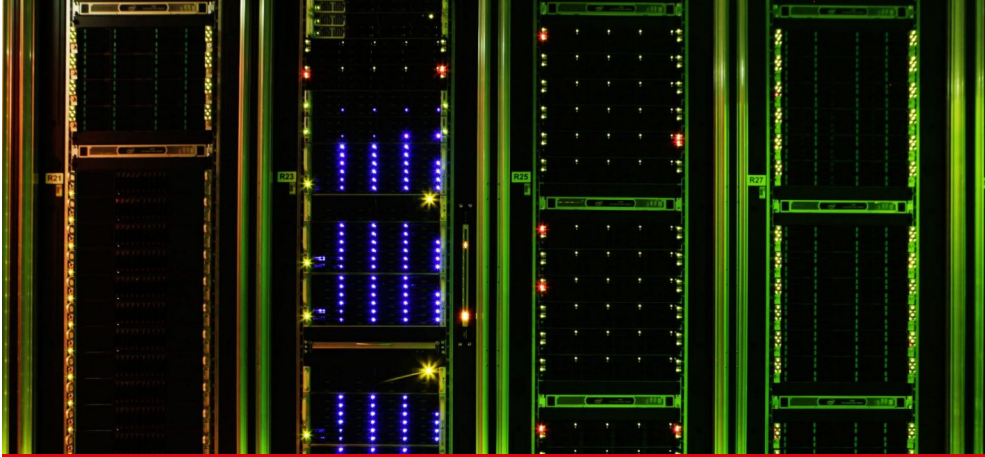


MATH-454 Parallel and High Performance Computing

Lecture 8: Advanced GPU computing

Pablo Antolin

Slides of N. Varini's lecture notes

April 17 2025

SCITAS

Outline

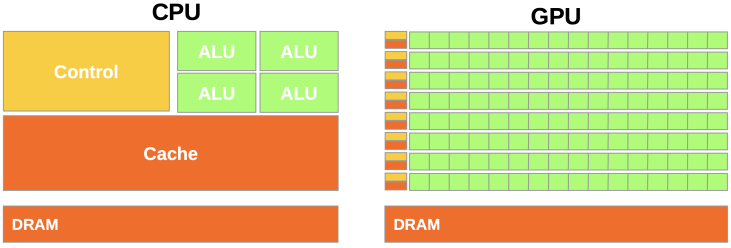
- Recap
- Thread Cooperation in GPU Computing
- GPU Memory Model
  - Shared memory
  - Constant memory
  - Global memory
- Test cases

Data-Parallel Computing - Recap

- Performs operations on a data set organized into a common structure (e.g., an array)
- A set of tasks work **collectively and simultaneously** on the same structure with each task operating on its own part of the structure
- Tasks perform identical operations on their parts of the structure. Operations on each part must be data independent
- CUDA provide built-in variables in order to access to different data: `threadIdx`, `blockIdx`, `blockDim`, ...

Data-Parallel Computing on the GPUs - Recap

- GPUs are suited for number crunching problems
- Identical operations executed on many data elements in parallel
- Lots of transistors are dedicated to the computation

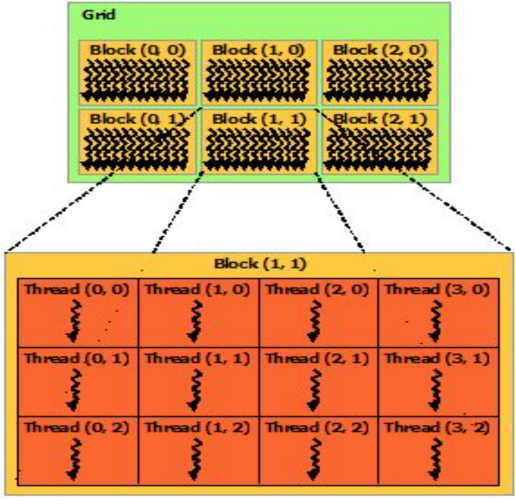


The CUDA Programming Model

- We know that GPUs are pieces of hardware that can run many threads
- Threads are organized in grids of blocks
- Threads are executed in warps (32 threads) in Streaming Multiprocessors (SM)
- But, how well do we need to know the hardware in order to obtain good performance?
- GPUs has different memory levels

