
PPAR: GPU code optimization

Caroline Collange
she/her

caroline.collange@inria.fr
<https://team.inria.fr/pacap/members/collange/>

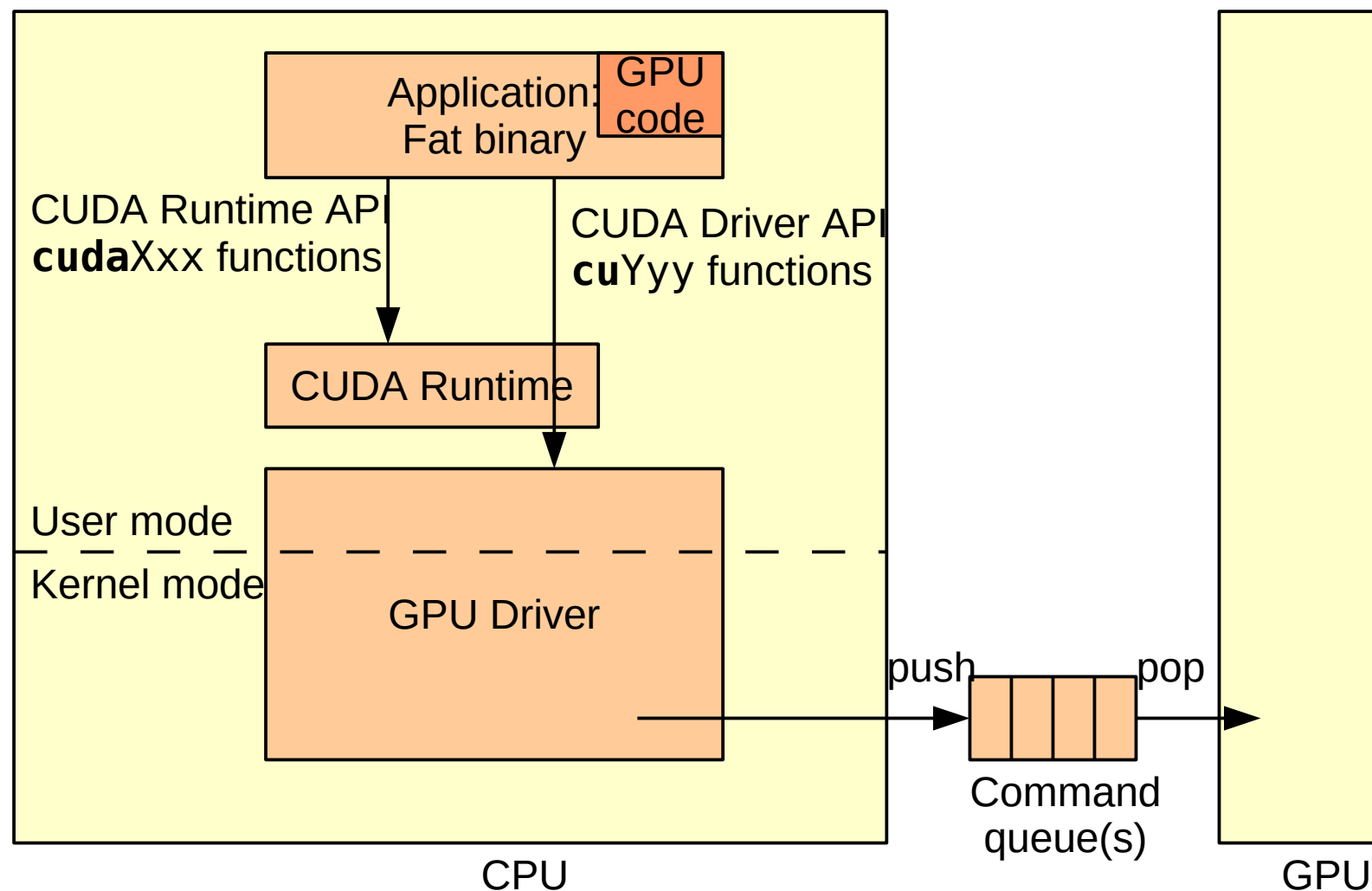
Master 1
PPAR - 2022

Outline

- Host-side task and memory management
 - ◆ Asynchronism and streams
 - ◆ Compute capabilities
- Work partitioning and memory optimization
 - ◆ Memory access patterns
 - ◆ Global memory optimization
 - ◆ Shared memory optimization
 - ◆ Back to matrix multiplication
- Instruction-level optimization

Asynchronous execution

- Most GPU commands in CUDA are buffered in a queue
- Commands execute asynchronously, in order

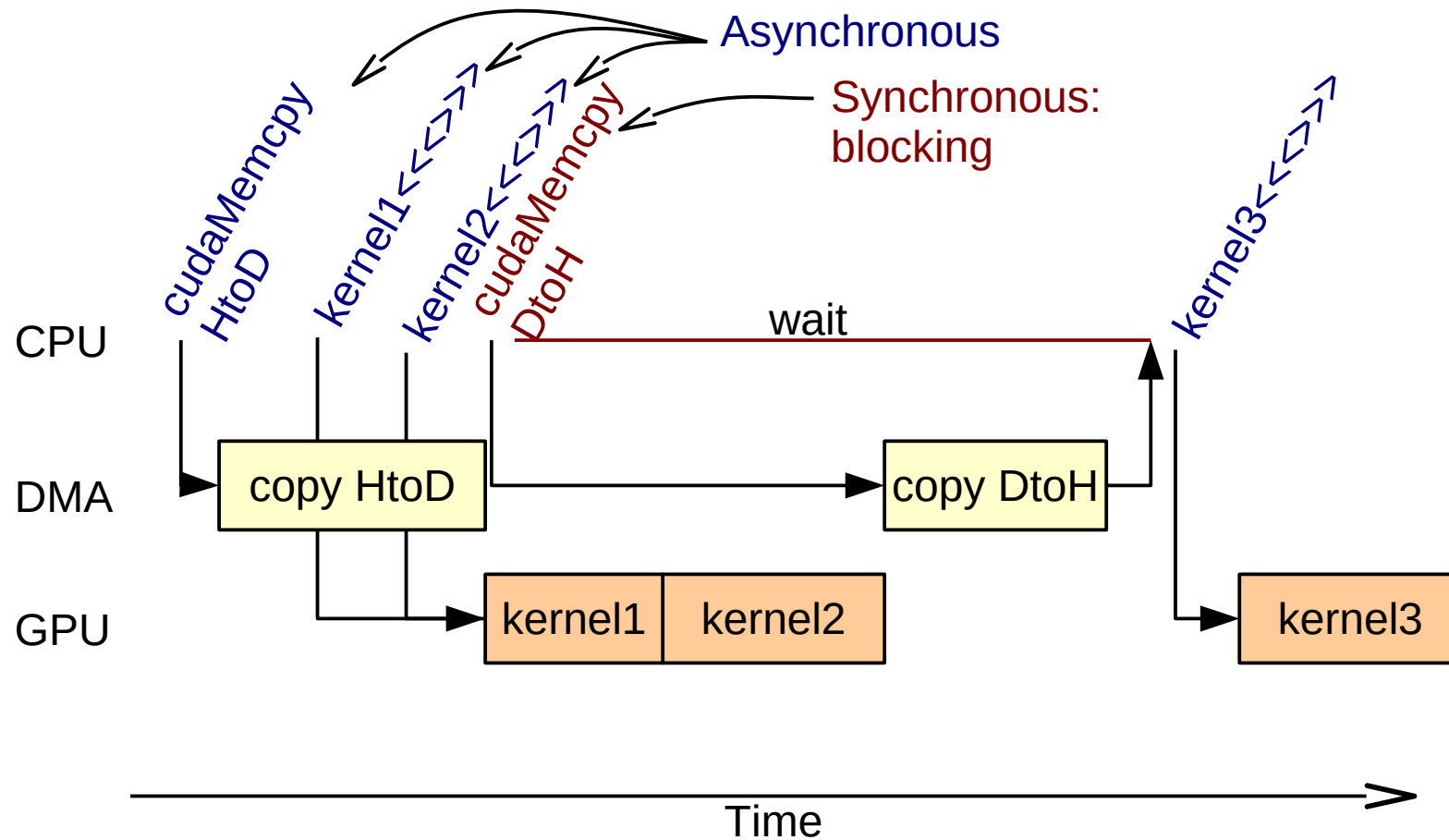


Asynchronous execution

- By default, most CUDA function calls are **asynchronous**
 - ◆ Returns immediately to CPU code
 - ◆ GPU commands are queued and executed in-order
- Some commands are **synchronous** by default
 - ◆ `cudaMemcpy(..., cudaMemcpyDeviceToHost)`
 - ◆ Asynchronous version: `cudaMemcpyAsync`
- Keep it in mind when checking for errors and measuring timing!
 - ◆ Error returned by a command may be caused by an earlier command
 - ◆ Time taken by `kernel<<<>>>` launch is meaningless
- To force synchronization: `cudaDeviceSynchronize()`

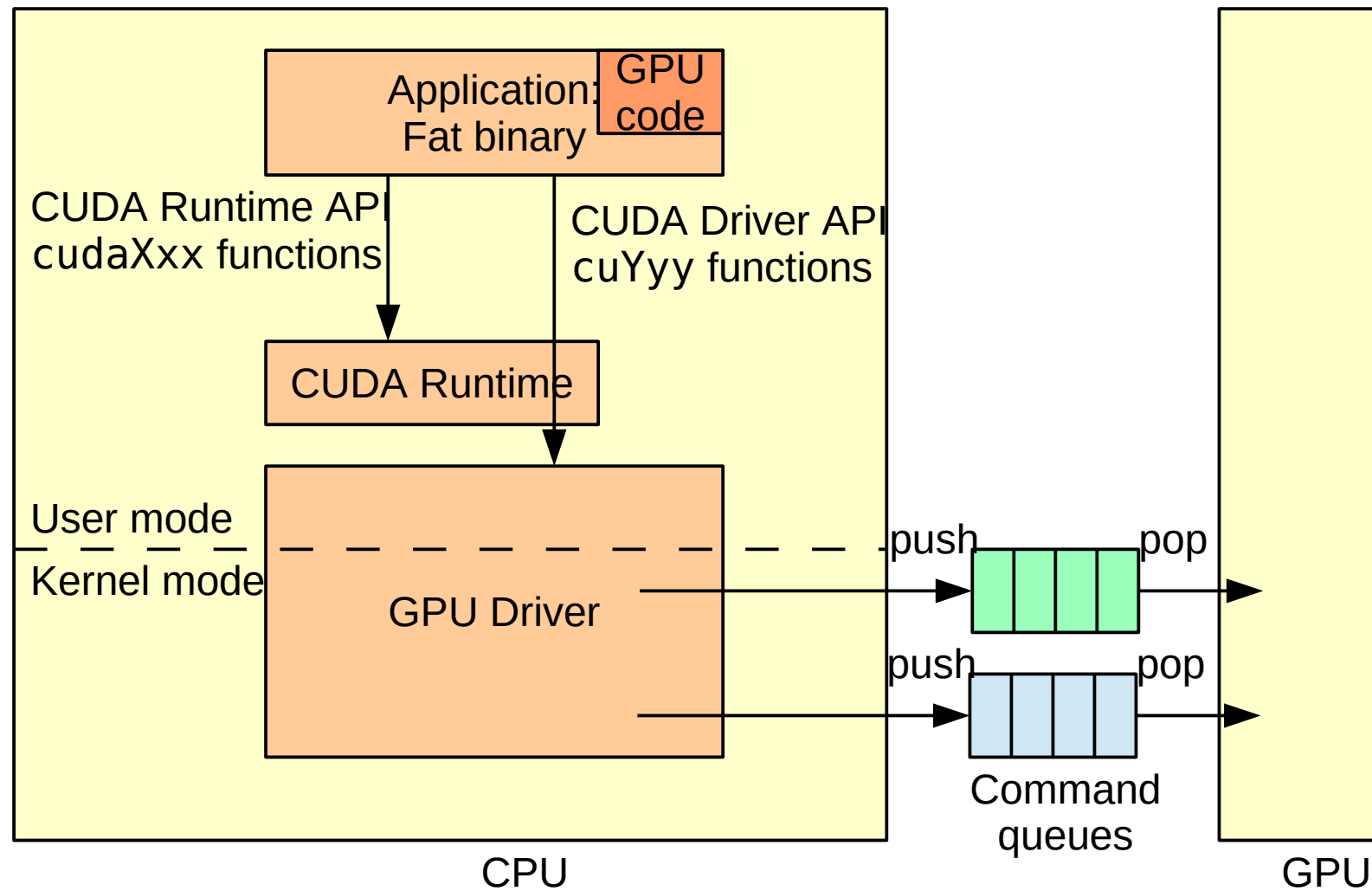
Asynchronous transfers

- Overlap CPU work with GPU work



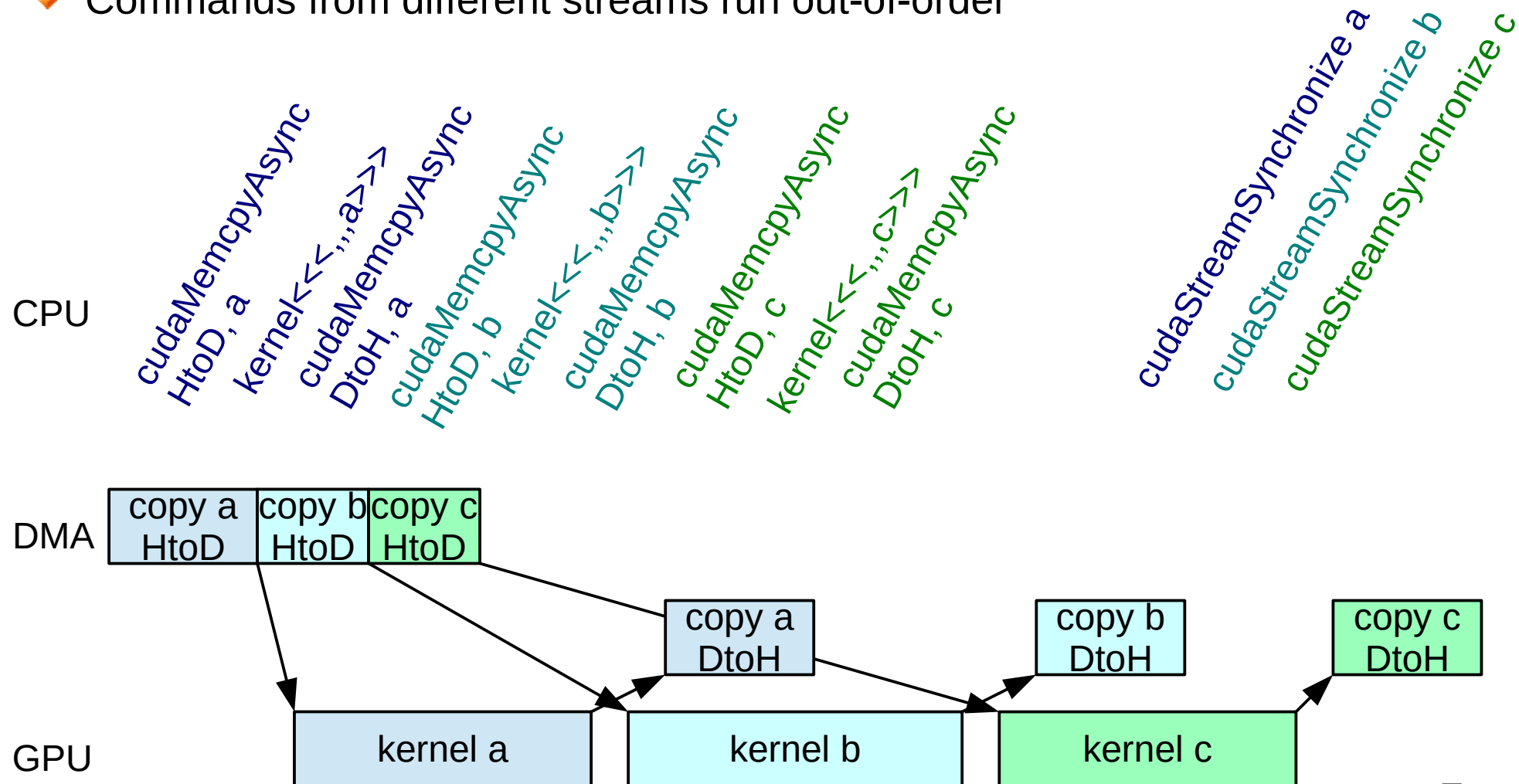
- Can we do better?

Multiple command queues / streams



Streams: pipelining commands

- *Command queues* in OpenCL
 - ◆ Commands from the same stream run in-order
 - ◆ Commands from different streams run out-of-order



Streams: benefits

- Overlap CPU-GPU communication and computation:
Direct Memory Access (DMA) copy engine
runs CPU-GPU memory transfers in background
 - ✦ Requires page-locked memory
 - ✦ Some Tesla GPUs have 2 DMA engines:
simultaneous send and receive
- Concurrent kernel execution
 - ✦ Start next kernel before previous kernel finishes
 - ✦ Mitigates impact of load imbalance / tail effect

Example

Kernel<<<5,,,a>>>

Kernel<<<4,,,b>>>

Serial kernel execution

a block 0	a 3	b 0	b 3
a 1	a 4	b 1	
a 2		b 2	

Concurrent kernel execution

a block 0	a 3	b 2	
a 1	a 4	b 1	
a 2	b 0	b 3	

Page-locked memory

- By default, allocated memory is *pageable*
 - ✦ Can be swapped out to disk, moved by the OS...
- DMA transfers are only safe on *page-locked* memory
 - ✦ Fixed virtual → physical mapping
 - ✦ `cudaMemcpy` needs an intermediate copy:
slower, **synchronous only**
- `cudaMallocHost` allocates page-locked memory
 - ✦ Mandatory when using streams
- Warning: page-locked memory is a limited resource!

