

# CMOR 421/521:

Matrix multiplication on GPUs  
Miscellaneous tips

M	W	F	
			1
			2
			3
			4
			5
			6
			7
			8
			9
			10
			11
			12
			13
			14
			15

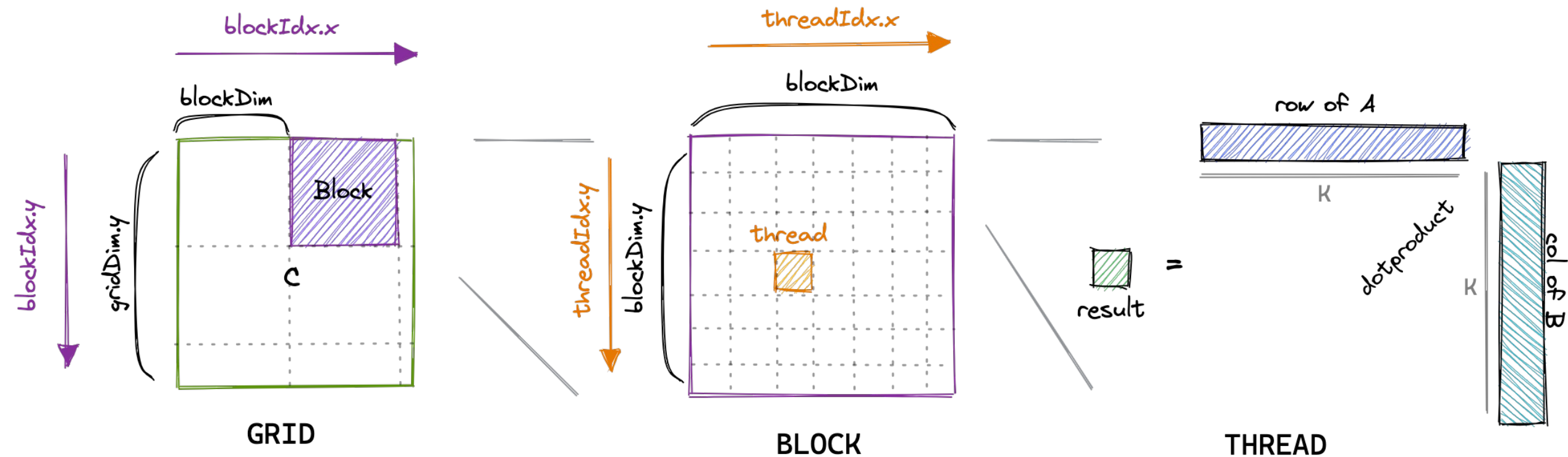
# Recap

- Writing, compiling, and executing a CUDA kernel:
  - `nvcc`, copying memory, `gridDim/blockDim`, ...
- Memory hierarchy on GPUs
  - `host << global << shared < register`
- Occupancy:
  - GPUs have massive parallelism but are constrained by shared resources (shared and register memory).
  - GPUs schedule thread blocks to run on streaming multiprocessors (SMs); using too much shared or register memory per block/thread leads to under-utilization

# A larger example: matmul on GPU

- Matrix-matrix multiplication  $C = C + A * B$  on GPU
  - Naïve vs tiled/blocked
- Similar concepts as serial optimization
  - Shared memory is now “fast memory”
  - Also have to balance issues of occupancy
  - Further optimizations: arithmetic intensity *per thread*
- Assume **row major** storage of A, B, C

# Version 1: naïve matmul



We put as many blocks into the grid as necessary to span all of  $C$

Each block is responsible for calculating a  $32 \times 32$  chunk of  $C$

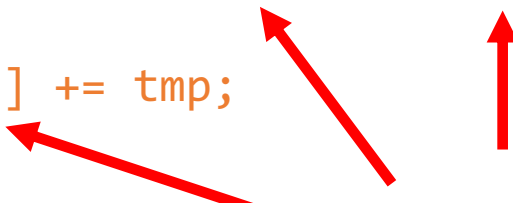
Each thread independently computes one entry of  $C$

# Version 1: naïve matmul

```
__global__ void matmul(int N, const float *A, const float *B, float *C) {  
  
    const int i = blockIdx.x * blockDim.x + threadIdx.x;  
    const int j = blockIdx.y * blockDim.y + threadIdx.y;  
  
    if (i < N && j < N) {  
        float val = 0.f;  
        for (int k = 0; k < N; ++k) {  
            val += A[i * N + k] * B[k * N + j];  
        }  
        C[i * N + j] += tmp;  
    }  
}
```

