for both global memory and local memory!

Important for programmer to be mindful of memory indexing

Example: avoid having each thread do its own malloc (source)

```
__shared__ int* data;
if (threadIdx.x == 0) {
    size_t size = blockDim.x * 64;
    data = (int*)malloc(size);
}
__syncthreads();
```

now adjacent threads can do adjacent accesses into `data`, eg `data[threadIdx.x]`!

Shared memory banks

Shared memory is banked (32 banks for 32 threads/warp; successive words in successive banks)

really fast as long as no bank conflicts (have to do an extra round of accesses for every conflict – can significantly slow down warp)

Which code is better for working with data of length `n = 2 * blocksize` when `i = threadIdx.x`?

```
A[i * 2] = A[i * 2] + B[i * 2] // 0, 2, 4, 6… 2n - 2
A[i * 2 + 1] = A[i * 2 + 1] + B[i * 2 + 1] // 1, 3, 5, 7… 2n - 1
```

vs

```
A[i] = A[i] + B[i] // 0, 1, 2, 3, … n - 1
A[n + i] = A[n + i] + B[n + 1] // n, n + 1, n + 2, … 2n - 1
```

What's wrong with our CSR mult?

```
void csrMult(int n, int* Rp, int* C, float* V, float* x, float* y) {
    int r = blockIdx.x * blockDim.x + threadIdx.x;
    if (r < n) {
        int rBeg = Rp[r];
        int rSize = Rp[r + 1] - Rp[r];
        float sum = 0
        for (int i = 0; i < rSize; i++) {
            sum += V[rBeg + i] * x[C[rBeg + i]];
        }
        y[r] = sum;
    }
}
```

Solution: pad and transpose

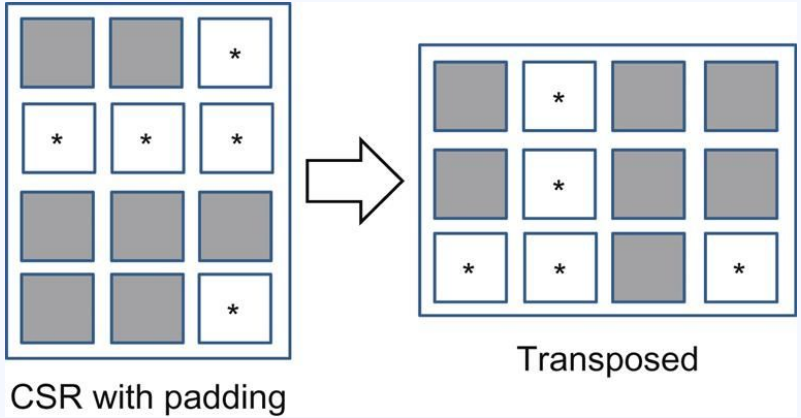


Image source: Kirk, David B., and W. Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016., figs 10.8 and 10.9
Brown library access

Padding: allows for avoiding control flow divergence
Transpose: allows for coalescing
In general will run faster, despite extraneous multiplies by 0