Streams: example

- Send data, execute, receive data

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);

float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);

for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);

    MyKernel <<<100, 512, 0, stream[i]>>>
        (outputDevPtr + i * size, inputDevPtr + i * size, size);

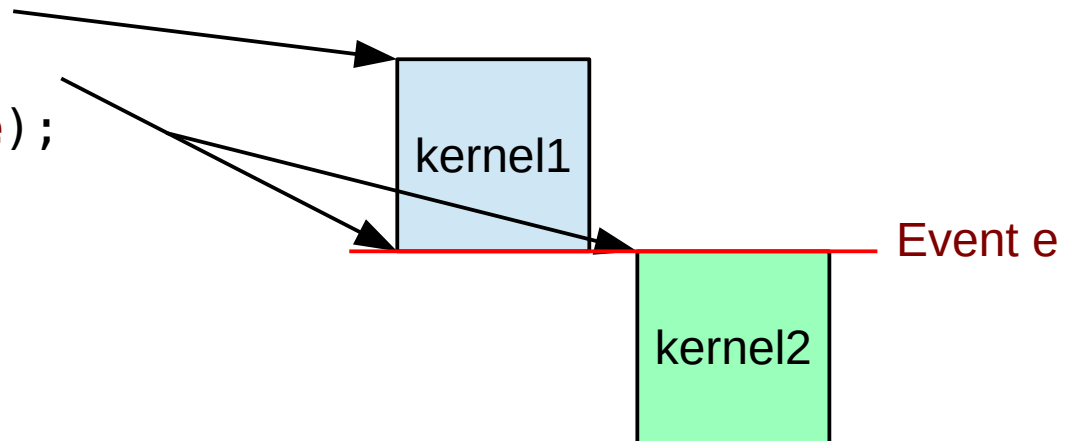
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
}

for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

Events: synchronizing streams

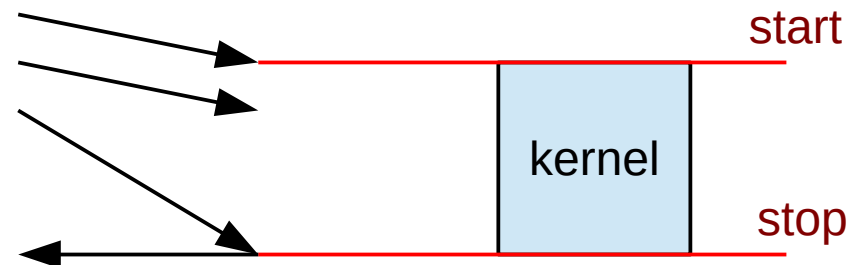
- Schedule synchronization of one stream with another
 - ◆ Specify dependencies between tasks

```
cudaEvent_t e;  
cudaEventCreate(&e);  
kernel1<<<,,a>>>();  
cudaEventRecord(e, a);  
cudaStreamWaitEvent(b, e);  
kernel2<<<,,b>>>();
```



- Measure timing

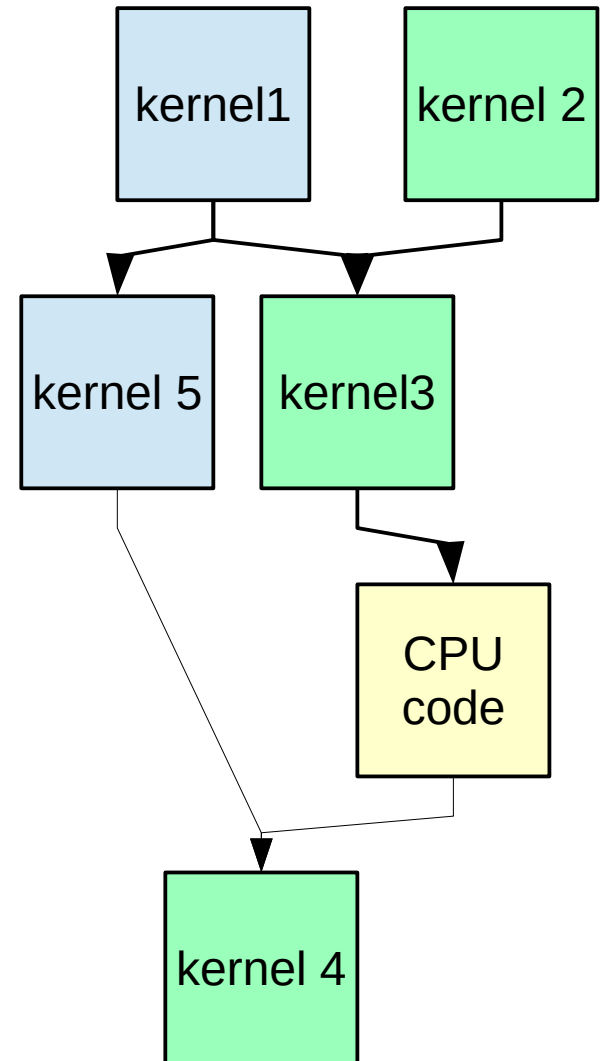
```
cudaEventRecord(start, 0);  
kernel<<<>>>();  
cudaEventRecord(stop, 0);  
cudaEventSynchronize(stop);
```



```
float elapsedTime;  
cudaEventElapsedTime(&elapsedTime, start, stop);
```

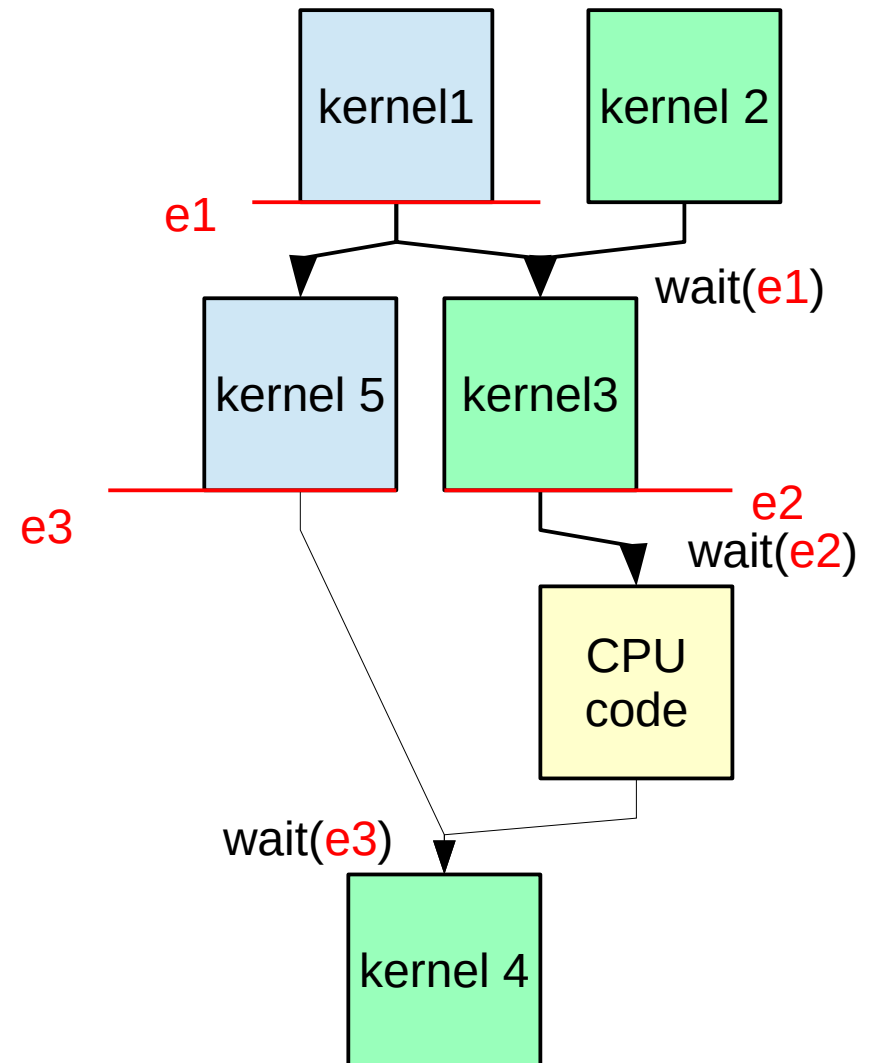
Scheduling data dependency graphs

- With streams and events, we can express task dependency graphs
 - ♦ Equivalent to threads and events (e.g. semaphores) on CPU
- Example:
 - ♦ 2 GPU streams: a b
 - and 1 CPU thread:
 - ♦ Where should we place events?




Scheduling data dependency graphs

```
kernel1<<<,,,a>>>();  
cudaEventRecord(e1, a);  
  
kernel2<<<,,,b>>>();  
  
cudaStreamWaitEvent(b, e1);  
kernel3<<<,,,b>>>();  
cudaEventRecord(e2, b);  
  
kernel5<<<,,,a>>>();  
cudaEventRecord(e3, a);  
  
cudaEventSynchronize(e2);  
  
CPU code  
  
cudaStreamWaitEvent(b, e3);  
kernel4<<<,,,b>>>();
```



NVIDIA Compute capabilities

- Newer GPUs introduce additional features

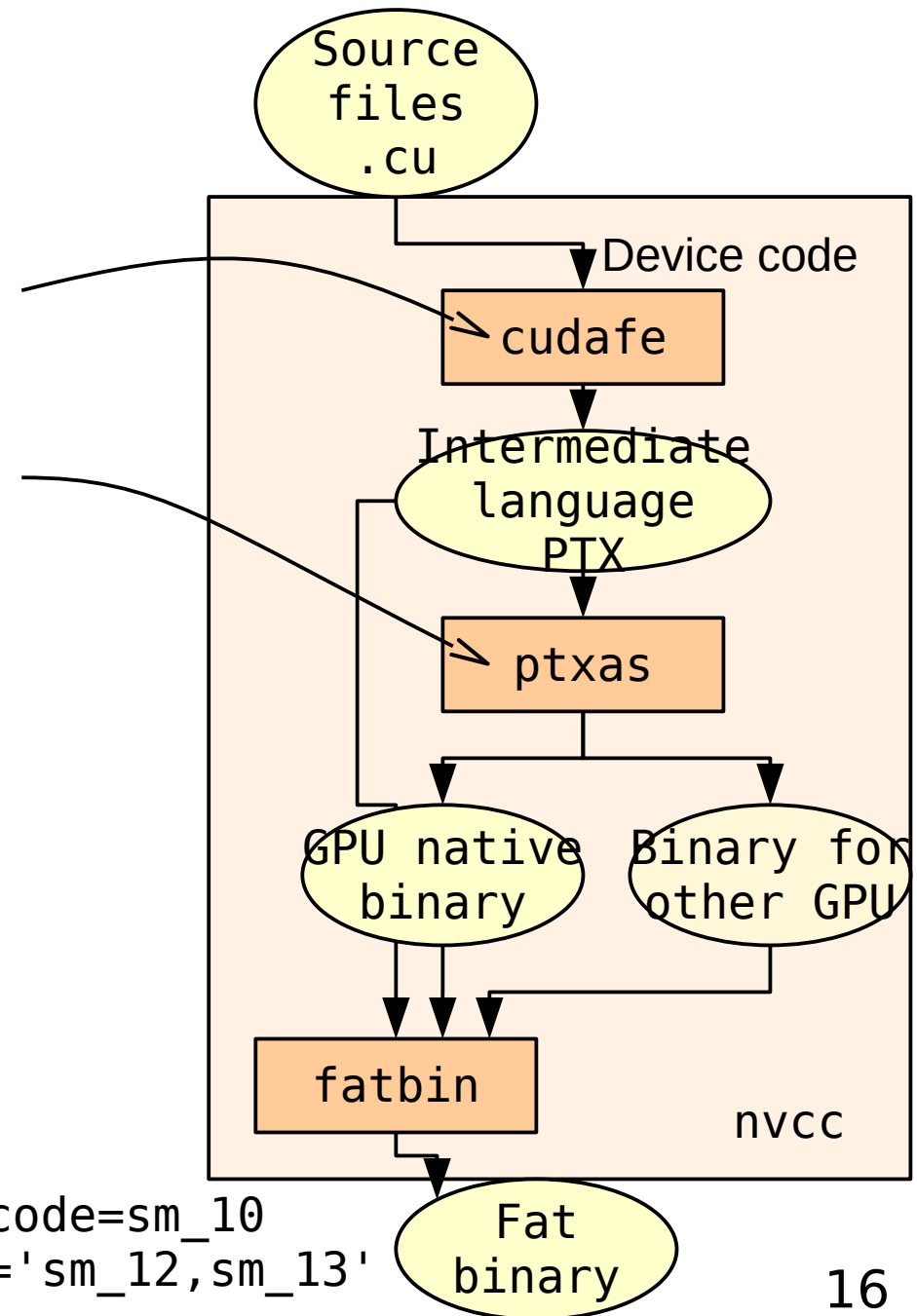
GPU	G80	G92	GT200	GT21x	GF100	GF104	GK104	GK110	GM107	GM204	GP100	GP102	GV100	TU102
Compute capability	1.0	1.1	1.3	1.2	2.0	2.1	3.0	3.5	5.0	5.2	6.0	6.1	7.0	7.5
														2020

- Compute capability means both
 - ◆ Set of features supported
Who can do more can do less: $x > y \rightarrow \text{CC } x \text{ includes CC } y$
 - ◆ Native instruction set
Not always backward-compatible
e.g. GPU of CC 6.0 cannot run binary for CC 5.2

Compiler targets

- Compiler flags: `--generate-code arch=<arch>, code=<code>, ...`
 - ✦ `arch=CC`: directs PTX generation
my code requires features of CC
 - ✦ `code=CC`: directs native code gen.
generate code for GPU CC
 - ✦ Multiple targets are possible
- CC can be
 - ✦ `compute_xx` for PTX
 - ✦ `sm_xx` for native
- Example

```
nvcc -generate-code arch=compute_10,code=sm_10  
-generate-code arch=compute_11,code='sm_12,sm_13'  
-o hello hello.cu
```

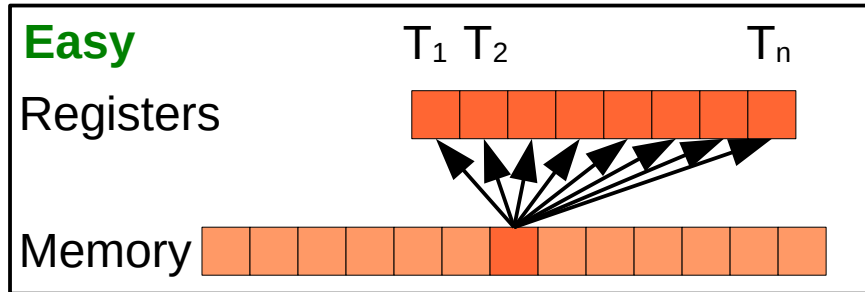


Outline

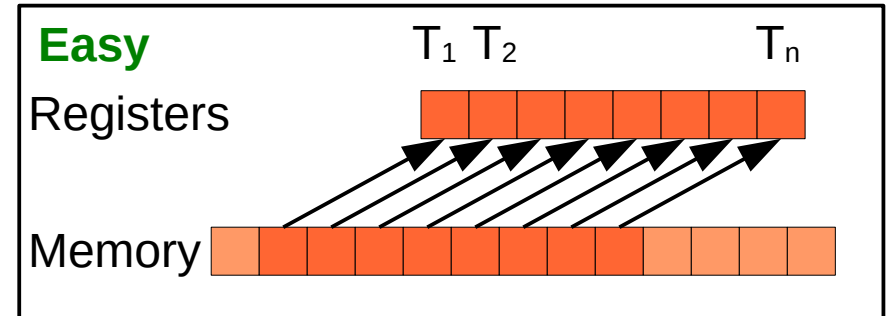
- Host-side task and memory management
 - ◆ Asynchronism and streams
 - ◆ Compute capabilities
- Work partitioning and memory optimization
 - ◆ Memory access patterns
 - ◆ Global memory optimization
 - ◆ Shared memory optimization
 - ◆ Back to matrix multiplication
- Instruction-level optimization

Memory access patterns

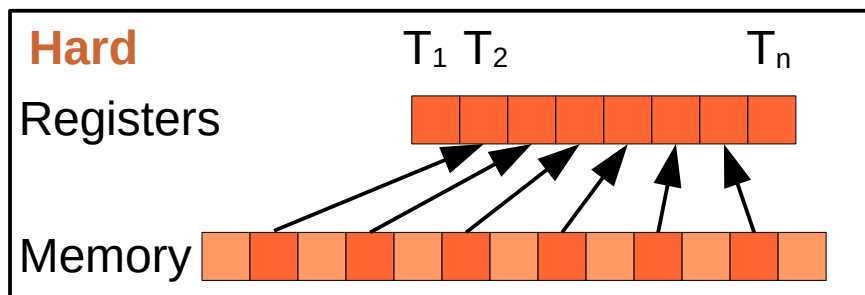
In traditional vector processing



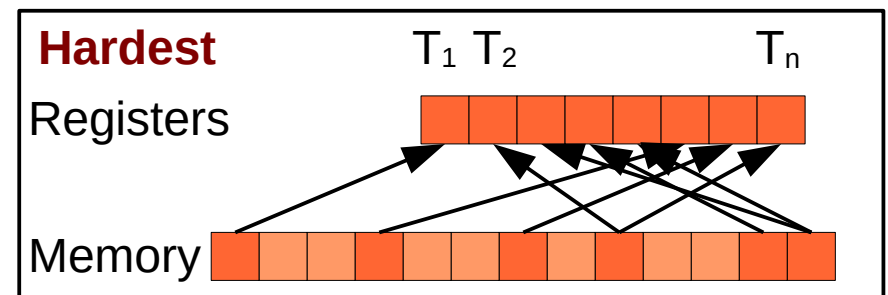
Scalar load & broadcast
Reduction & scalar store