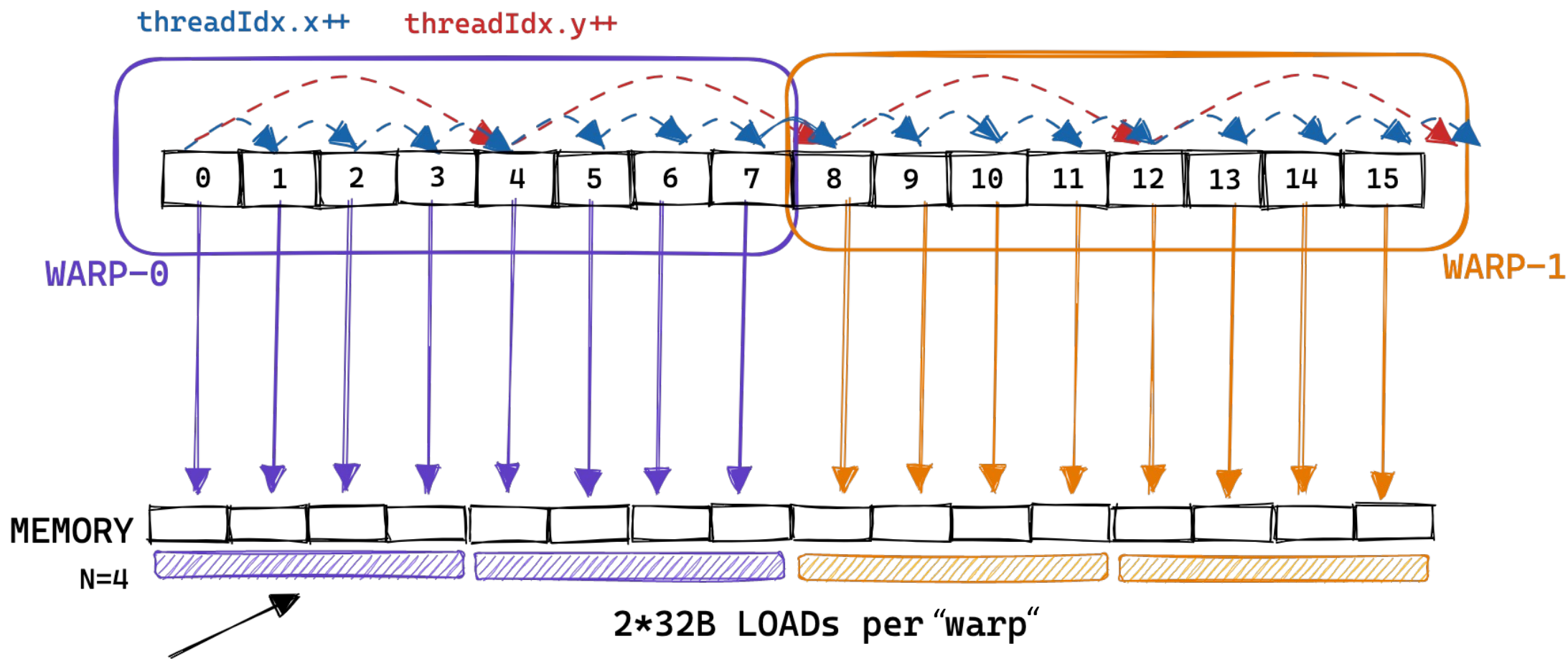
# Problems with Version 1

```
__global__ void matmul(int N, const float *A, const float *B, float *C) {  
  
    const int i = blockIdx.x * blockDim.x + threadIdx.x;  
    const int j = blockIdx.y * blockDim.y + threadIdx.y;  
  
    if (i < N && j < N) {  
        float val = 0.f;  
        for (int k = 0; k < N; ++k) {  
            val += A[i * N + k] * B[k * N + j];  
        }  
        C[i * N + j] += tmp;  
    }  
}
```



Non-coalesced memory reads/writes!

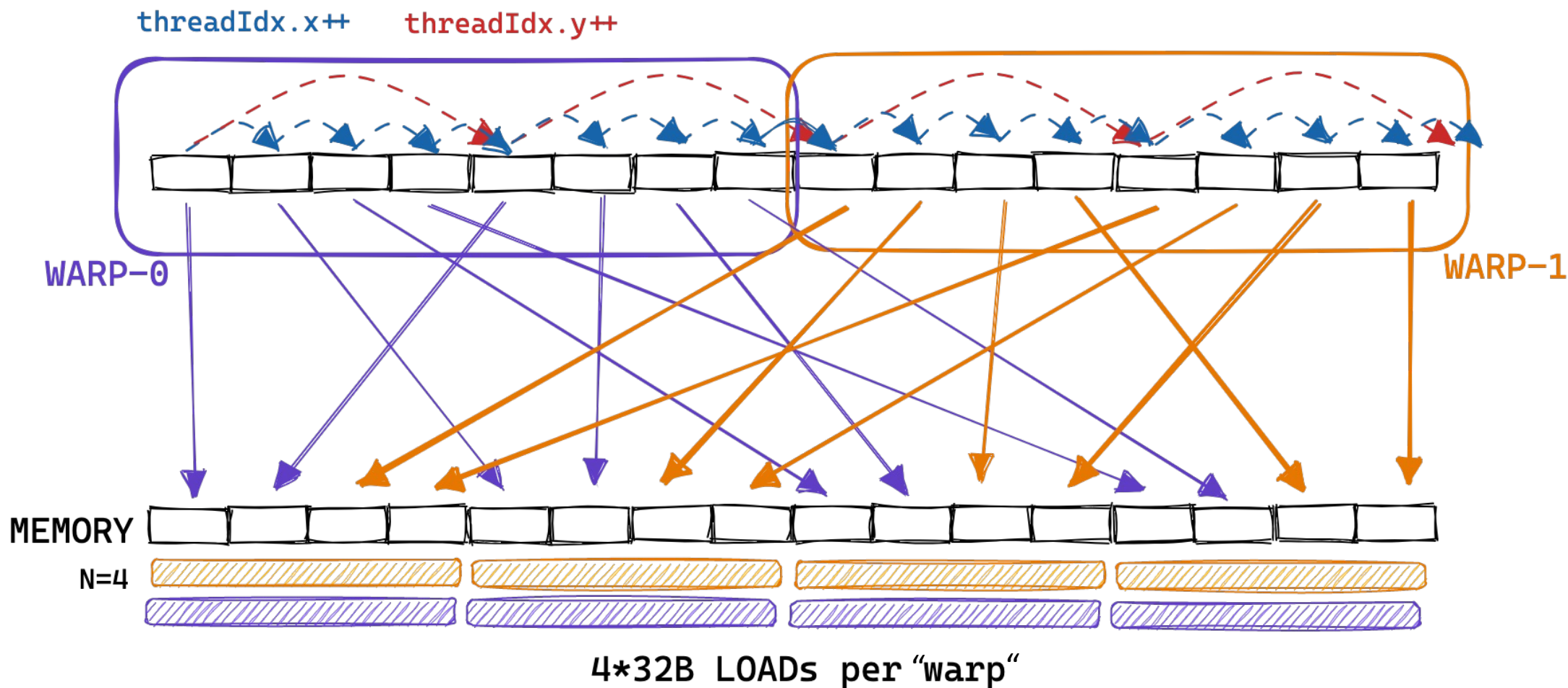
# Review of memory coalescing



4 consecutive memory accesses are grouped and executed as one LOAD

Pretend a warp is 8 threads for this illustration...