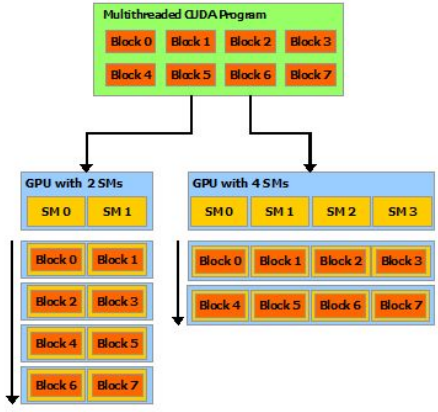
CUDA Thread Hierarchy

- Thread blocks and grids can be 1D, 2D or 3D
- Dimensions set at launch time
- Thread blocks and grids do not need to have the same dimensionality, e.g. 1D grid of 2D blocks



The CUDA Programming Model

- Blocks from the grid are distributed across the SM
- The programmer has no control on this distribution
- A block will execute on one (and only one) SM



Blocks Must Be Independent!

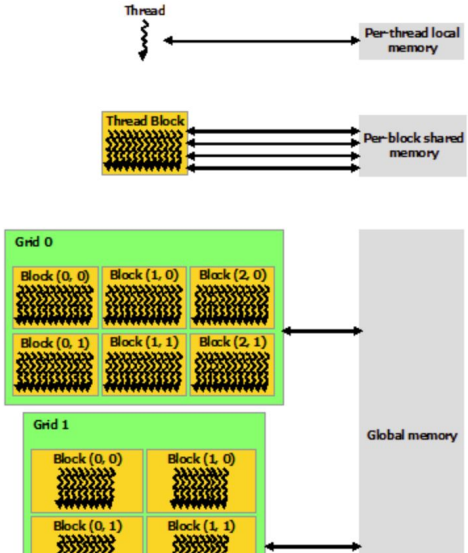
- Any possible distribution of blocks could be valid
  - Can run in any order
  - Can run sequentially or concurrently
- Blocks might need to be synchronized once in a while (more later)
- Independence requirements gives scalability
- There are mechanisms for synchronization among blocks, but we don't cover it in this course, so we consider them as independent.

The CUDA Programming Model

- However, within a block, CUDA permits **non data-parallel approaches**
  - Implemented via control-flows statements in a kernel
  - Threads are free to execute unique paths through a kernel
- Since all threads within a block are active at the same time they can communicate between each other

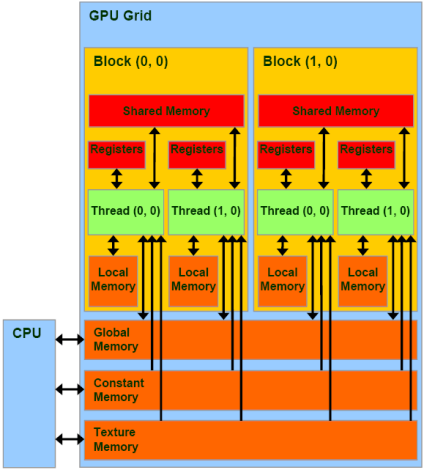
CUDA threads may access data from **multiple memory spaces** during their execution.

- Each thread has its own registers private local memory.
- Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block.
- All threads have access to the same global memory (and constant and texture memories).



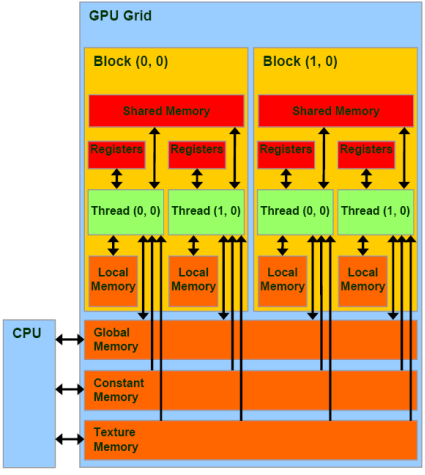
GPU Memory Overview

- Transfer to/from CPU is very slow
- Global memory is slow
- Texture, Constant, and Shared Memory are fast
- Registers are very fast