Unit-strided load
Unit-strided store



(Non-unit) strided load
(Non-unit) strided store



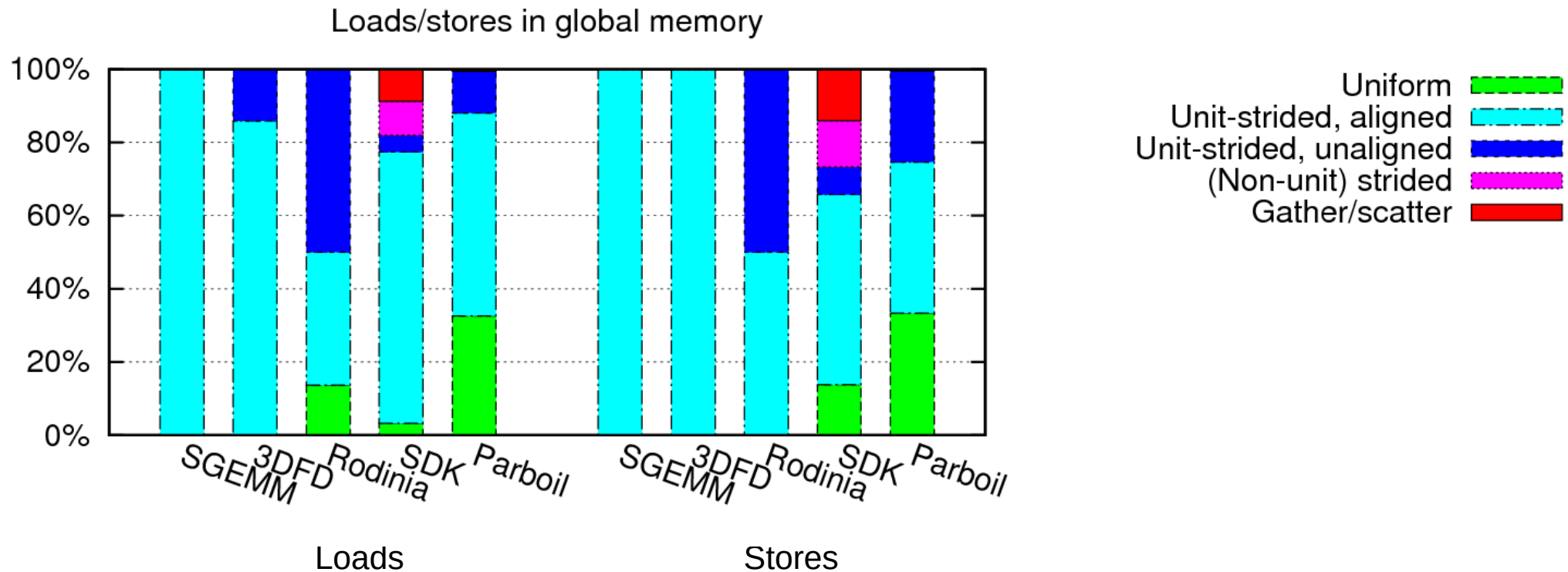
Gather
Scatter

On GPUs

- Every load is a gather, every store is a scatter

Breakdown of memory access patterns

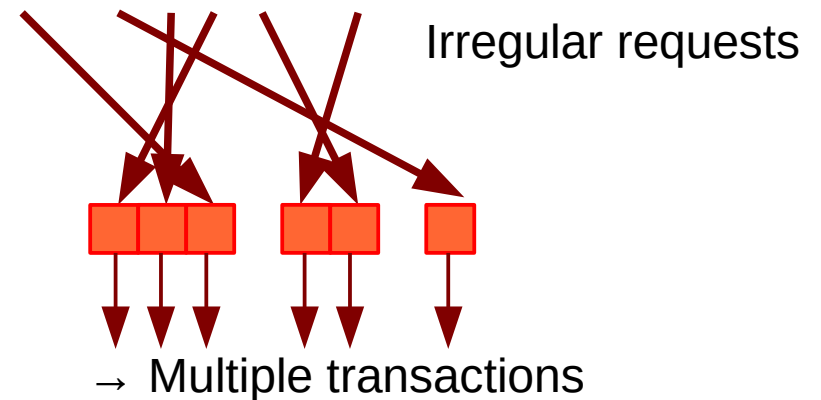
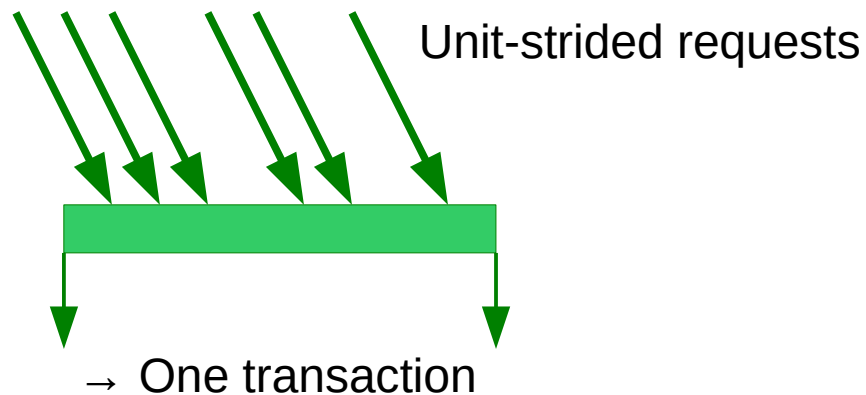
- Vast majority: uniform or unit-strided
 - ◆ And even aligned vectors



- Hardware is optimized for the common case

Memory coalescing

- In hardware: compare the address of each vector element
- Coalesce memory accesses that fall within the same segment

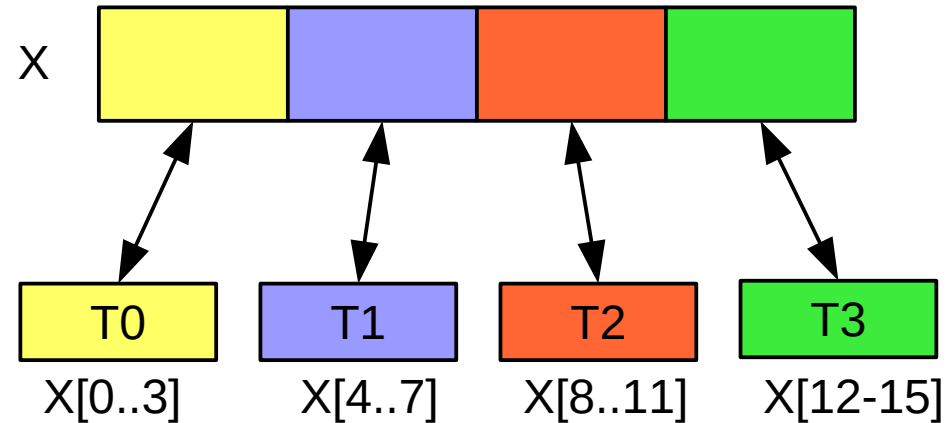


- Dynamically detects parallel memory regularity

Consequences: threading granularity

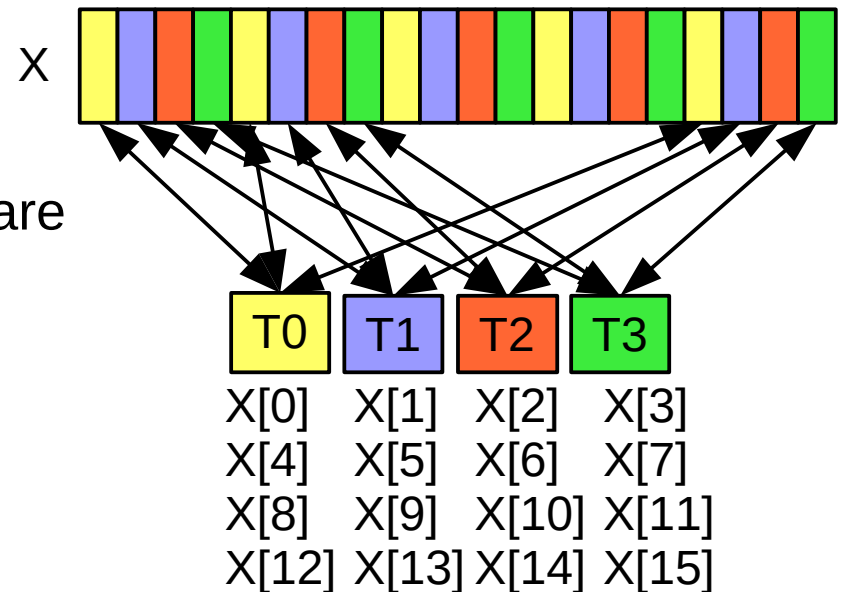
- Coarse-grained threading

- ➔ **Decouple** tasks to reduce **conflicts** and inter-thread communication
- e.g. MPI, OpenMP



- Fine-grained threading

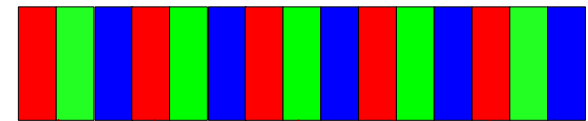
- ➔ **Interleave** tasks
- ➔ Exhibit **locality**: neighbor threads share memory
- ➔ Exhibit **regularity**: neighbor threads have a similar behavior
- e.g. CUDA, OpenCL



Array of structures (AoS)

- Programmer-friendly memory layout
 - ◆ Group data logically
- Memory accesses not coalesced
 - ◆ Bad performance on GPU

```
struct Pixel {  
    float r, g, b;  
};  
Pixel image_AoS[480][640];
```



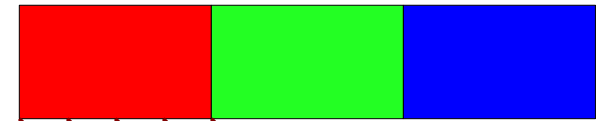
```
kernel void luminance(Pixel img[][],  
    float luma[][]) {  
    int x=tid.x; int y=tid.y;  
    luma[y][x]=.59*img[y][x].r  
        + .11*img[y][x].g  
        + .30*img[y][x].b;  
}
```

- Need to rethink data structures for fine-grained threading

Structure of Arrays (SoA)

- Transpose the data structure
 - ◆ Group together similar data for different threads
- Benefits from memory coalescing
 - ◆ Best performance on GPU

```
struct Image {  
    float R[480][640];  
    float G[480][640];  
    float B[480][640];  
};  
Image image_SoA;
```



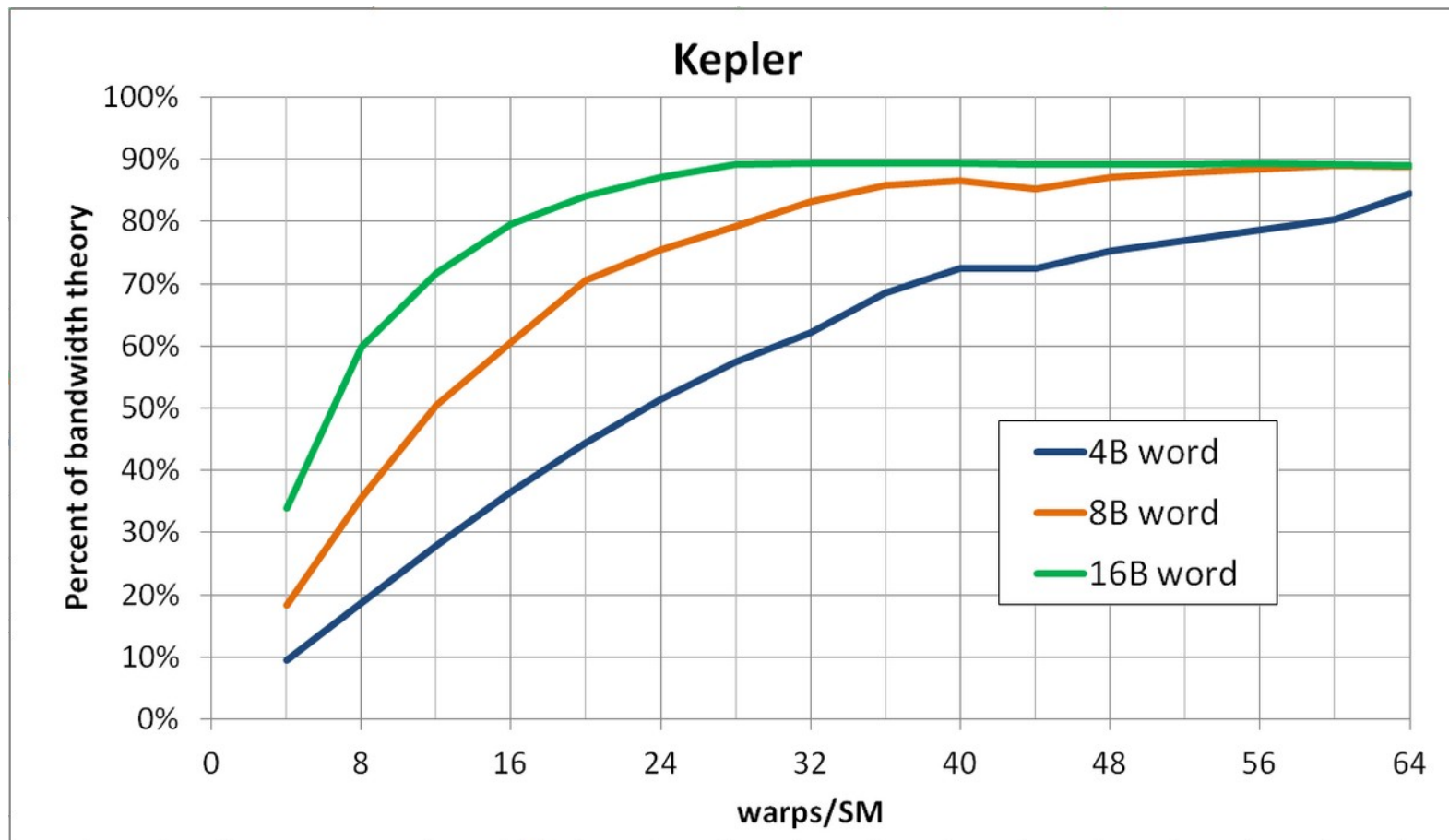
```
kernel void luminance(Image img,  
    float luma[][]) {  
    int x=tid.x; int y=tid.y;  
    luma[y][x]=.59*img.R[y][x]  
        + .11*img.G[y][x]  
        + .30*img.B[y][x];  
}
```

Outline

- Host-side task and memory management
 - ◆ Asynchronism and streams
 - ◆ Compute capabilities
- Work partitioning and memory optimization
 - ◆ Memory access patterns
 - ◆ Global memory optimization
 - ◆ Shared memory optimization
 - ◆ Back to matrix multiplication
- Instruction-level optimization

Vector loads

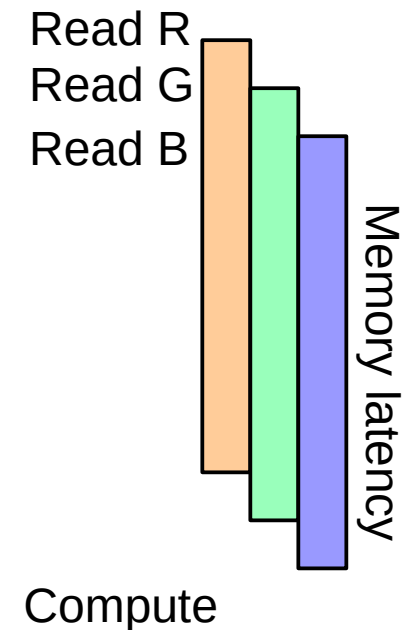
- We can load more data at once with vector types
 - ◆ float2, float4, int2, int4...
 - ◆ More memory parallelism
 - ◆ Allows to reach peak throughput with fewer threads



Multiple outstanding loads

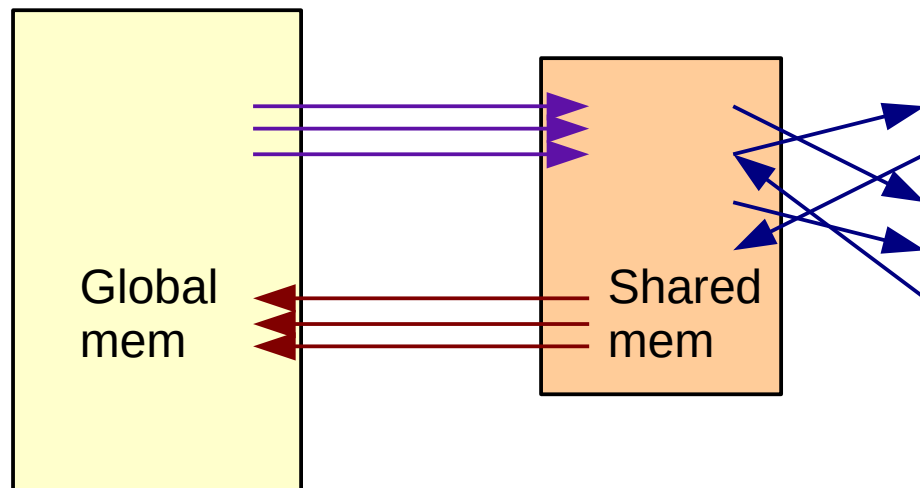
- Multiple independent loads from the same thread can be pipelined
 - ◆ More memory parallelism
 - ◆ Peak throughput with yet fewer threads

```
__global__ void luminance(Image img,  
    float luma[][]) {  
    int x=threadIdx.x, y=threadIdx.y;  
    luma[y][x]=.59*img.R[y][x]  
        + .11*img.G[y][x]  
        + .30*img.B[y][x];  
}
```



Buffer accesses through shared memory

- Global memory accesses are the most expensive
 - ◆ Focus on optimizing global memory accesses
- Strategy: use shared memory as a temporary buffer



1. Load with regular accesses
2. Read and write shared memory with original pattern
3. Store back to global memory with regular accesses

Example: matrix transpose

- $B = A^T$
- Naive algorithms

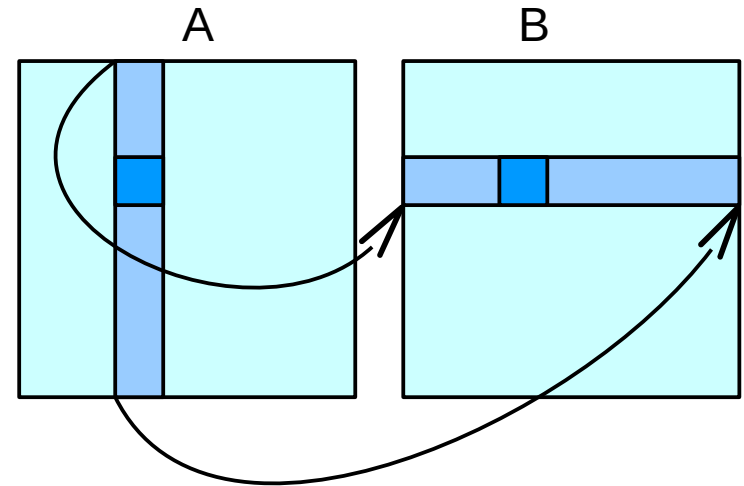
- ◆ Option 1

Thread i, j :
 $B[j, i] = A[i, j]$

- ◆ Option 2

Thread i, j :
 $B[i, j] = A[j, i]$

- Which one is better?
 - ◆ What is the problem?