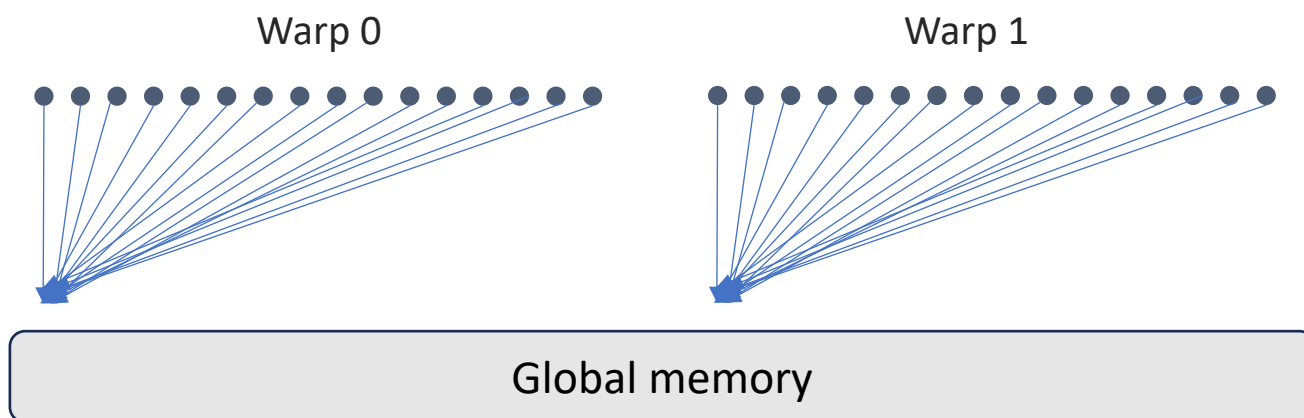
# Review of memory coalescing



Slow because memory requests are **serialized**



# Exception: global memory broadcasts



This is not coalesced, but is still fast (**does not** result in serialized memory accesses)

Robert\_Crovella  Moderator

Oct 24 '18

Memory accesses from separate warps (i.e. from threads not in the same warp) are never coalesced. Memory accesses from the same warp but emanating from different instructions in the instruction stream are never coalesced. However in each case you may still get some benefit from the cache, compared to going to global memory for all accesses.

If threads in the same warp access (read) the same location in the same instruction/clock cycle, the memory subsystem will use a “broadcast” mechanism so that only one read of that location is required, and all threads using that data will receive it in the same transaction. This will not result in additional transactions to service the request from multiple threads in this scenario.

# Fixing Version 1

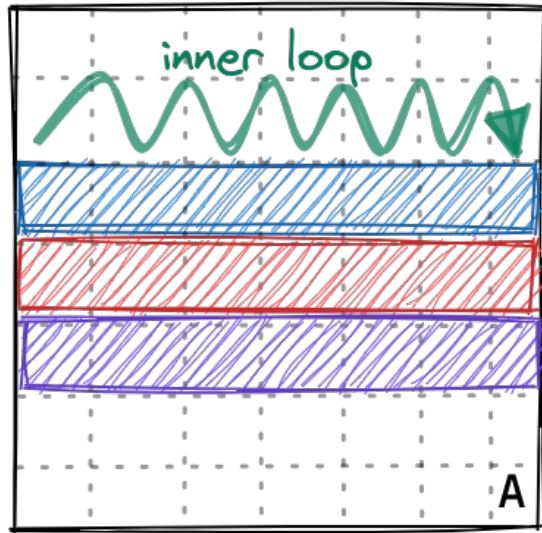
```
__global__ void matmul(int N, const float *A, const float *B, float *C) {  
  
    const int i = blockIdx.x * blockDim.x + threadIdx.x;  
    const int j = blockIdx.y * blockDim.y + threadIdx.y;  
  
    if (i < N && j < N) {  
        float val = 0.f;  
        for (int k = 0; k < N; ++k) {  
            val += A[i * N + k] * B[k * N + j];  
        }  
        C[i * N + j] += tmp;  
    }  
}
```

If all threads in a warp  
(or block) access this,  
the memory read is  
“broadcasted” instead of  
serialized.

Non-coalesced memory reads/writes!