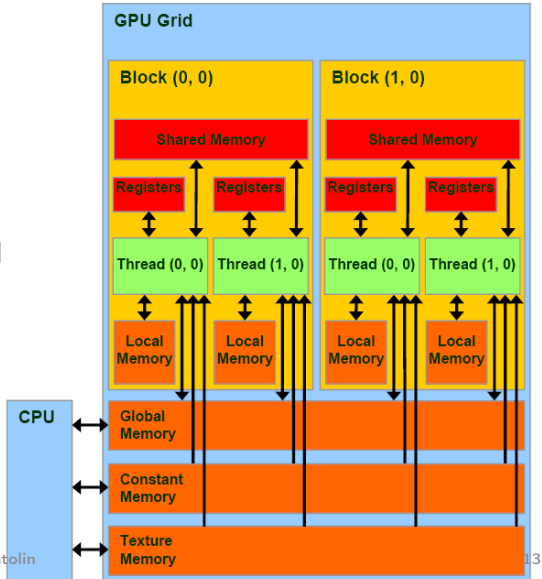
Global Memory

- Visible by all threads
- Read/write
- Shared between blocks and grids
- Stays alive during multiple kernel executions
- Slow access
- Programmer explicitly manages allocation and deallocation with cuda API (`cudaMallocManaged`, `cudaFree`)



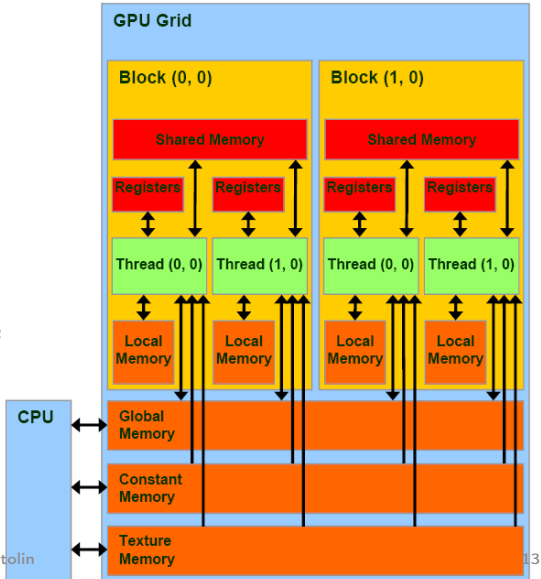
Constant Memory

- Special region of device memory
- Read-only in device
- Cached in multiprocessor
- Fairly quick, cache can broadcast to all active threads
- It is small: 64KB in NVIDIA V100
- To use when:
  - All threads access to the same location
  - Data is constant



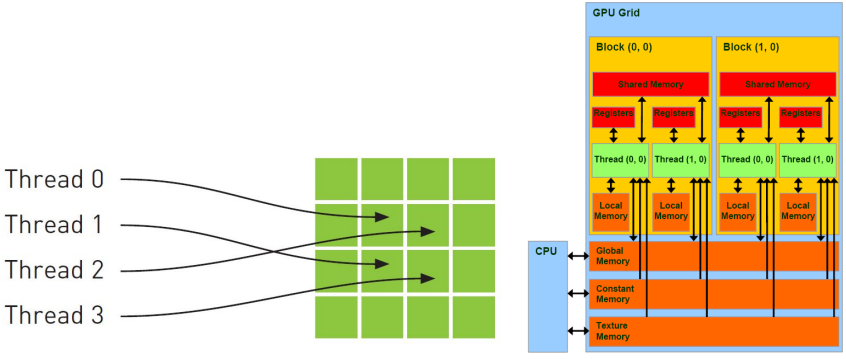
Constant Memory

- Read-only from kernel
- Constants are declared at file scope
  - `__device__ __constant__`
- Constant values are set from host code
  - `cudaMemcpyToSymbol()`