Example: matrix transpose

- $B = A^T$
- Naive algorithms

- ◆ Option 1

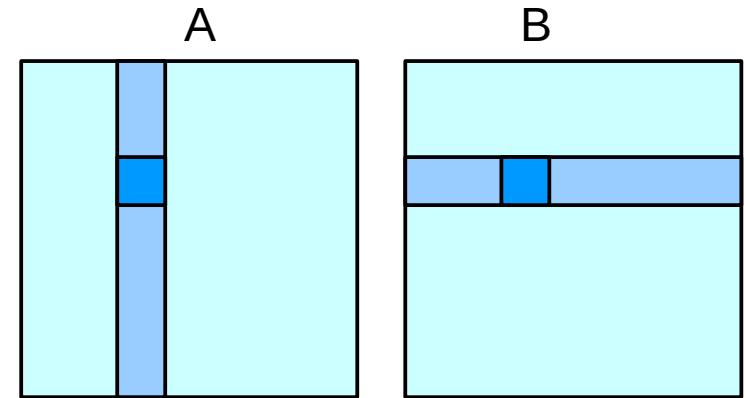
Thread i, j :
 $B[j, i] = A[i, j]$

Coalesced

Non-coalesced

- ◆ Option 2

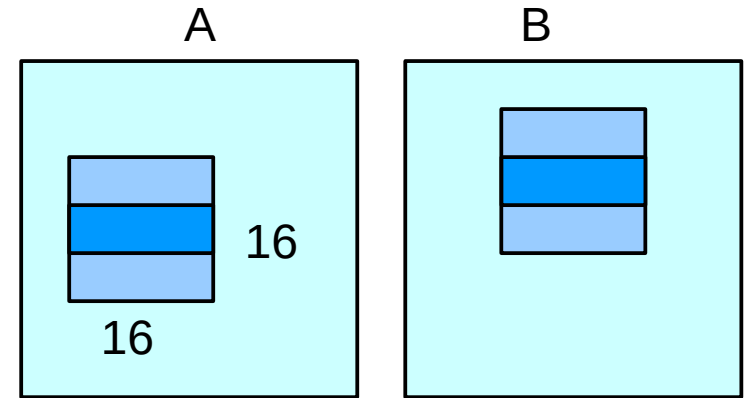
Thread i, j :
 $B[i, j] = A[j, i]$



- Both are equally bad
 - ◆ Access to one array is non-coalesced

Matrix transpose using shared memory

- Split matrices in blocks
- Load the block in shared memory
- Transpose in shared memory
- Write the block back



Example with 16×16 blocks

Block bx, by , Thread tx, ty :

$a[ty, tx] = A[by*16+ty, bx*16+tx]$

Syncthreads

$b[ty, tx] = a[tx, ty]$

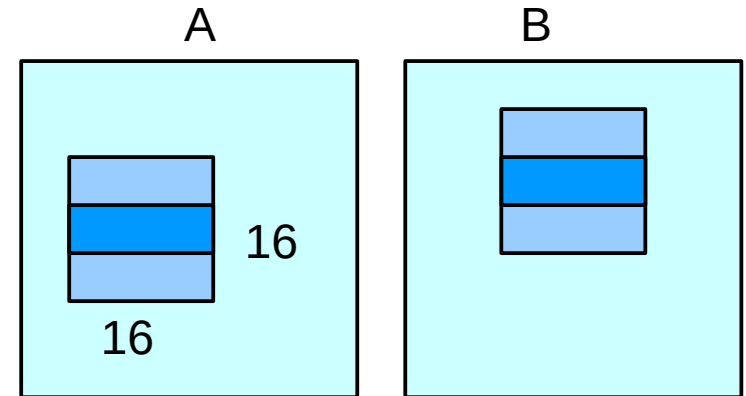
Syncthreads

$B[by*16+ty, bx*16+tx] = b[ty, tx]$

Coalesced

Matrix transpose using shared memory

- Split matrices in blocks
- Load the block in shared memory
- Transpose in shared memory
- Write the block back



Example with 16×16 blocks

Block bx, by , Thread tx, ty :
 $a[ty, tx] = A[by * 16 + ty, bx * 16 + tx]$
Syncthreads

$b_local = a[tx, ty]$

$B[by * 16 + ty, bx * 16 + tx] = b_local$

Same location read and written:
→ local variable, per thread

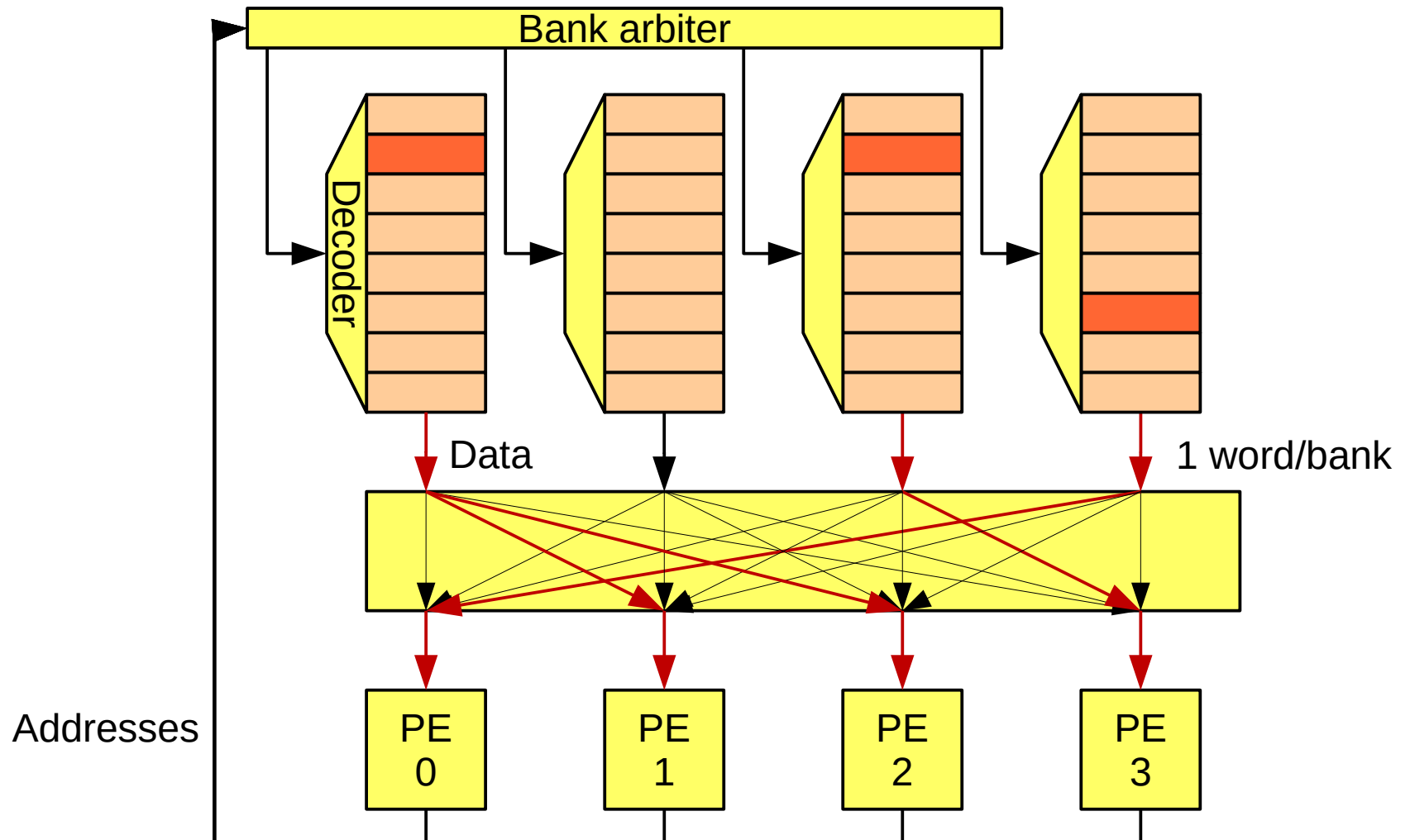
Objection

- Isn't it just moving the problem to shared memory?
- Yes: shared memory has access restrictions too
- But
 - ◆ Shared memory is much faster, even for irregular accesses
 - ◆ We can optimize shared memory access patterns too

Outline

- Host-side task and memory management
 - ◆ Asynchronism and streams
 - ◆ Compute capabilities
- Work partitioning and memory optimization
 - ◆ Memory access patterns
 - ◆ Global memory optimization
 - ◆ Shared memory optimization
 - ◆ Back to matrix multiplication
- Instruction-level optimization

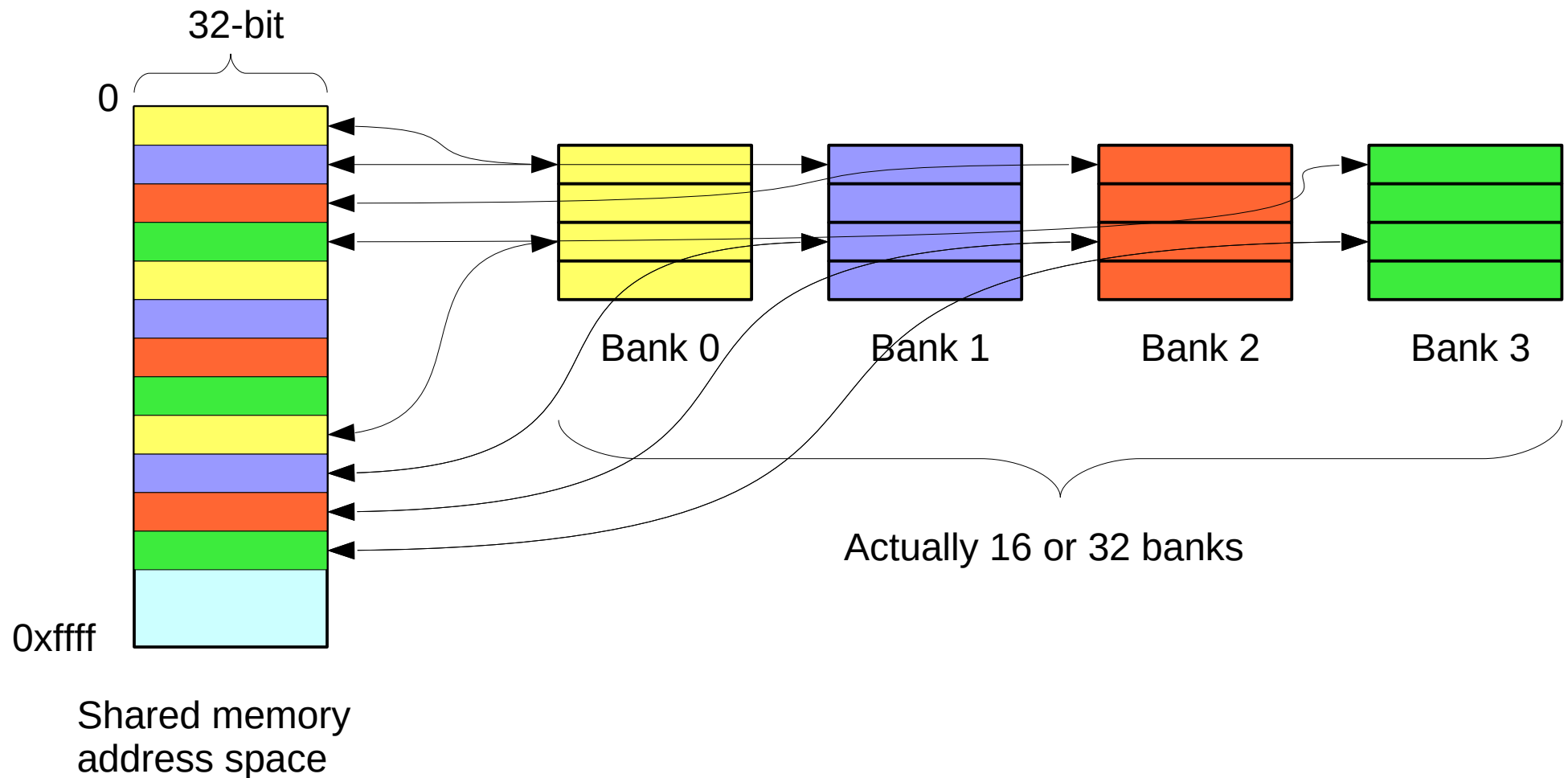
Shared memory: banked



- Inside each SM, shared memory is distributed between multiple banks
 - ◆ 16 or 32 banks