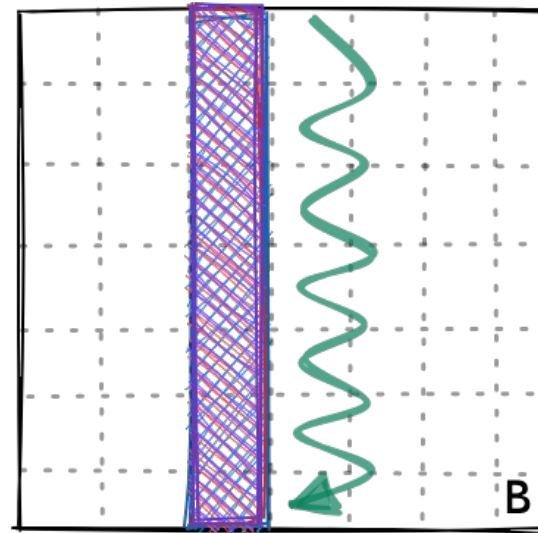
# Fixing Version 1

Naive kernel:



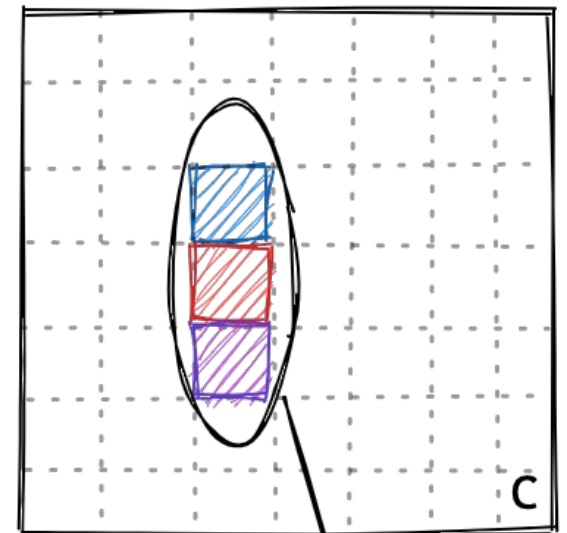
threads access non-consecutive values  $\Rightarrow$  cannot coalesce

@



all threads access same values  $\Rightarrow$  within-warp broadcast

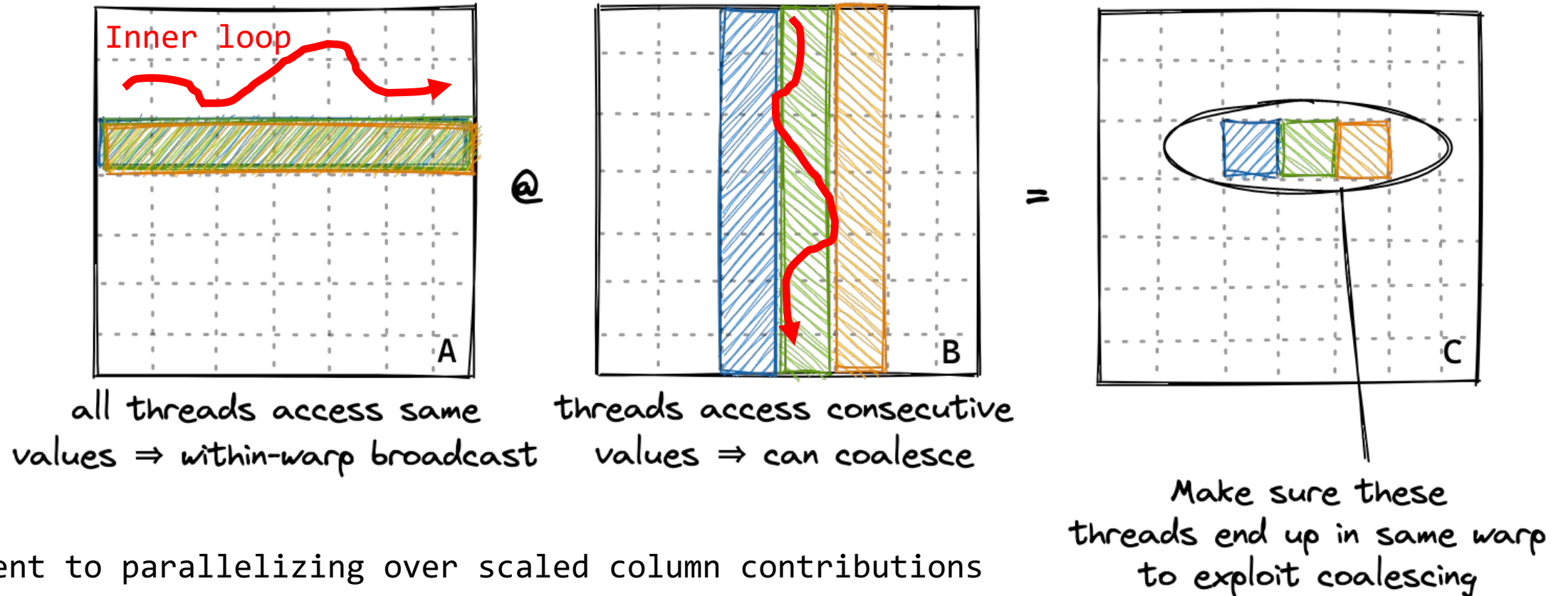
"



No benefit to putting these threads in same warp

# Version 2: coalesced memory

Coalescing kernel:



Equivalent to parallelizing over scaled column contributions

# Version 2: coalesced memory

```
__global__ void matmul(int N, const float *A, const float *B, float *C) {  
  
    const int i = blockIdx.x * BLOCKSIZE + (threadIdx.x / BLOCKSIZE);  
    const int j = blockIdx.y * BLOCKSIZE + (threadIdx.x % BLOCKSIZE);  
  
    // note: this part didn't change!  
    if (i < N && j < N) {  
        float val = 0.f;  
        for (int k = 0; k < N; ++k) {  
            val += A[i * N + k] * B[k * N + j];  
        }  
        C[i * N + j] += tmp;  
    }  
}
```

Note: this implementation only uses a 1D thread block (divided into warp-sized 32 thread chunks).