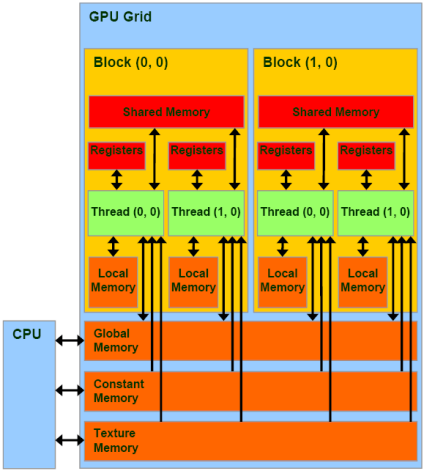
Texture Memory

- Texture caches are designed for graphics applications where memory access patterns exhibit a great deal of **spatial locality**.
- A thread is likely to read from an address "near" to the address that nearby threads area



Shared Memory

- High performance memory
- Read/write per block
- Memory is shared within a block
- Very fast
  - 2 orders of magnitude lower latency than global memory
  - Order of magnitude higher bandwidth than global memory
- In V100, up to 128 KB per multiprocessor, but a maximum of 48 KB per block

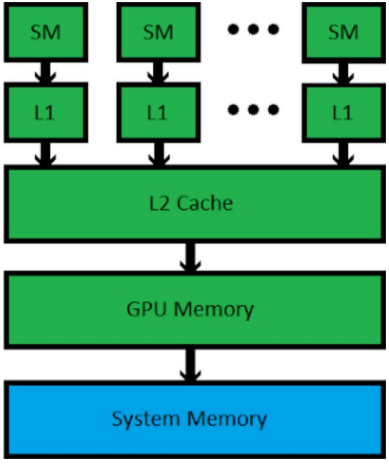


Shared Memory

- Shared memory has block scope (stays alive while the block lives)
- Only visible to threads in the same block
- Threads can share results, avoid redundant computations
- Threads can share memory access (avoid redundant accesses to global memory)
- Similar benefits as CPU cache, however, must be explicitly managed by the programmer with the qualifier `__shared__`

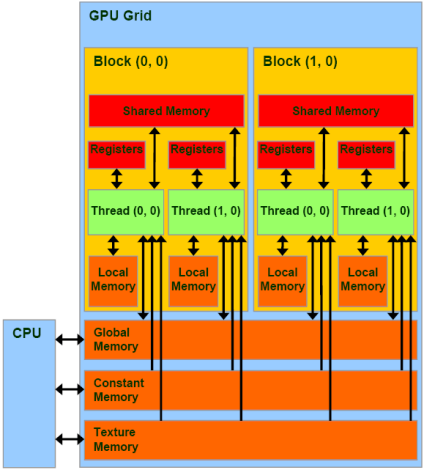
Shared Memory

- When a variable is declared in shared memory the compiler creates a copy of that variable for each block.
- Every thread within the blocks sees this memory, can access and modify its content. Threads from other blocks do not see the same memory.
- This provides an excellent means by which threads within a block can communicate and collaborate on computations.
- However, threads have to be synchronized explicitly.



Local Memory

- Part of the global memory private to a thread.
- Read/write
- Slow
- Used for whatever does not fit into registers (including arrays)
- http://developer.download.nvidia.com/CUDA/training/register_spilling.pdf



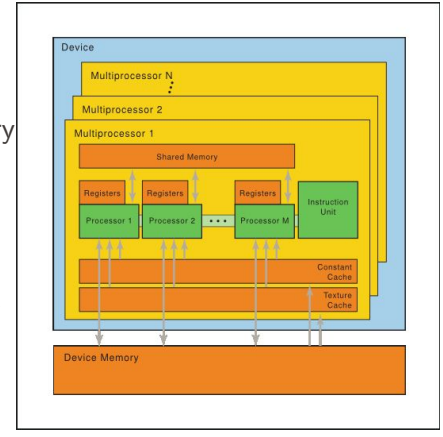
Local Memory

- Variable declared within a kernel is allocated per thread
- It is only accessible by the threads
- It has the lifetime of a thread

```
__global__
void kernel() {
    // Each thread has its own copy of idx and
     ↪  array
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    float array[16];
  }
```