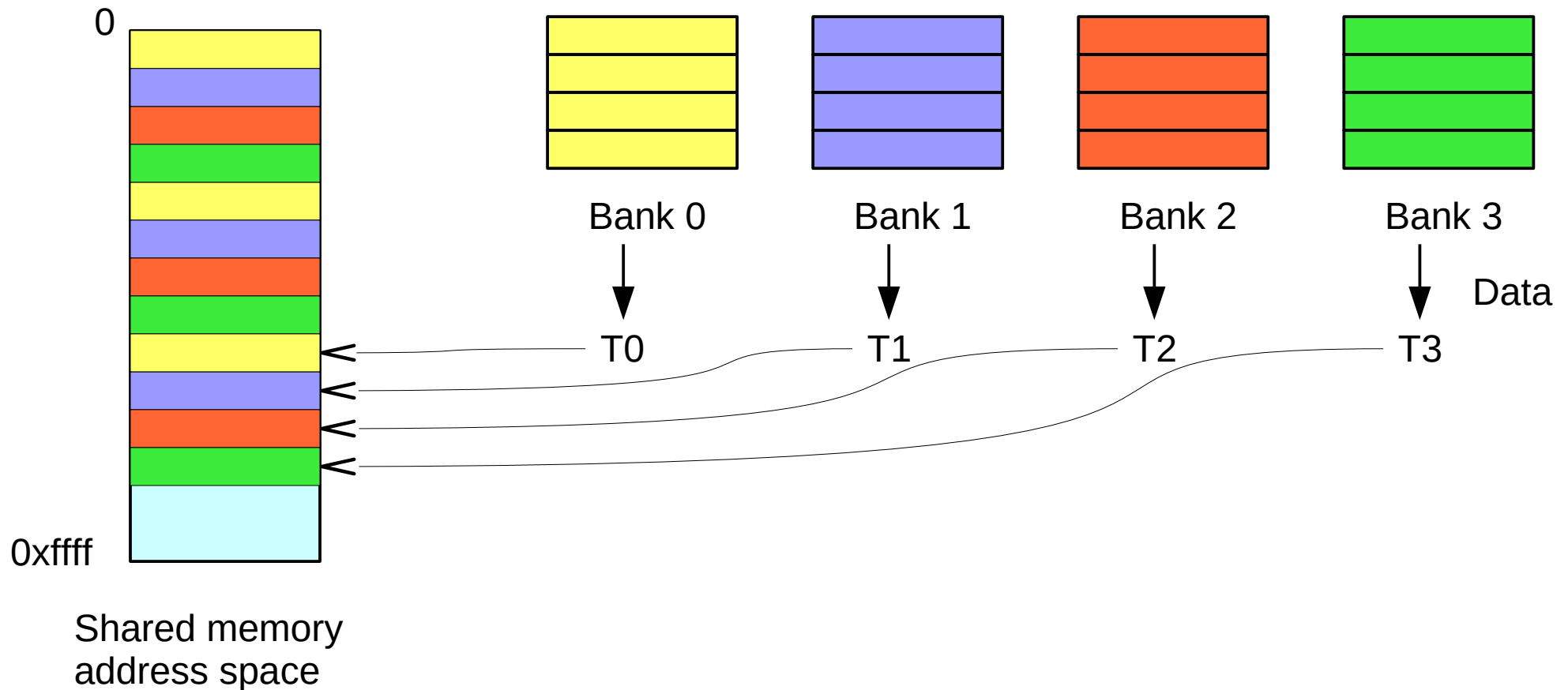
Shared memory bank assignment

- Interleaved on a word-by-word basis:
Modulo placement of data



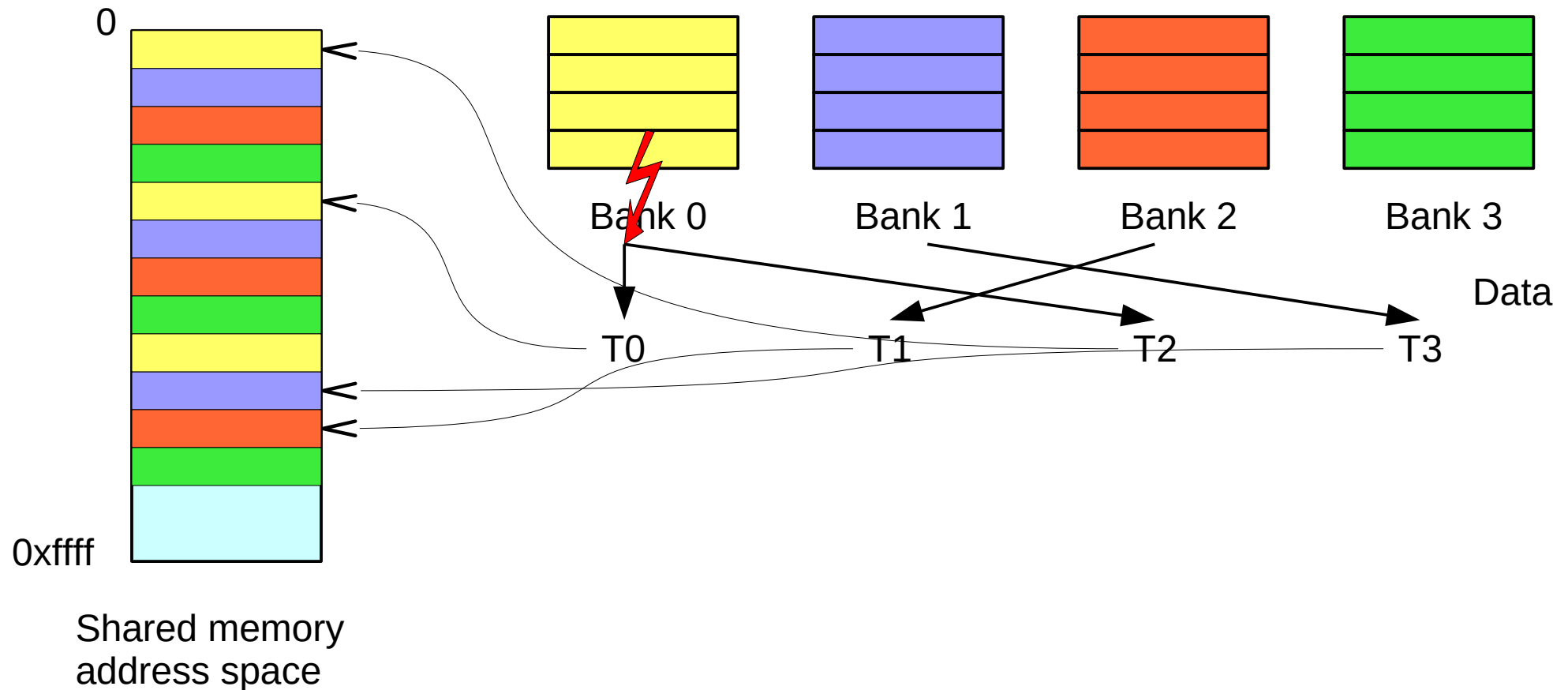
Shared memory: good case

- Threads access contiguous locations: no conflict
 - ◆ All threads can be served concurrently



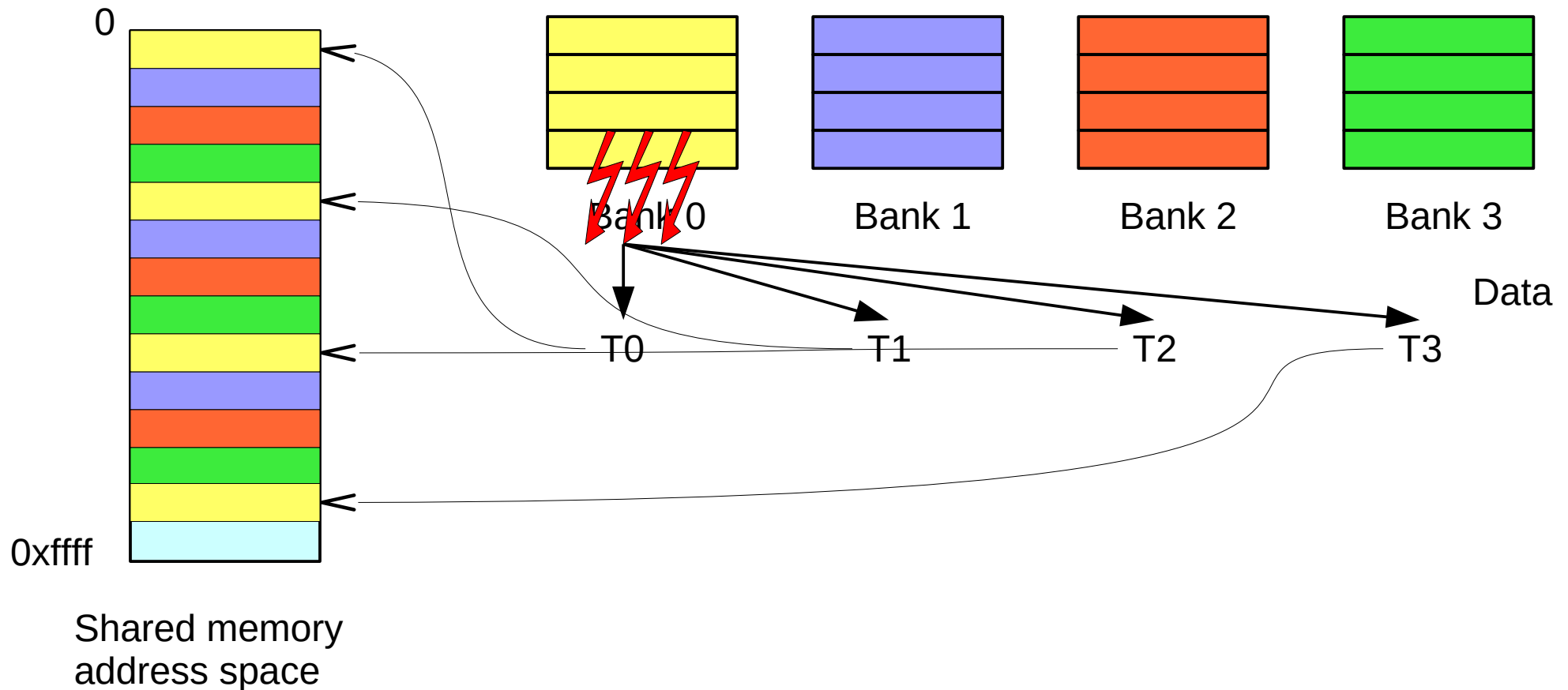
Shared memory: bad case

- Threads access random locations: some conflicts
 - ◆ Some threads have to wait for a bank



Shared memory: worst case

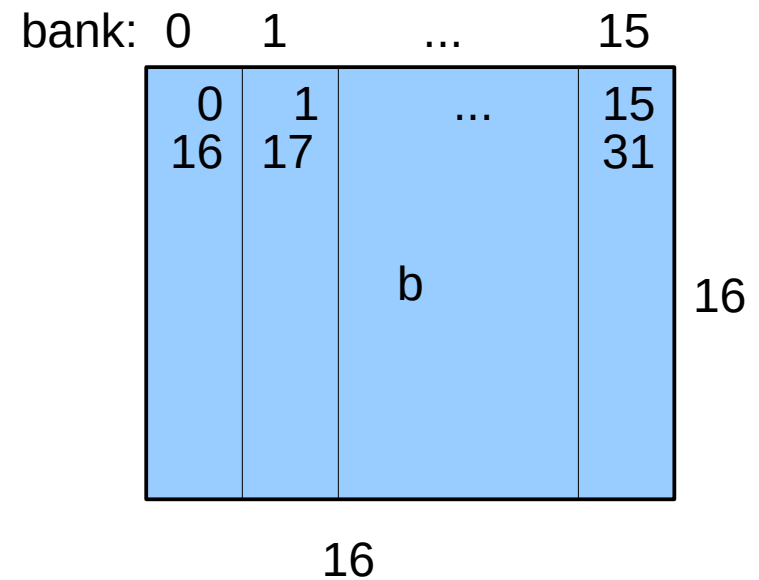
- Threads access locations spaced by 16: systematic conflict
 - ◆ All threads have to wait for the same bank



Example: matrix transpose

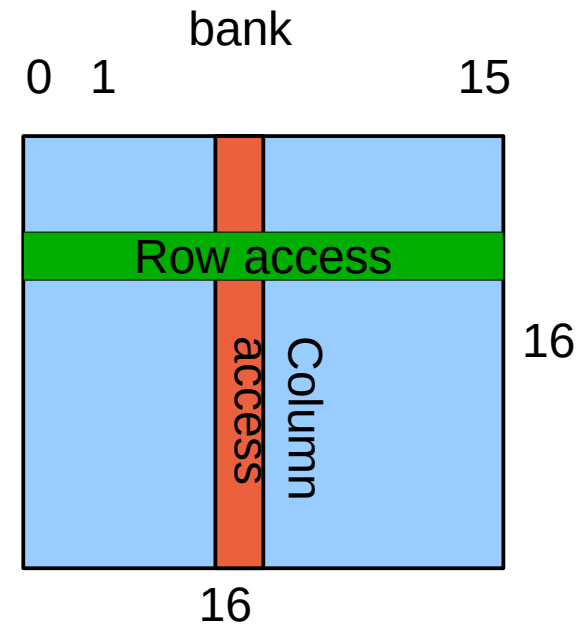
- Where are bank conflicts?

Block b_x, b_y , Thread t_x, t_y :
 $a[t_y * 16 + t_x] = A[b_y * 16 + t_y, b_x * 16 + t_x]$
Syncthreads
 $b[t_y * 16 + t_x] = a[t_x * 16 + t_y]$
Syncthreads
 $B[b_y * 16 + t_y, b_x * 16 + t_x] = b[t_y * 16 + t_x]$



Example: matrix transpose

- Where are bank conflicts?



Block b_x, b_y , Thread t_x, t_y :

$a[t_y * 16 + t_x] = A[b_y * 16 + t_y, b_x * 16 + t_x]$

Syncthreads

$b[t_y * 16 + t_x] = a[t_x * 16 + t_y]$

Syncthreads

$B[b_y * 16 + t_y, b_x * 16 + t_x] = b[t_y * 16 + t_x]$

Column access: systematic conflicts

- How to avoid them?

Remapping data

- Solution 1: pad with empty cells

Block b_x, b_y , Thread t_x, t_y :

$a[t_y * 17 + t_x] = A[b_y * 16 + t_y, b_x * 16 + t_x]$

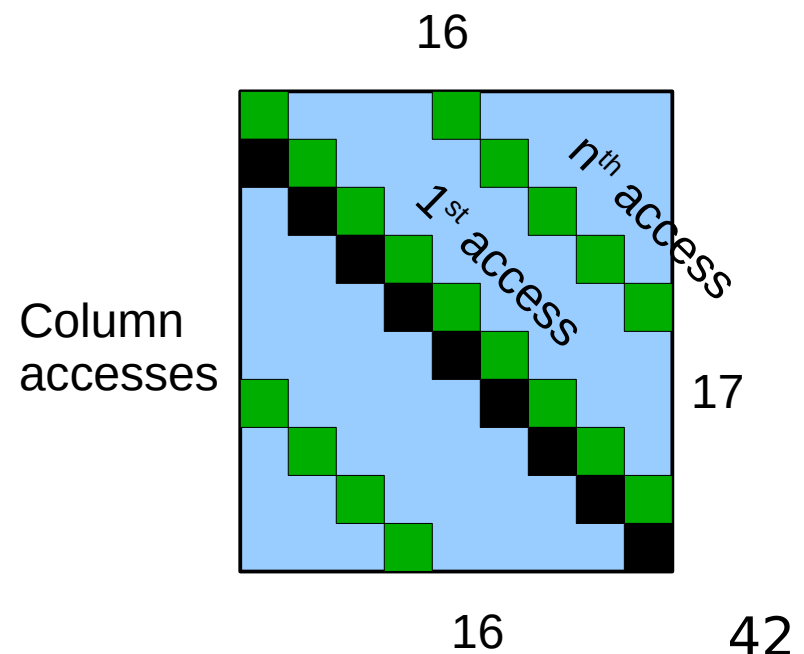
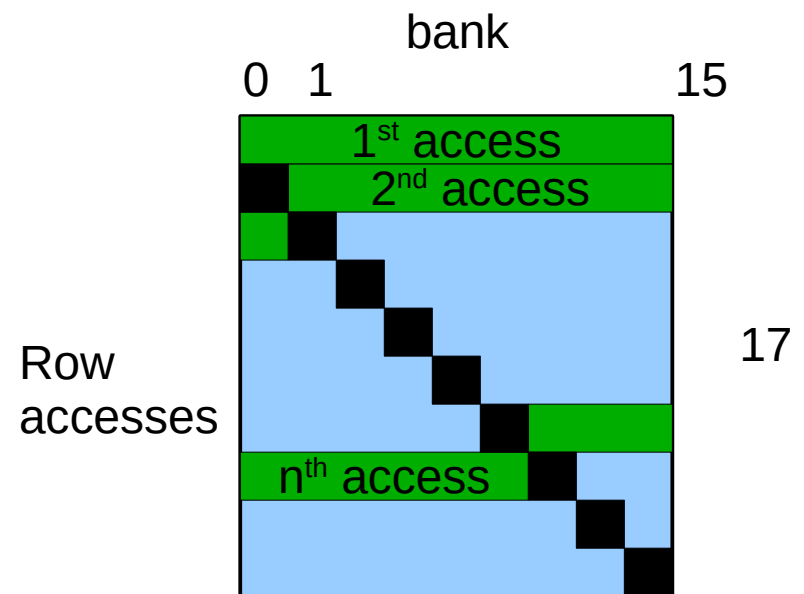
Syncthreads

$b[t_y * 16 + t_x] = a[t_x * 17 + t_y]$

Syncthreads

$B[b_y * 16 + t_y, b_x * 16 + t_x] = b[t_y * 17 + t_x]$

- No bank conflicts
- Memory overhead



Remapping data

- Solution 2: different mapping function
 - ♦ Example: map $[y,x]$ to $y*16+(x+y \bmod 16)$
 - ♦ Or $y*16+(x \wedge y)$

Block b_x, b_y , Thread t_x, t_y :

$a[t_y*16+(t_x+t_y)\%16] = A[b_y*16+t_y, b_x*16+t_x]$

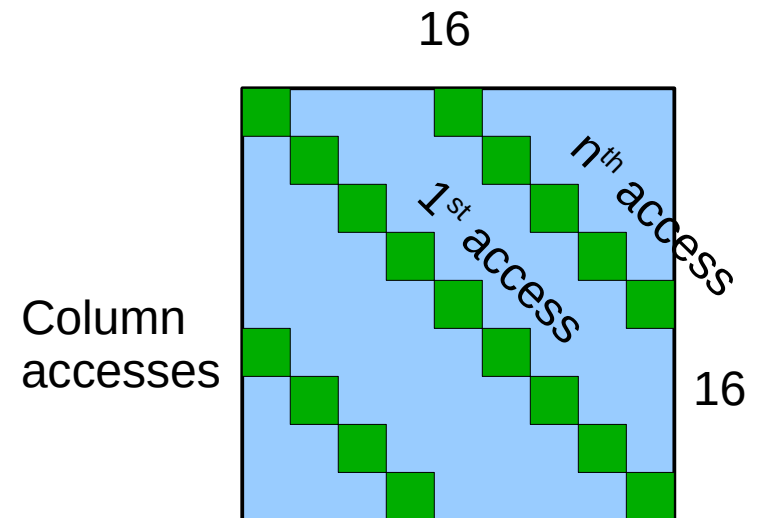
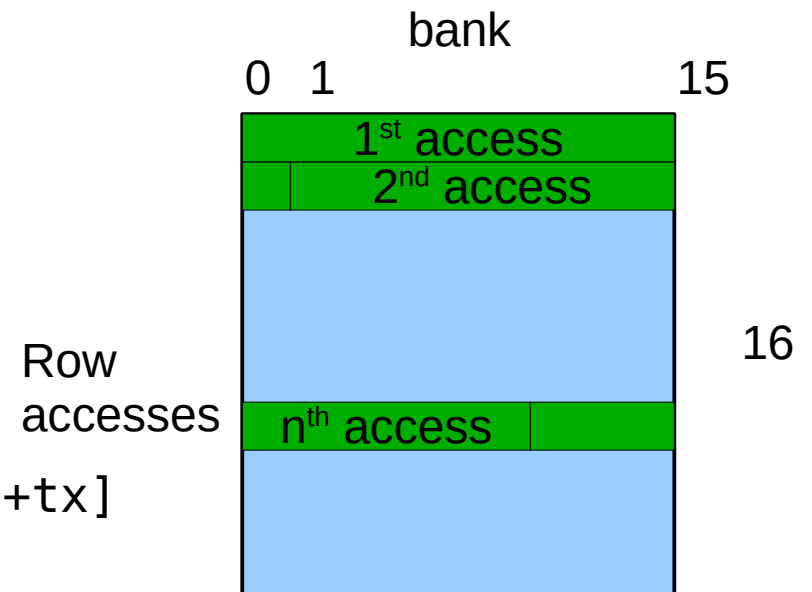
Syncthreads