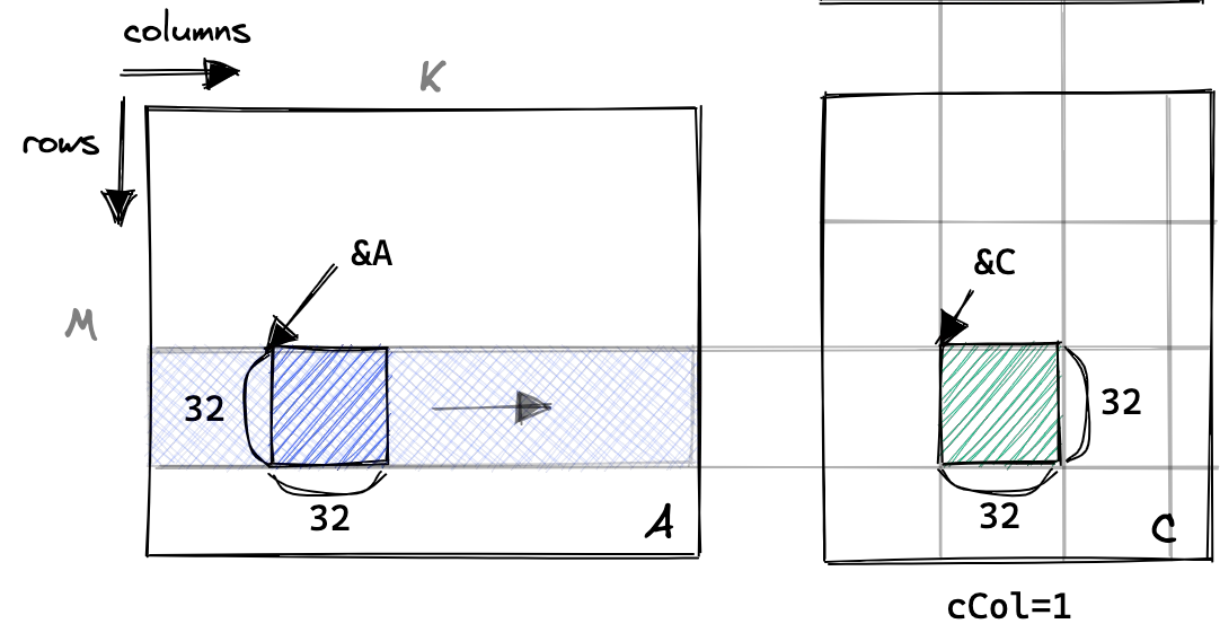
e.g., `dim3 blockDim(32*32);`

# Version 3: shared memory

- Store A, B in shared memory ( $32 \times 32 = 1024$  entries = maximum number of threads)
- Sweep over blocks of A, B, with each thread computing a single entry of  $A \times B$

Outer loop:

Advance  $\&A, \&B$  by size of cacheblock ( $= 32 \times 32$ ) until C is fully calculated



# Version 3: shared memory

- Left as an exercise for Homework 4
- Some tips:
  - 1D thread block (same as Version 2) to exploit coalescing when reading global memory into shared memory.
  - Statically #define BLOCKSIZE, s\_A and s\_B should be BLOCKSIZE \* BLOCKSIZE.
  - Local indexing into s\_A, s\_B based on threadIdx
  - Have to keep track of starting location of current block of A, B. Pointer arithmetic may help.