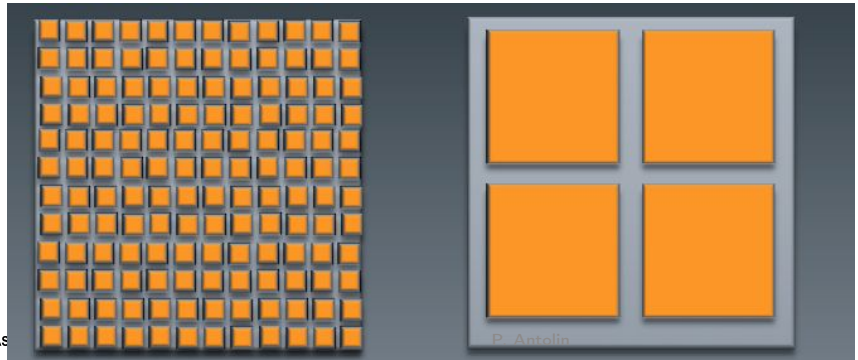
Local Memory

- Compilers control where these variables are stored in physical memory (programmer does not have control)
- Registers: fastest memory, on chip
- Local memory: when registers are not available compilers put off chip



Memory issues

- Each multiprocessor has limited amount of memory
- Limits amount of blocks we can have
- $\#\text{blocks} \times \text{mem\_used\_per\_block} \Leftarrow \text{total memory}$
- Either lots of blocks using little memory, or fewer blocks using lots of memory



Memory issues

- Register memory is limited!
- Shared memory in blocks is limited!
- Can have many threads using fewer registers, or few threads using many registers
- What does not fit into registers goes to local memory (slow).

Memory issues

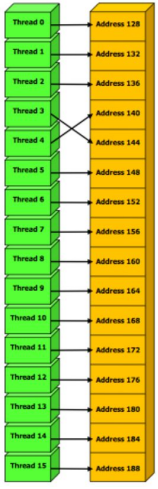
- Global accesses: slow!
- Can be sped up when memory is contiguous
  - All threads in a warp execute the same instruction
  - During a load the hardware detect whether all threads access to consecutive memory locations.
  - If thread 0 access location n, thread 1 to location n+1, thread 31 to location n+31 the all accesses are **coalesced**: combined in a single memory access.
  - Coalesced access are: **contiguous**, **in-order**, **aligned**

Memory Coalescing: Contiguous access

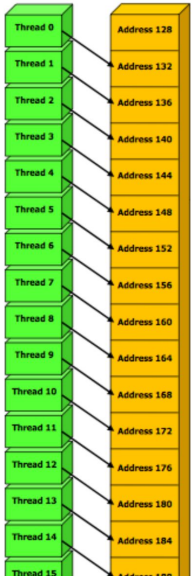
- Contiguous = memory is together
- Do not skip addresses
- Example of non-contiguous memory
  - Left: Address 140 skipped
  - Right: Lots of addresses skipped

Memory Coalescing: In-order access

- In-order access
- Access addresses in order in memory
- Examples of non-ordered accesses
  - Thread 3 and 4 swapped accesses



Memory Coalescing: Aligning access



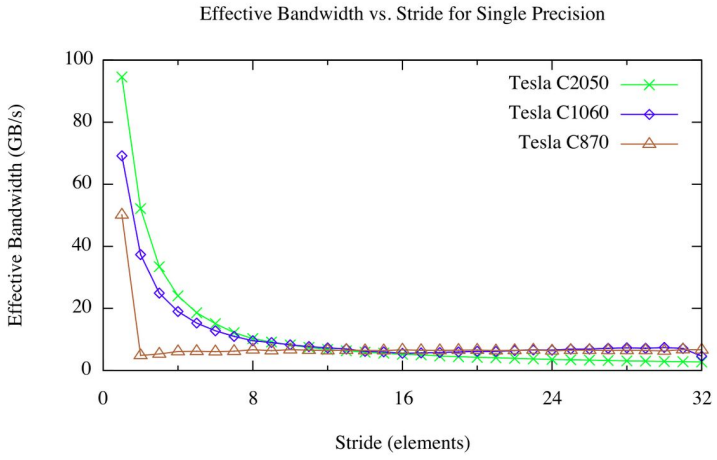
Bad alignment

- Built-in types force alignment
- float3(12B) takes up the same space as float4(16B)
- float3 arrays are not aligned;
- To align a struct use `__align__(x) // x = 4, 8, 16`
- `CudaMalloc` aligns the start of each block automatically