$b[t_y*16+t_x] = a[t_x*16+(t_y+t_x)\%16]$

Syncthreads

$B[b_y*16+t_y, b_x*16+t_x] = b[t_y*16+t_x]$

- No bank conflicts
- No memory overhead



Recap

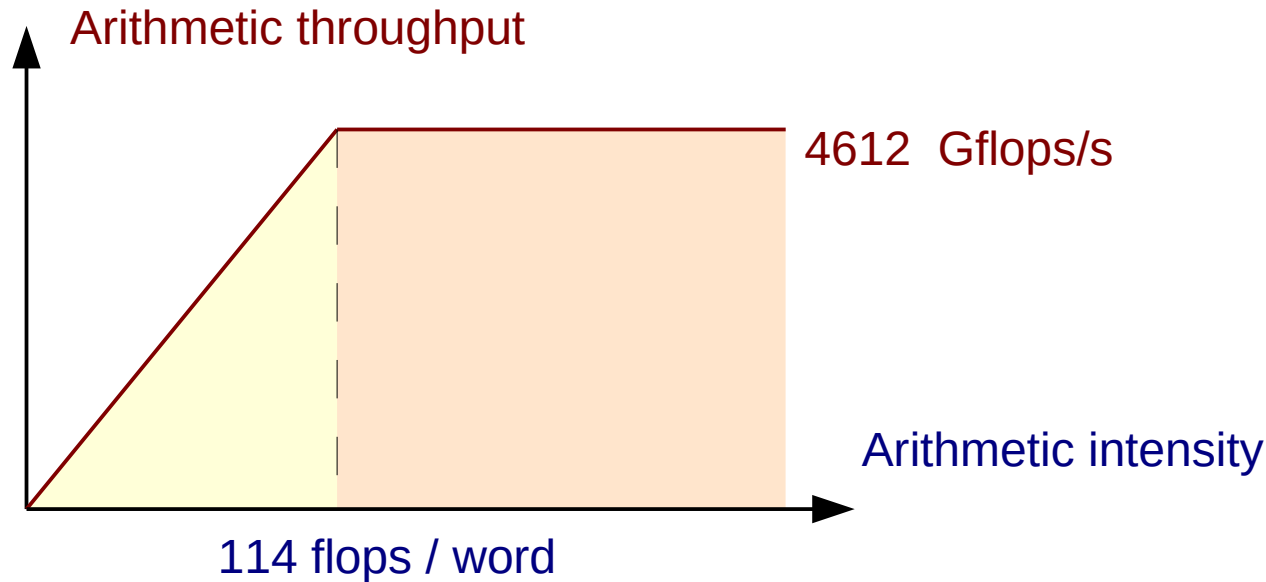
- Overlap long-latency communications with computations
- Avoid global accesses when you can
 - ◆ Reuse data to get enough arithmetic intensity
 - ◆ Use registers and shared memory whenever possible
- Make consecutive threads access contiguous data
 - ◆ Stage data in shared memory if needed
- Avoid bank conflicts in shared memory
- Express locality and regularity

Break

Outline

- Host-side task and memory management
 - ◆ Asynchronism and streams
 - ◆ Compute capabilities
- Work partitioning and memory optimization
 - ◆ Memory access patterns
 - ◆ Global memory optimization
 - ◆ Shared memory optimization
 - ◆ Back to matrix multiplication
- Instruction-level optimization

Arithmetic intensity

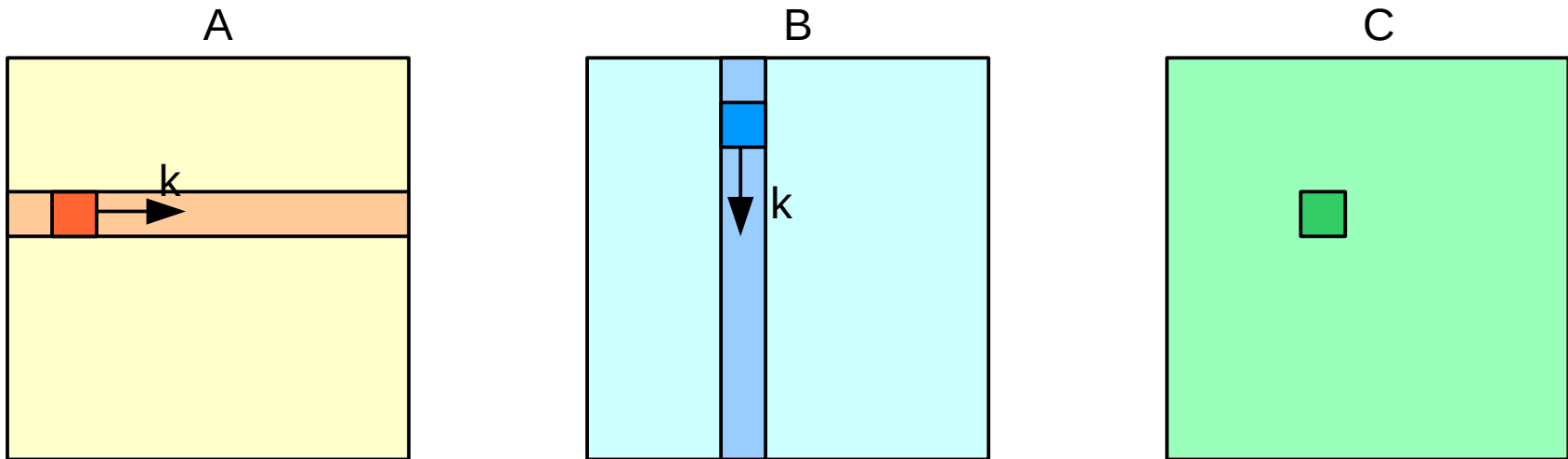


- Example from first lecture
 - ♦ NVIDIA GTX 980 needs ≥ 114 flops / word to reach peak performance
- How to reach enough arithmetic intensity?
 - ♦ Need to **reuse** values loaded from memory

Classic example: matrix multiplication

- Naive algorithm

```
for i = 0 to n-1
  for j = 0 to n-1
    for k = 0 to n-1
      C[i,j] += A[i,k]*B[k,j]
```

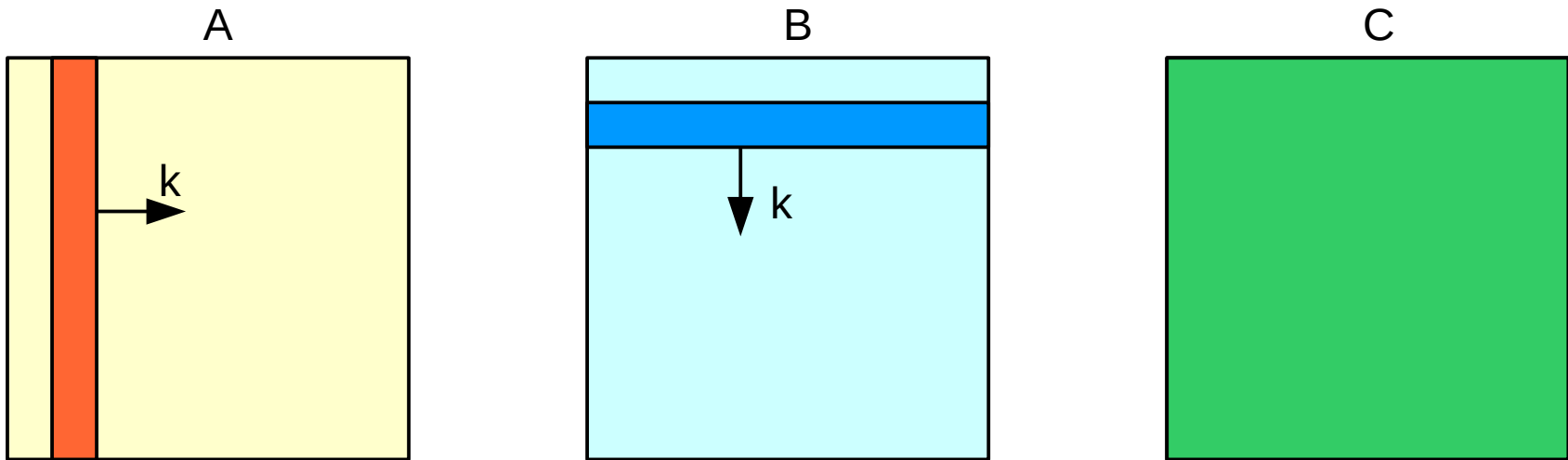


- Arithmetic intensity: 1:1 :(

Reusing inputs

- Move loop on k up

```
for k = 0 to n-1  
  for i = 0 to n-1  
    for j = 0 to n-1  
      C[i,j] += A[i,k]*B[k,j]
```



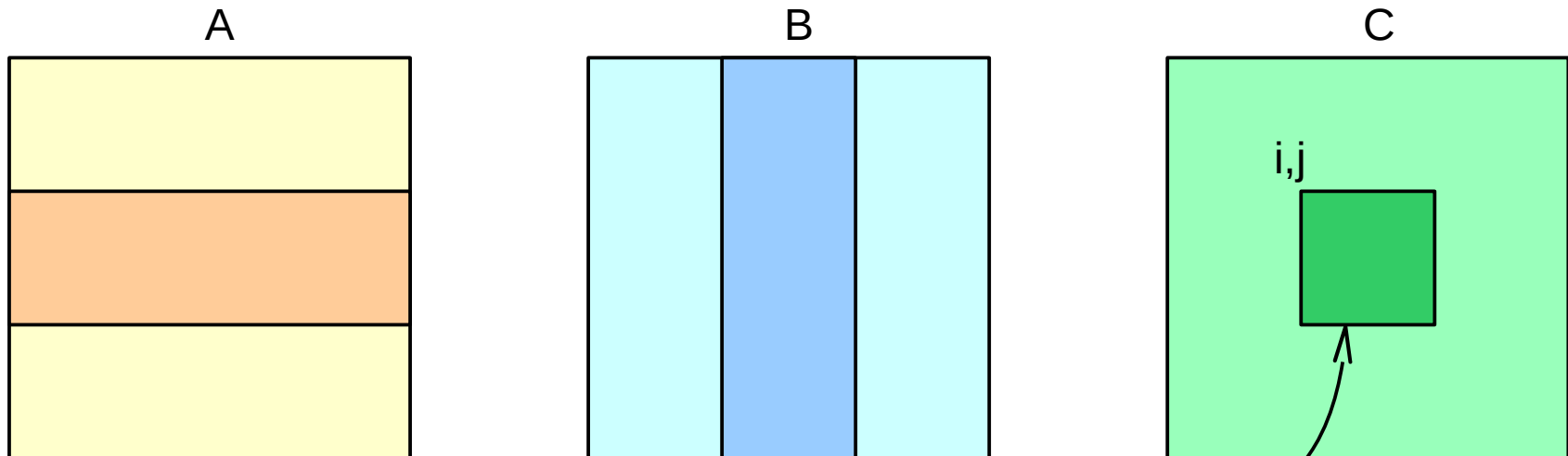
- Enable data reuse on inputs A and B
- But no more reuse on matrix C !

With tiling

- Block loops on i and j

```
for i = 0 to n-1 step 16
  for j = 0 to n-1 step 16
    for k = 0 to n-1
      for i2 = i to i+15
        for j2 = j to j+15
          C[i2,j2] += A[i2,k]*B[k,j2]
```

- For one block: product between horizontal panel of A and vertical panel of B

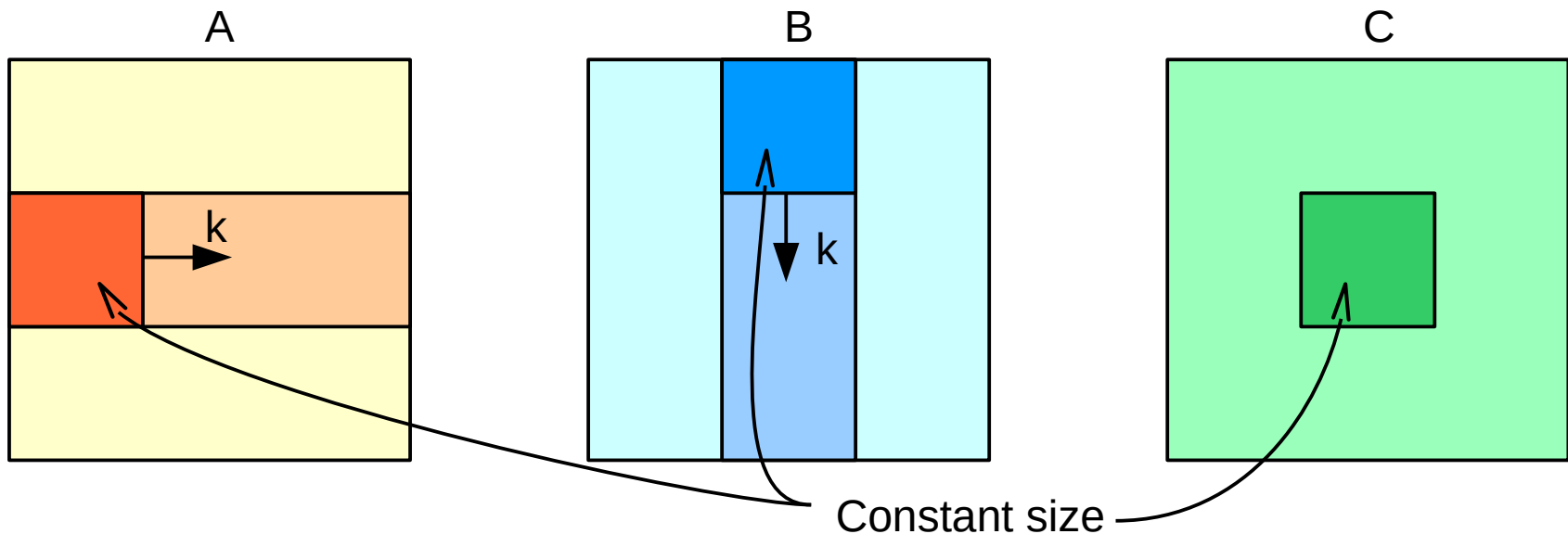


Constant size

With more tiling

- Block loop on k

```
for i = 0 to n-1 step 16
  for j = 0 to n-1 step 16
    for k = 0 to n-1 step 16
      for k2 = k to k+15
        for i2 = i to i+15
          for j2 = j to j+15
            C[i2,j2] += A[i2,k]*B[k,j2]
```



Pre-loading data

```
for i = 0 to n-1 step 16
  for j = 0 to n-1 step 16
    c = {0}
    for k = 0 to n-1 step 16
      a = A[i..i+15,k..k+15]
      b = B[k..k+15,j..j+15]
```

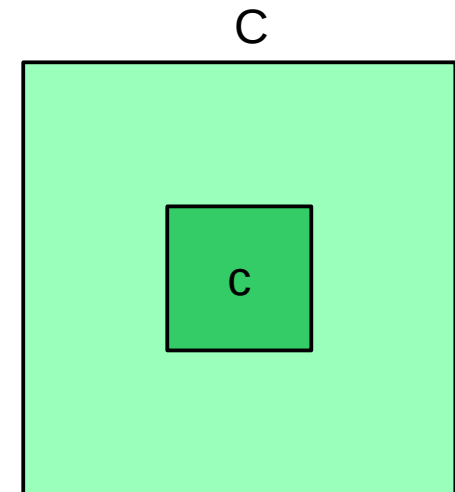
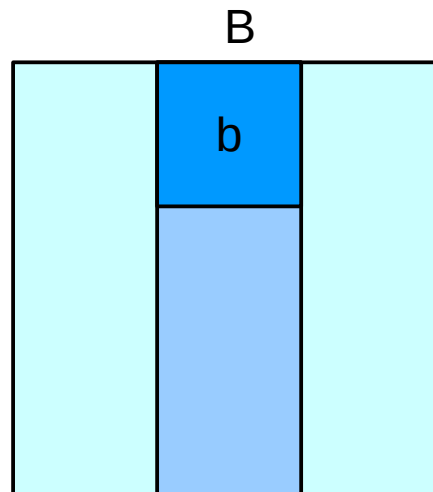
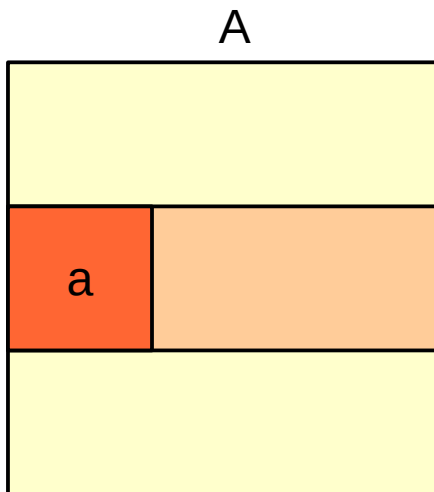
} Load submatrices a and b

```
    for k2 = 0 to 15
      for i2 = 0 to 15
        for j2 = 0 to 15
          c[i2,j2] += a[i2,k2]*b[k2,j2]
```

} Multiply submatrices
 $c = a \times b$

$C[i..i+15,j..j+15] = c$

} Store submatrix c



Arithmetic intensity?