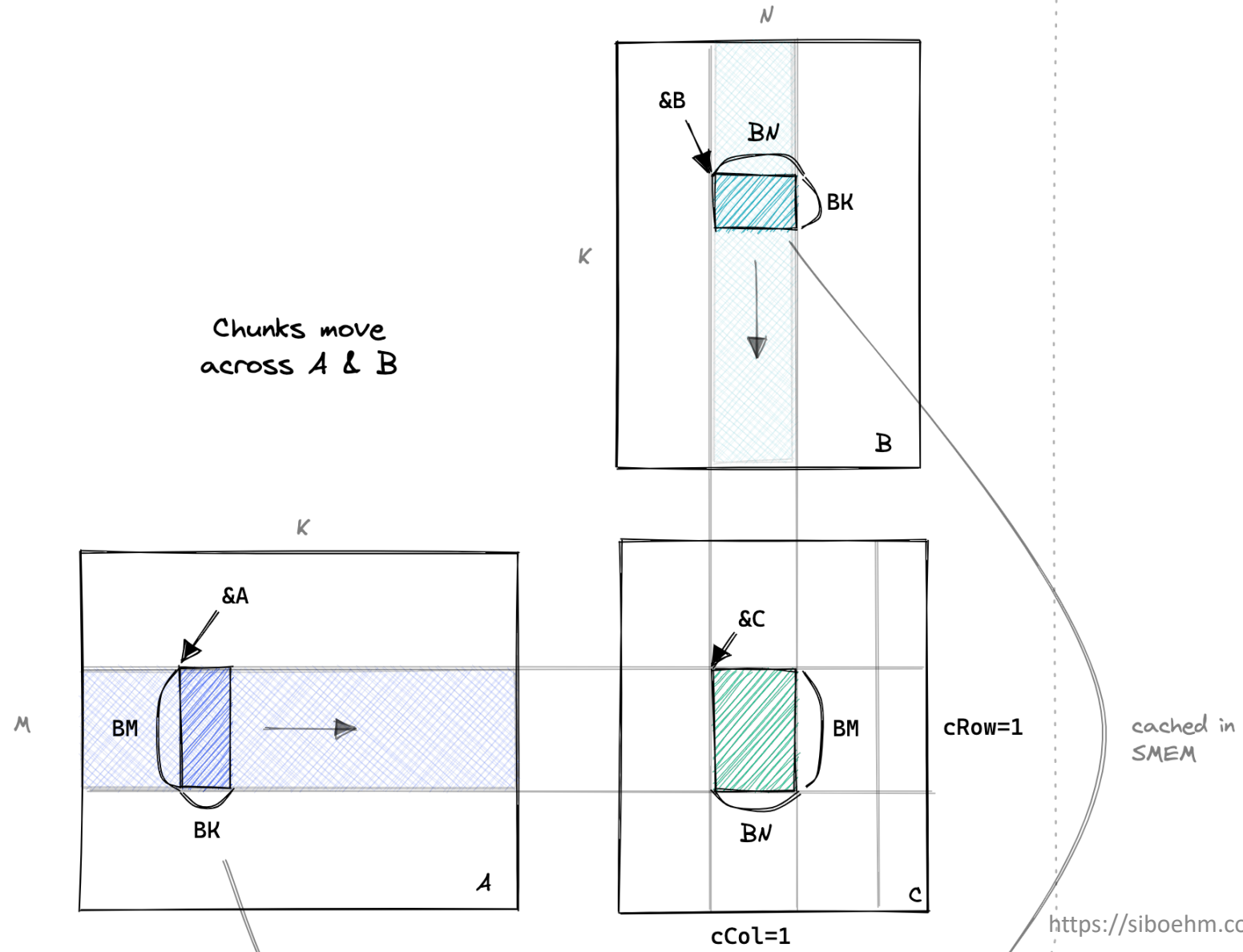
# Version 4

- Version 3 (~3000 GFLOPS/sec) provides a speedup of about 2.5x over Version 1 (~300 GFLOPS/sec)
- Version 4 achieves ~8500 GFLOPS/sec
- Motivation: profiling can reveal that shared memory reads take up most of the time.
  - If shared memory reads are the bottleneck, reducing shared memory usage should speed things up more.
  - Solution: each thread processes *multiple* elements of  $A*B$ , use **register memory** instead of shared memory.

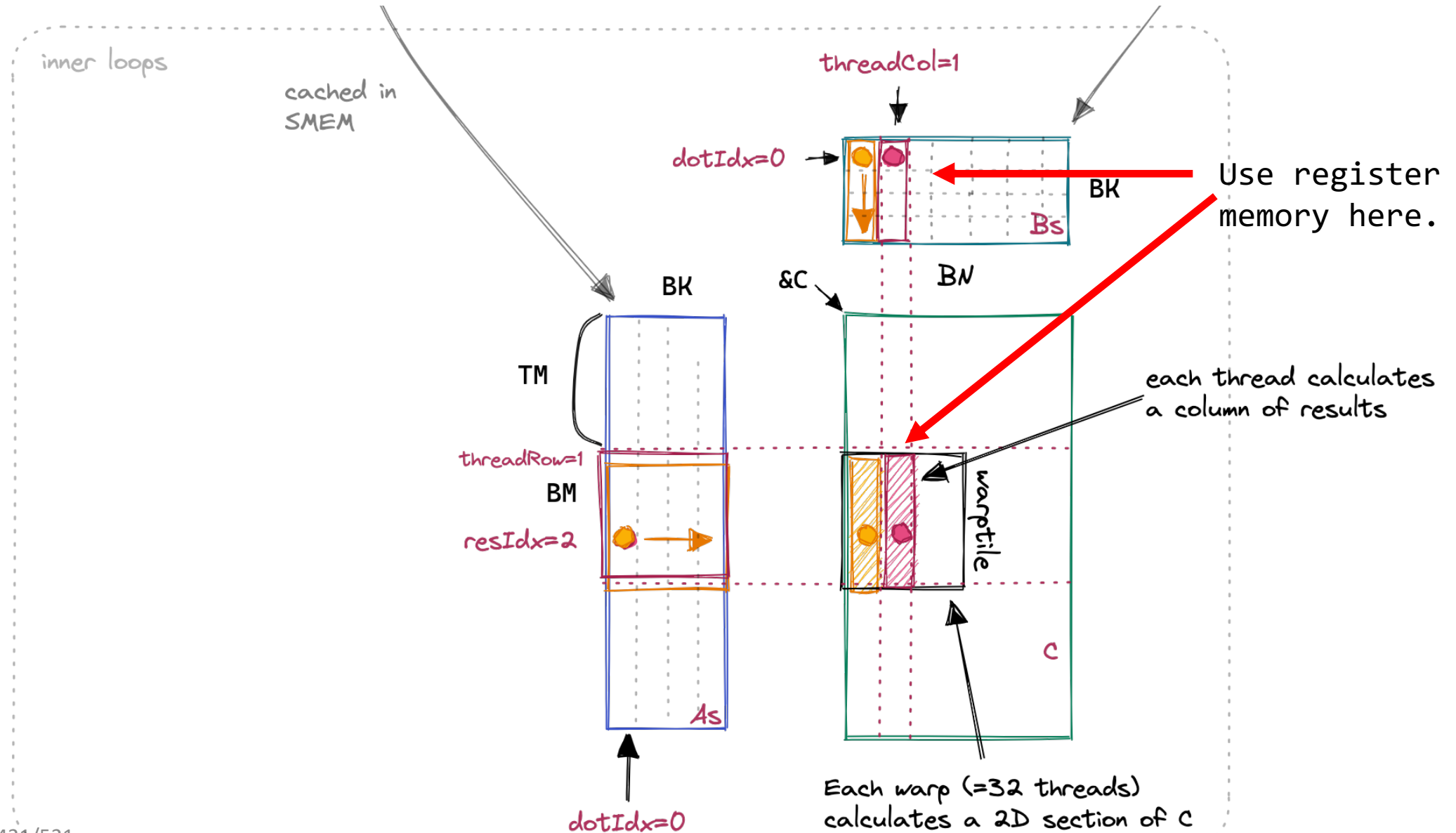


# Version 4: outer iteration

outer loop



# Version 4: inner iteration



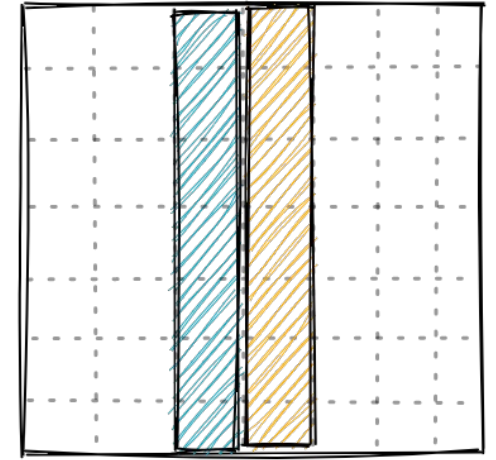
# Version 5: now in 2D

Calculating 4 results per thread requires:

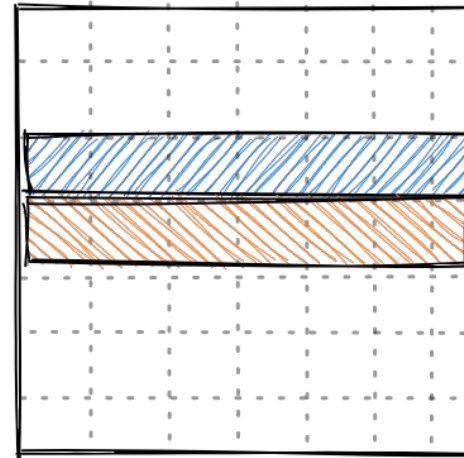
- 14 loads from A
- 14 loads from B
- 4 loads & 4 stores to C

⇒ 8 loads & 1 store per result

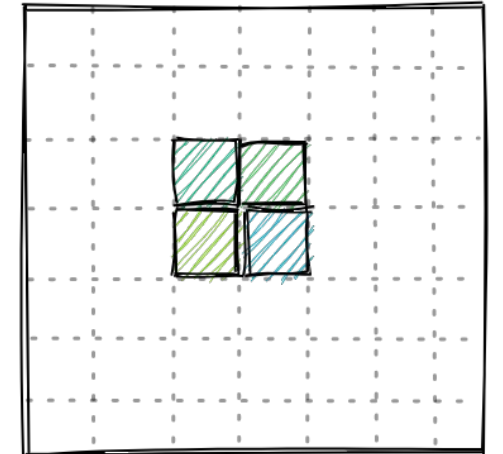
- About 1/2 as many memory accesses per result
- Version 5 achieves ~16000 GFLOPS/sec vs ~8500 GFLOPS/sec (Version 4)



B



A



C

# Versions 6, 7, 8

- Version 6 (**vectorizing** global and shared memory accesses) achieves ~18000 GFLOPS/sec
- Version 7 (autotuning computational parameters) achieves ~20000 GFLOPS/sec
- Version 8 (exploiting warp structure – somewhat hardware specific) achieves ~22000 GFLOPS/sec
- Still the overall winner: Nvidia's [cuBLAS library](https://siboehm.com/articles/22/CUDA-MMM) achieves ~24000 GFLOPS/sec
  - In general, no need to write your own matmul...

# When to stop?

- General rule of thumb on code optimizing: 90% of your total effort will be spent achieving the last 10% of maximum performance.
- Profiling usually reveals low-hanging fruit
- Examine instructions generated by a compiler (e.g., using [Godbolt](#)) can guide more nuanced optimization.
- In real life, developer time is still a lot more expensive than compute time...

# Some miscellaneous topics

- There are a lot of ways to do small (<5% speedup) GPU optimizations. However, these usually add up.
- Some useful topics:
  - Error checking (saving developer time)
  - Single vs double precision operations
  - Loop unrolling in CUDA
  - Vectorization using packed float4, int4

# Error checking

- An annoying aspect of CUDA: silent failures
- “add\_vectors” as an example
- Can add error handling for CUDA calls (CUDA library functions are handled a little differently).

```
add_vectors <<< gridDim, blockDim >>> (N, d_x, d_y);

cudaError_t code = cudaGetLastError();
if (code != cudaSuccess){
    printf("GPUassert: %s\n", cudaGetErrorString(code));
}
```

# More robust: grid-stride loops

- Can avoid invalid kernel arguments (e.g., too many blocks) by using grid-stride loops.
- `blockSize` and `numBlocks` can be chosen arbitrarily.

```
__global__ void add(const int N, const float *x, float *y){  
    const int i = blockIdx.x * blockDim.x + threadIdx.x;  
    for (int ii = i; ii < N; ii += blockDim.x * gridDim.x){  
        y[ii] += x[ii];  
    }  
}
```



# An example of grid-stride loops

- Compare `add_vectors.cu` and `add_vectors_gridstride.cu`

# float vs double in CUDA

- GPUs are generally much more efficient at single precision operations (more SP cores).
  - Your performance may drop if you introduce double precision operations
- Be careful with numeric constants (e.g., 1.0 (double) vs 1.f (float) or “#define PI 3.14159**f**”)

```
__global__ void add(const int N, const float *x, float *y){  
    const int i = blockIdx.x * blockDim.x + threadIdx.x;  
    if (i < N){  
        y[i] = y[i] + 2.0 * x[i]; // this result gets promoted to double!  
    }  
}
```

# float vs double in CUDA

- TL;DR: be careful of 1.0 vs 1.f!
- Use “nvprof” to catch stray double (nvprof -metrics flop\_count\_dp, flop\_count\_sp ...)
- Can be tricky if you want to keep the option of both single and double precision
  - Option: use a macro to switch between float and double

```
#define datafloat (float or double)  
datafloat a = 1.f; // will cast to a double if datafloat==double
```