Breaking into two levels

- Run loops on i, j, i2, j2 in parallel

```
for // i = 0 to n-1 step 16
  for // j = 0 to n-1 step 16
```

Level 2:
Blocks

```
    c = {0}
    for k = 0 to n-1 step 16
      a = A[i..i+15,k..k+15]
      b = B[k..k+15,j..j+15]

      for k2 = 0 to 15
        for // i2 = 0 to 15
          for // j2 = 0 to 15
            c[i2,j2] += a[i2,k2]*b[k2,j2]

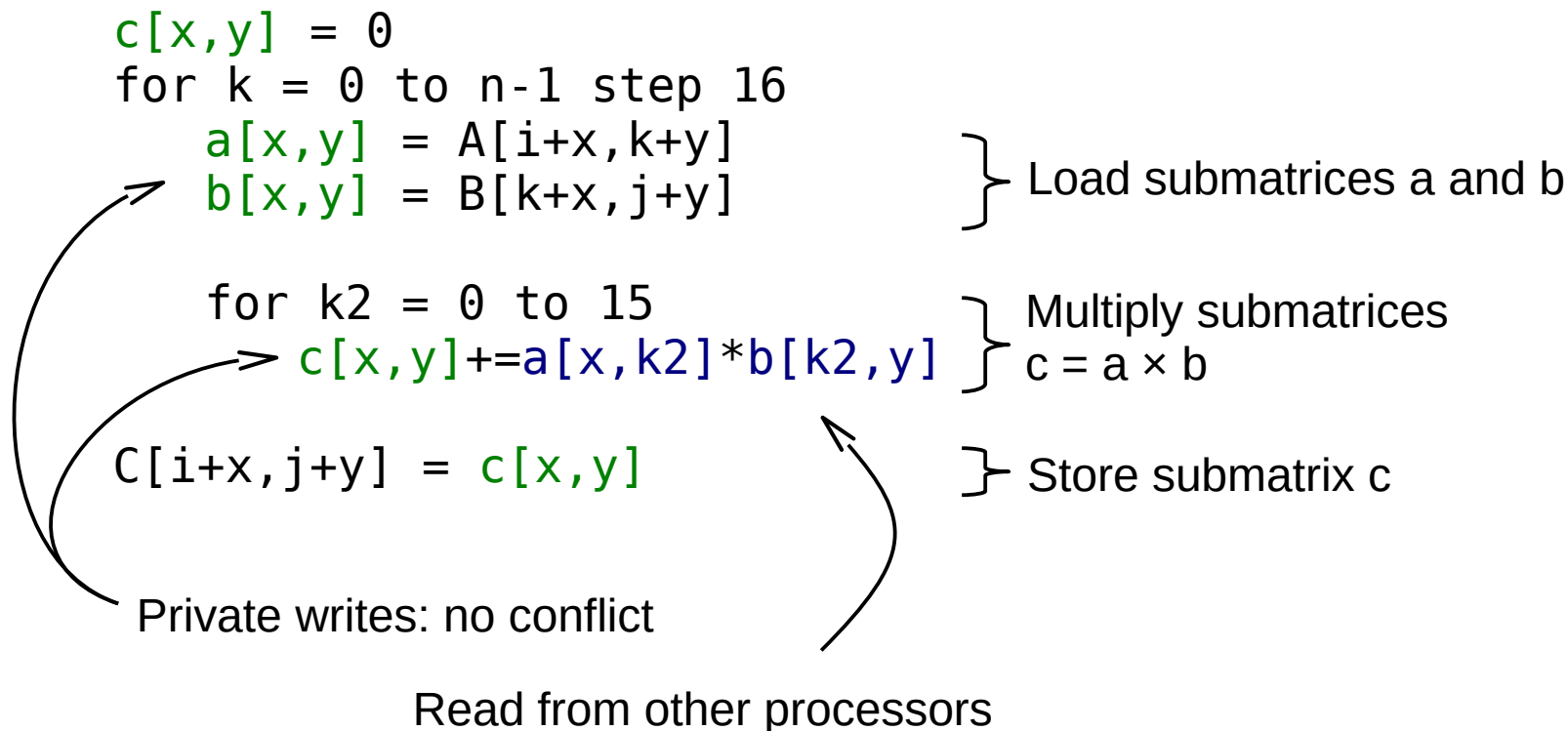
    C[i..i+15,j..j+15] = c
```

Level 1:
Threads

- Let's focus on threads
- 

Level 1: SIMD (PRAM-style) version

- Each processor has ID (x,y)
 - ◆ Loops on i2, j2 are implicit

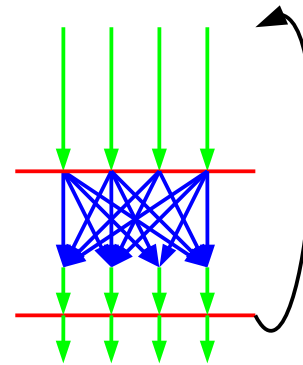


- How to translate to SPMD (BSP-style) ?

SPMD version

- Place synchronization barriers

```
c[x,y] = 0
for k = 0 to n-1 step 16
  a[x,y] = A[i+x,k+y]
  b[x,y] = B[k+x,j+y]
  Barrier
  for k2 = 0 to 15
    c[x,y] += a[x,k2]*b[k2,y]
  Barrier
C[i+x,j+y] = c[x,y]
```



- Why do we need the second barrier ?

Data allocation

- 3 memory spaces: Global, Shared, Local
 - ◆ Where should we put: A, B, C, a, b, c ?

```
c[x,y] = 0
for k = 0 to n-1 step 16
    a[x,y] = A[i+x,k+y]
    b[x,y] = B[k+x,j+y]
    Barrier
    for k2 = 0 to 15
        c[x,y] += a[x,k2]*b[k2,y]
    Barrier
C[i+x,j+y] = c[x,y]
```

Data allocation

- Memory spaces: **Global**, **Shared**, **Local**

- As local as possible

```
c = 0
for k = 0 to n-1 step 16
  a[x,y] = A[i+x,k+y]
  b[x,y] = B[k+x,j+y]
  Barrier
  for k2 = 0 to 15
    c += a[x,k2]*b[k2,y]
  Barrier
C[i+x,j+y] = c
```

Local: private to each thread
(indices are implicit)

Global: shared between blocks /
inputs and outputs

Shared: shared between threads,
private to block

CUDA version

- Straightforward translation

```
float Csub = 0;
for(int a = aBegin, b = bBegin;
    a <= aEnd;
    a += aStep, b += bStep) {
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    As[ty][tx] = A[a + wA * ty + tx];
    Bs[ty][tx] = B[b + wB * ty + tx];

    __syncthreads();
    for(int k = 0; k < BLOCK_SIZE; ++k)
    {
        Csub += As[ty][k] * Bs[k][tx];
    }
    __syncthreads();
}

int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
```

Precomputed base addresses

Declare shared memory

Linearized arrays

Optimizing memory access patterns

- Back to the tiled matrix multiply algorithm

```
for // i = 0 to n-1 step 16  
  for // j = 0 to n-1 step 16
```

Level 2:
Blocks

```
  c = {0}  
  for k = 0 to n-1 step 16  
    a = A[i..i+15,k..k+15]  
    b = B[k..k+15,j..j+15]  
  
    for k2 = 0 to 15  
      for // i2 = 0 to 15  
        for // j2 = 0 to 15  
          c[i2,j2] += a[i2,k2]*b[k2,j2]  
  
  C[i..i+15,j..j+15] = c
```

Level 1:
Threads

- Let's focus on threads
- 

Memory access patterns

- On a block of 256 threads

	T0	T1	T2	T3		T16	T17		T255
x	0	1	2	3	...	0	1	...	15
y	0	0	0	0	...	1	1	...	15

```
c = 0
for k = 0 to n-1 step 16
  a[x,y] = A[i+x,k+y]
  b[x,y] = B[k+x,j+y]
  Barrier
  for k2 = 0 to 15
    c += a[x,k2]*b[k2,y]
  Barrier
  C[i+x,j+y] = c
```

Global

Shared

- Which accesses are coalesced?
- Are there bank conflicts?

Memory access patterns

- On a block of 256 threads

	T0	T1	T2	T3		T16	T17		T255
x	0	1	2	3	...	0	1	...	15
y	0	0	0	0	...	1	1	...	15

c = 0

for k = 0 to n-1 step 16

a[x,y] = A[i+x,k+y]

b[x,y] = B[k+x,j+y]

Barrier

for k2 = 0 to 15

c += **a[x,k2]*b[k2,y]**

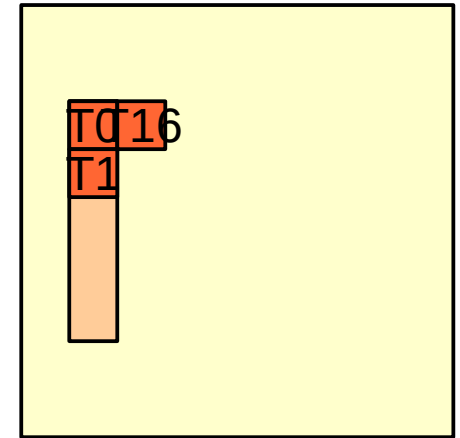
Barrier

C[i+x,j+y] = c

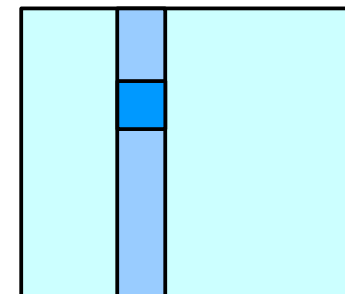
Global

Shared

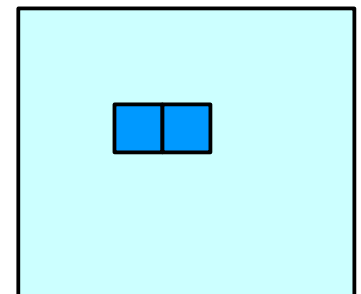
A,B



a



b



:(

- No coalesced access
- Massive bank conflicts

```
c = 0
for k = 0 to n-1 step 16
```

```
  a[x,y] = A[i+x,k+y]
```

```
  b[x,y] = B[k+x,j+y]
```

```
  Barrier
```

```
  for k2 = 0 to 15
```

```
    c += a[x,k2]*b[k2,y]
```

```
  Barrier
```

```
  C[i+x,j+y] = c
```

Column access: non coalesced

Broadcast from same element: no conflict

Column access: systematic conflicts

- Can we improve it?

Memory optimization

- Exchange x and y

```
c = 0
for k = 0 to n-1 step 16
  a[y,x] = A[i+y,k+x]
  b[y,x] = B[k+y,j+x]
  Barrier
  for k2 = 0 to 15
    c += a[y,k2]*b[k2,x]
  Barrier
  C[i+y,j+x] = c
```

Contiguous:
coalesced

Contiguous:
conflict-free

Broadcast:
conflict-free

- Success!
- Now can we improve memory parallelism?

Outline

- Host-side task and memory management
 - ◆ Asynchronism and streams
 - ◆ Compute capabilities
- Memory optimization
 - ◆ Memory access patterns
 - ◆ Global memory optimization
 - ◆ Shared memory optimization
 - ◆ Back to matrix multiplication
- Workload partitioning
- Instruction-level optimization

Workload partitioning

How to choose grid dimensions?

- Number of blocks per grid
 - ◆ Linear with data size, or constant
 - ◆ Min: at least number of SMs * blocks per SM
 - ◆ No max in practice
- Number of threads per block
 - ◆ Constant: should not depend on dataset size
 - ◆ Max: hardware limitation, 512 or 1024 threads
 - ◆ Min: size of a warp: 32 threads
- Iterations per thread
 - ◆ Constant or variable
 - ◆ Min: enough to amortize thread creation overhead
 - ◆ No max, but shorter-lived threads reduce load imbalance

Multiple grid/block dimensions

- Grid and block size are of type `dim3`

- ◆ Support up to 3 dimensions

```
dim3 dimBlock(tx, ty, tz);  
dim3 dimGrid(bx, by, bz);  
my_kernel<<<dimGrid, dimBlock>>>(arguments);
```

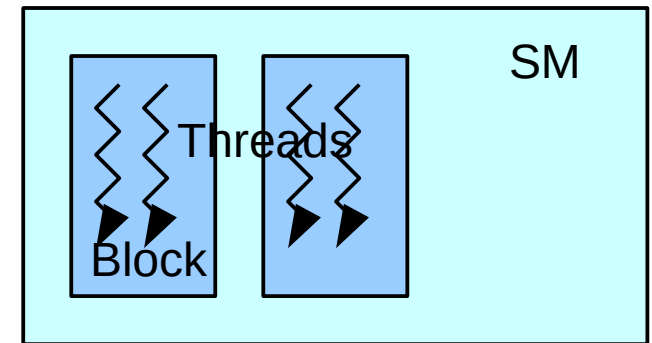
- ◆ Implicit cast from `int` to `dim3`
y and z sizes are 1 by default

- On device side, `threadIdx`, `blockDim`, `blockIdx`, `gridDim` are also of type `dim3`

- ◆ Access members with `.x`, `.y`, `.z`

Occupancy metric

- Threads per SM / max threads per SM
- Resource usage may cause non-ideal occupancy
 - ◆ Register usage
 - ◆ Shared memory usage
 - ◆ Non-dividable block size



Available registers: 32768



Usage: 64 registers/thread,
blocks of 256 threads

→ Only 2 blocks / SM

Available shared memory: 16KB



Usage: 12KB/block

→ Only 1 block / SM

Max threads/SM: 768 threads



Block size: 512 threads

→ Only 1 block / SM

Could run 3 blocks of 256 threads

GPU: on-chip memory

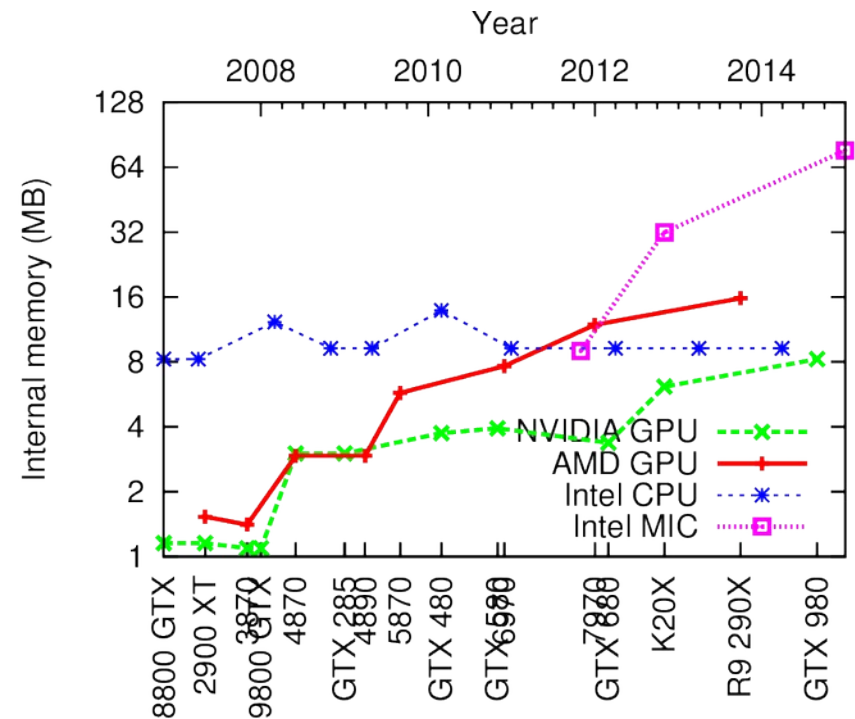
- Conventional wisdom
 - ◆ Cache area in CPU vs. GPU according to the NVIDIA CUDA Programming Guide:



Figure 1-2. The GPU Devotes More Transistors to Data Processing

- But... if we include registers:

GPU	Register files + caches
NVIDIA GM204 GPU	8.3 MB
AMD Hawaii GPU	15.8 MB
Intel Core i7 CPU	9.3 MB



- GPU/accelerator internal memory exceeds desktop CPU's

How many threads?

- As many as possible (maximize occupancy)?
 - + Maximal data-parallelism
 - Latency hiding
 - Locality
 - Store private data of each thread
 - Thread management overhead
 - Initialization, redundant operations
- Trade-off between parallelism and memory locality

Multiple elements per thread

- Block size (16, 16) \rightarrow (8, 16)
- 2 elements per thread: (x, y) and (x+8, y)

```
c[0] = 0
c[1] = 0
for k = 0 to n-1 step 16
    a[y,x] = A[i+y,k+x]
    b[y,x] = B[k+y,j+x]
    a[y+8,x] = A[i+y+8,k+x]
    b[y+8,x] = B[k+y+8,j+x]
    Barrier
    for k2 = 0 to 15
        c[0] += a[y,k2]*b[k2,x]
        c[1] += a[y+8,k2]*b[k2,x]
    Barrier
    C[i+y,j+x] = c[0]
    C[i+y+8,j+x] = c[1]
```

More outstanding loads



- What about shared memory?

Data reuse

- Share reads to submatrix b
 - ✦ Fewer shared memory accesses
 - ➔ Exchange data through registers

```
c[0] = 0
c[1] = 0
for k = 0 to n-1 step 16
    a[y,x] = A[i+y,k+x]
    b[y,x] = B[k+y,j+x]
    a[y+8,x] = A[i+y+8,k+x]
    b[y+8,x] = B[k+y+8,j+x]
    Barrier
    for k2 = 0 to 15
        bl = b[k2,x]
        c[0] += a[y,k2]*bl
        c[1] += a[y+8,k2]*bl
    Barrier
    C[i+y,j+x] = c[0]
    C[i+y+8,j+x] = c[1]
```

- Improves register usage too. Why?

Expressing vector loads

- Multiple **consecutive** elements per thread
 - ◆ Here: $4*x$, $4*x+1$, $4*x+2$, $4*x+3$
- Load, store, and compute on short vectors

```
c[0..3] = 0
for k = 0 to n-1 step 4*16
    a[y, 4*x..4*x+3] = A[i+y, k+4*x..k+4*x+3]
    b[y, 4*x..4*x+3] = B[k+y, j+4*x..j+4*x+3]
    Barrier
    for k2 = 0 to 15
        bl[0..3] = b[k2, 4*x..4*x+3]
        c[0..3] += a[y, k2] * bl[0..3]
    Barrier
    C[i+y, 4*(j+x)..4*(j+x)+3] = c[0..3]
```

Vector loads from global memory

Vector loads/store-in shared memory

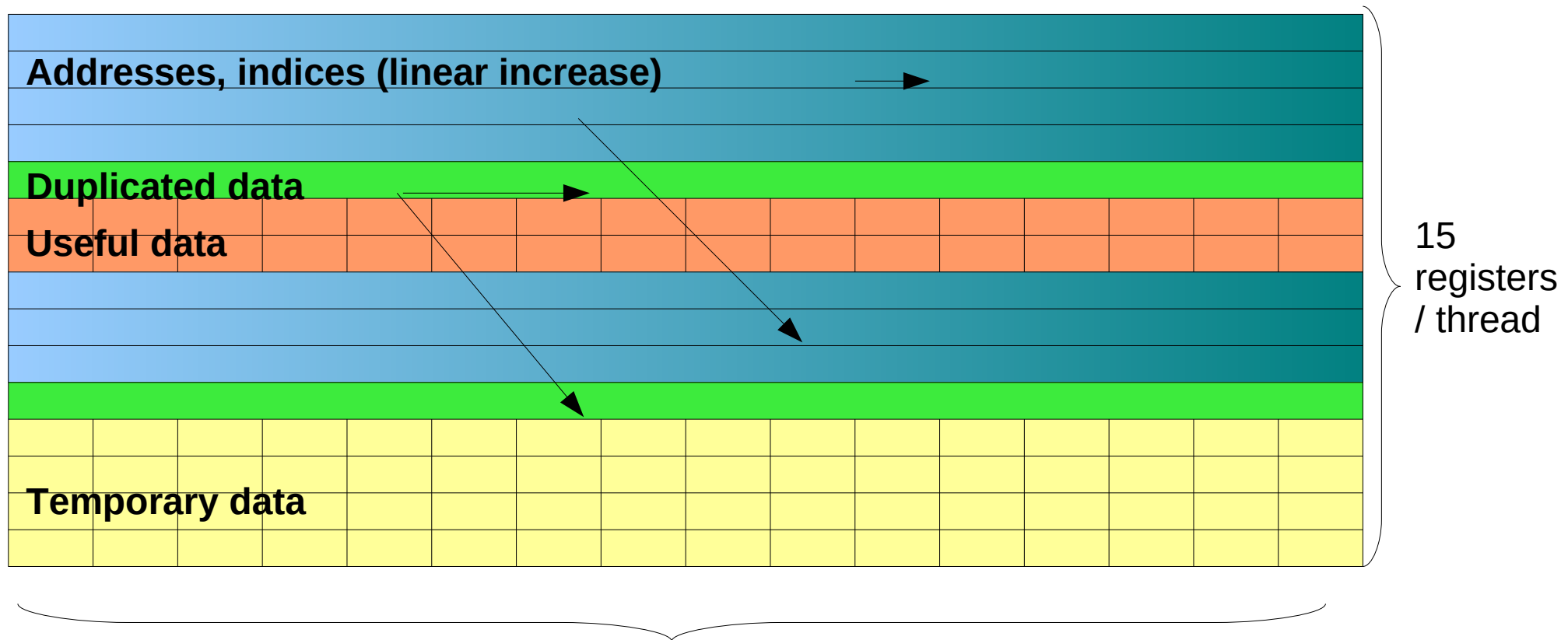
Scalar-vector product

Vector store to global memory

Example: SGEMM from CUBLAS 1.1

NVIDIA's reference matrix multiplication code in 2008

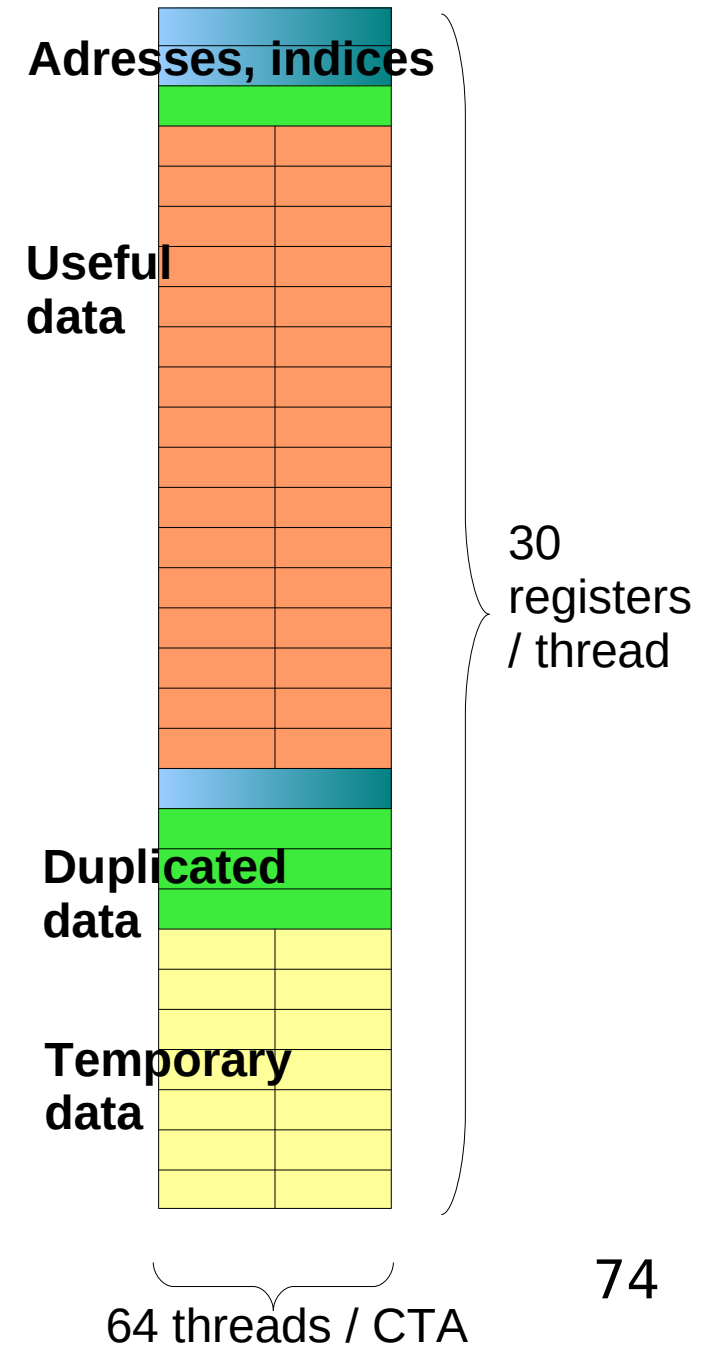
- 512 threads / CTA, 15 registers / thread
- 9 registers / 15 contain redundant data
- Only 2 registers really needed



Example from: Volkov. Programming inverse memory hierarchy : case of stencils on GPUs. *ParCFD*, 2010.

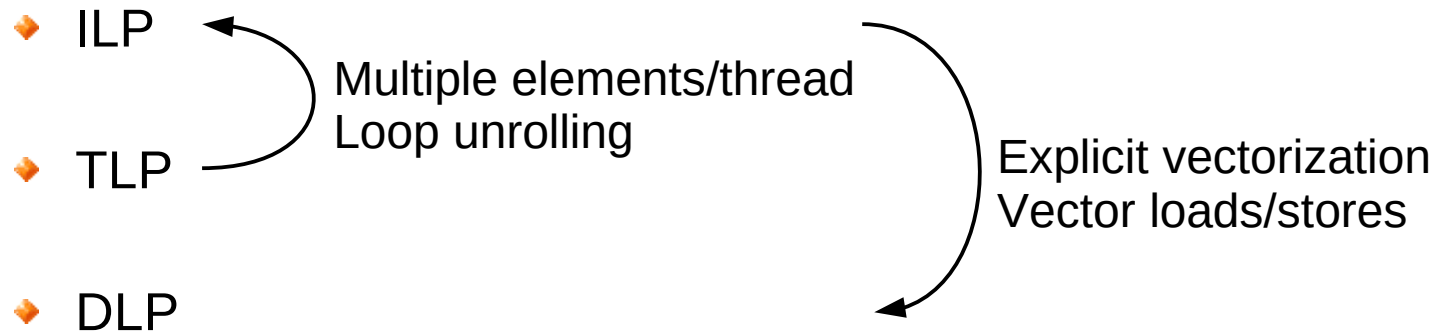
Fewer threads, more computations

- Optimized version:
SGEMM in CUBLAS 2.0
 - ◆ 8 elements computed / thread
 - ◆ Unrolled loops
 - ◆ Less traffic through shared memory, more through registers
- Overhead amortized
 - ◆ 1920 registers vs. 7680 for the same amount of work
 - ◆ Works for redundant computations too
- ➔ Instruction-level parallelism is still relevant



Re-expressing parallelism

- Converting types of parallelism



- General strategy

- ◆ Design phase: focus on thread-level parallelism
- ◆ Optimization phase:
convert TLP to Instruction-level or Data-level parallelism

Outline

- Host-side task and memory management
 - ◆ Asynchronism and streams
 - ◆ Compute capabilities
- Memory optimization
 - ◆ Memory access patterns
 - ◆ Global memory optimization
 - ◆ Shared memory optimization
 - ◆ Back to matrix multiplication
- Workload partitioning
- **Instruction-level optimization**

Loop unrolling

- Can improve performance
 - ◆ Amortizes loop overhead over several iterations
 - ◆ May allow constant propagation, common sub-expression elimination...
- Unrolling is **necessary** to keep arrays in registers

Not unrolled

```
int a[4];  
for(int i = 0; i < 4; i++) {  
    a[i] = 3 * i;  
}
```

Indirect addressing:
a in local memory

Unrolled

```
int a[4];  
a[0] = 3 * 0;  
a[1] = 3 * 1;  
a[2] = 3 * 2;  
a[3] = 3 * 3;
```

Static addressing:
a in registers

Trivial
computations:
optimized away

- The compiler can unroll for you

```
#pragma unroll  
for(int i = 0; i < 4; i++) {  
    a[i] = 3 * i;  
}
```

Warp-based execution

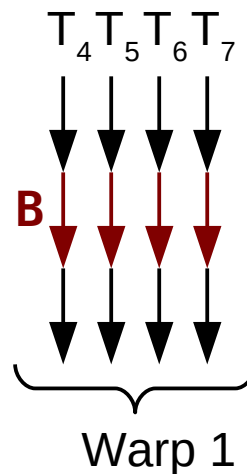
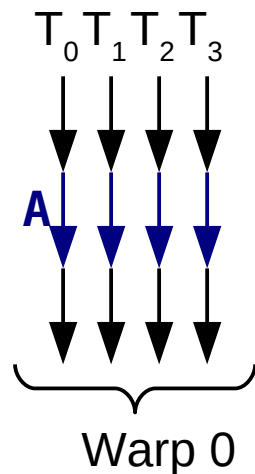
- Threads in a warp run in lockstep
- On NVIDIA architectures, warp is 32 threads
- A block is made of warps
(warps do not cross block boundaries)
 - ◆ Block size multiple of 32 for best performance

Branch divergence

- Conditional block

```
if(c) {  
    // A  
}  
else {  
    // B  
}
```

- All threads of a warp take the same path



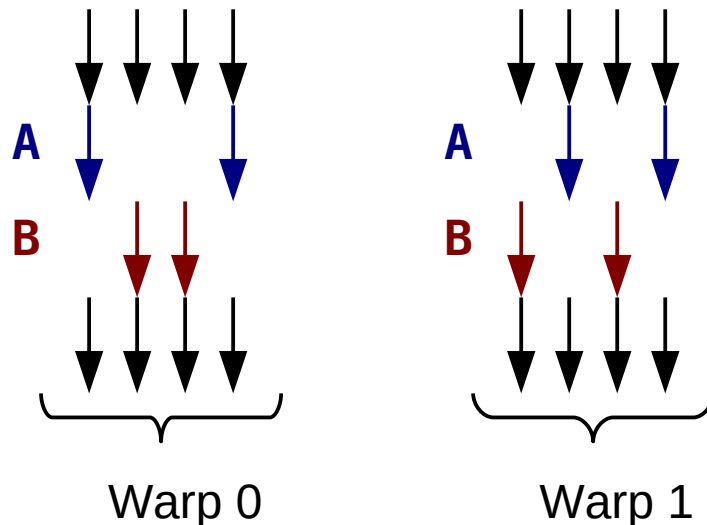
With imaginary 4-thread warps

Branch divergence

- Conditional block

```
if(c) {  
    // A  
}  
else {  
    // B  
}
```

- Threads in a warp take different paths



- Warps have to go through both **A** and **B**: lower performance

Avoiding branch divergence

- Hoist identical computations and memory accesses outside conditional blocks

```
if(tid % 2) {  
    s += 1.0f/tid;  
}  
else {  
    s -= 1.0f/tid;  
}
```

```
float t = 1.0f/tid;  
if(tid % 2) {  
    s += t;  
}  
else {  
    s -= t;  
}
```

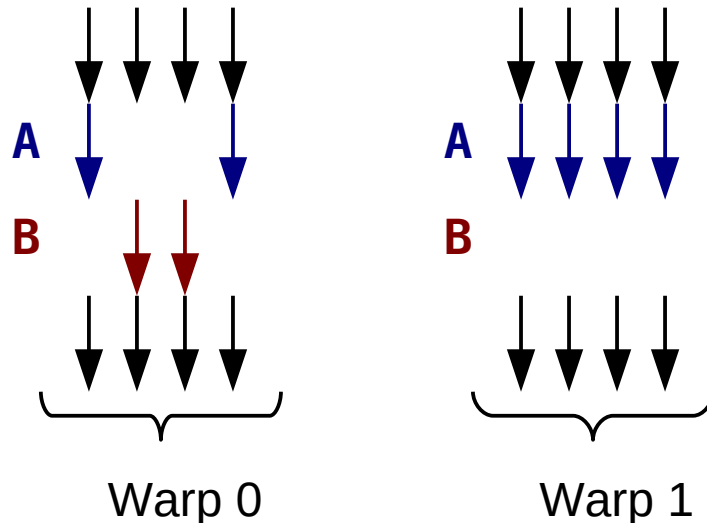
- When possible, re-schedule work to make non-divergent warps

```
// Compute 2 values per thread  
int i = 2 * tid;  
s += 1.0f/i - 1.0f/(i+1);
```

- What if I use C's ternary operator (?:) instead of if?
(or tricks like ANDing with a mask, multiplying by a boolean...)

Ternary operator ? good : bad

- Run both branches and select: $R = c ? A : B;$
 - ◆ No more divergence?
- All threads have to take both paths
No matter whether the condition is divergent or not



- Does **not** solve divergence: we lose in all cases!
- Only benefit: fewer instructions
 - ◆ May be faster for short, often-divergent branches
- Compiler will choose automatically when to use predication
 - ◆ Advice: keep the code readable, let the compiler optimize

Recap

- Beware of local arrays
use static indices and loop unrolling
- Keep in mind branch divergence when writing algorithm
but do not end up in managing divergence yourself

Takeaway

- Distribute work and data
 - ◆ Favor SoA
 - ◆ Favor locality and regularity
 - ◆ Use common sense (avoid extraneous copies or indirections)
- More threads \neq higher performance
 - ◆ Saturate instruction-level parallelism first (almost free)
 - ◆ Complete with data parallelism (expensive in terms of locality)
- Usual advice applies
 - ◆ First write correct code
 - ◆ Profile
 - ◆ Optimize
 - ◆ Repeat

References and further reading/watching

- CUDA C Programming Guide
- Mark Harris. Introduction to CUDA C.
<http://developer.nvidia.com/cuda-education>
- David Luebke, John Owens. Intro to parallel programming.
Online course. <https://www.udacity.com/course/cs344>
- Paulius Micikevicius. GPU Performance Analysis and Optimization.
GTC 2012.
<http://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC-2012-GPU-Performance-Analysis.pdf>