# Loop unrolling in CUDA

- Common compiler optimization: unrolling a statically sized for loop
- After loop unrolling, the following code becomes:

```
#pragma unroll
for (int i = 0; i < 4; ++i)
    y[i] = y[i] + x[i];
```

```
y[0] = y[0] + x[0];
y[1] = y[1] + x[1];
y[2] = y[2] + x[2];
y[3] = y[3] + x[3];
```

# Loop unrolling in CUDA

- Why is loop unrolling an optimization?
  - Slightly reduces work (initializing “int i”, checking if “i < 4”, incrementing “i”)
  - Probably more impactful: unrolling tends to improve Instruction Level Parallelism (ILP)
- Downside: unrolling a large loop can lead to higher register usage (“register pressure”).

# Tools: nvprof

- nvcc is useful for static analysis of kernels
- nvprof: useful for runtime analysis
  - Not just an easier way to time kernels
  - nvprof metrics provide very detailed insight into performance of CUDA kernels
- Nvidia Visual Profiler (nvvp)
  - Will suggest areas to optimize based on nvprof data
- Modern GPUs (compute capability 7.5 and up) have moved to “Nsight”

# Getting nvprof to work on NOTS

- `ml purge`
- `module load GCC/8.3.0`
- `module load CUDA/10.1.168`
- `nvcc add_vectors.cu`
  - Optional: `-arch=sm_37` will give more precise nvprof info
- `nvprof --metrics all ./a.out 10000 128`

# Vectorization using int4, float4

- In addition to coalesced memory reads, GPUs are capable of SIMD memory reads and vectorized operations per-thread
  - 128-bit vector lanes = can vectorize load/stores (or operations) on 4 int or float values
  - Same with 2 double values, or a struct whose size is a power of 2

```
struct Foo {int a, int b, double c}; // 16 bytes in size
Foo *x, *y;
...
x[i]=y[i];
```



# Vectorization example

```
const float4 x1 = x[i];  
float4 y1 = y[i];
```

```
y1.x += x1.x;  
y1.y += x1.y;  
y1.z += x1.z;  
y1.w += x1.w;
```

```
y[i] = y1;
```

```
reinterpret_cast<float4*>(y)[i] += reinterpret_cast<float4*>(x)[i];
```

# Vectorization demo

- Compare `add_vectors_gridstride.cu` and `add_vectors_float4.cu`

# Alternatives to CUDA

- Some GPU support in OpenMP 4.0 and beyond
- OpenACC (acc = accelerators) is more mature
  - IMO not as good as OpenMP is for multi-threading, only really supports Nvidia GPUs and CUDA

```
#pragma acc kernels copyin(a[0:n],b[0:n]), copyout(c[0:n])  
for(i=0; i<n; i++) {  
    c[i] = a[i] + b[i];  
}  
  
// compile with -acc flag, similar to -fopenmp framework
```

# Alternatives to CUDA

- “Portability” libraries: translate from a domain-specific language to CUDA, Metal, OpenMP, etc.
- Kokkos: Sandia National Laboratories, most popular, aims to work with standard C++ functionality
- Raja: Lawrence Livermore National Labs (LLNL), allows for more detailed optimizations.
- OCCA: Virginia Tech + LLNL, lightweight, maps to both GPU/OpenMP/CPU. Highly optimizable.
- Others: AMD HIP, SYCL, HPX, ROCm.

# GPU computing using Julia

- `CUDA.jl`
- `KernelAbstractions.jl`
- A little annoying to set up
  - Have to be careful about Julia versions (cannot use module). I used Julia v1.10.2 installed via [juliaup](#).
  - I used CUDA/11.7.0 and GCC/11.3.0 (or lower). Note that one can compile kernels with “`nvcc -arch=sm_37 ...`”, but `nvprof` doesn't work with this version...
  - Request 8G or more memory (to avoid precompilation hanging)
  - I use VSCode + Remote SSH for syntax highlighting.

# GPU computing using Julia

- `CUDA.jl`
- `KernelAbstractions.jl`

# CUDA.jl

- CUDA.jl has many limitations compared to Julia:
  - (Generally) cannot allocate memory
  - File I/O is not allowed
  - Kernels cannot return an argument
  - Type unstable code will not compile
- Note: uses 1-indexing (to match Julia).
- Works well with broadcast and map
- Linear algebra routines use CUBLAS

# CUDA.jl was hard to set up

- Have to be careful about Julia versions (cannot use the module version on NOTS). I used Julia v1.10.2 installed via [juliaup](#).
- I used CUDA/11.7.0 and GCC/11.3.0 (or lower).
  - Note that one can compile kernels with “nvcc -arch=sm\_37 ...”, but nvprof doesn’t work with this version...
- I needed to request 8G or more memory (to avoid precompilation hanging when installing CUDA.jl)
- I use VSCode + Remote SSH for syntax highlighting.



# CUDA.jl

- Demo

# CUDA.jl

- Introspection on kernel objects
  - `CUDA.registers(kernel)`
  - `CUDA.memory(kernel)`
- Usually [CUDA.@profile](#) serves as a substitute for `nvprof`, but it doesn't seem to work on NOTS.
- Note: docs may introduce features which are not available for your GPU's compute capability

# KernelAbstractions.jl (KA.jl)

- Main steps
  - Write a function using KA.jl syntax
  - If the input is a GPU data type and CUDA.jl is installed, it will generate a CUDA kernel
  - If the input is a CPU data type, it will generate a regular Julia function.
- A little rough around the edges, very much WIP.