Let's put it to the test - non contiguous memory access

```
template<typename T>
__global__
void stride(T *a, int s)
{
    int i = (blockDim.x * blockIdx.x + threadIdx.x) * s;
    a[i] = a[i] + 1;
}
```

Let's put it to the test - non contiguous memory access



A deeper dive into shared memory

- It is on-chip memory $\rightarrow$ much faster
- Latency can be $\sim$100x lower than global memory access
- Allocated per block. All the threads can access to it.
- If threads A and B load data from global memory and write in shared there could be race conditions $\rightarrow$ explicit synchronization

Static shared memory

```
__global__ void staticFunction(int *arr, int N)
{
__shared__ int shared_array[THREADS_PER_BLOCK];  //shared block memory
int idx = blockIdx.x * blockDim.x + threadIdx.x;

// ... calculate results
shared_array[threadIdx.x] = results;

//synchronize the shared threads writing to the shared memory cache
__syncthreads();

// read the results of another thread in the current thread
int val = shared_array[(threadIdx.x + 1) % THREADS_PER_BLOCK];

arr[idx] = val; // write back the value to global memory
}
```

Dynamic shared memory

```
__global__
void dynamicFunction(int *arr, int N)
{
extern __shared__ int local_array[];

...
}

...

dynamicFunction<<<1,N,N*sizeof(int)>>>(arr,N)
```

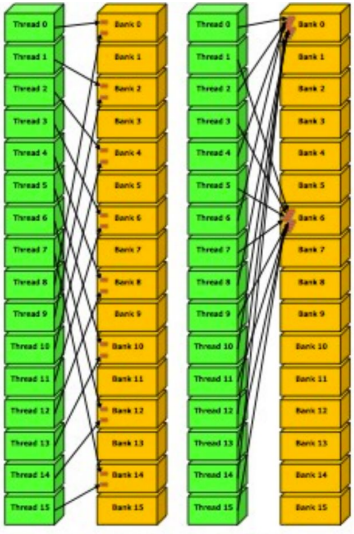
- The amount of shared memory is not known till runtime
- The amount of memory must be specified as 3rd parameter when then kernel is launched

Shared memory bank conflict

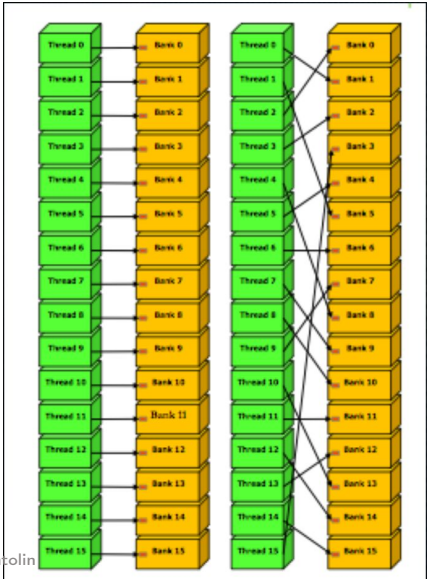
- To achieve high memory bandwidth for concurrent access, shared memory is divided into banks that cannot be accessed simultaneously.
- If multiple threads requested addresses map to the same memory bank, the accesses are serialized.
- The hardware splits a conflicting memory request into as many separate conflict-free requests as necessary, decreasing the effective bandwidth by a factor equal to the number of colliding memory requests.
- To minimize bank conflicts, it is important to understand how memory addresses map to memory banks.
- Shared memory banks are organized such that successive 32-bit words are assigned to successive banks and the bandwidth is 32 bits per bank per clock cycle.
- For devices of compute capability 7.X, the warp size is 32 threads and the number of banks is also 32.

Bank conflicts

Bad: many threads trying to access to the same bank



Good: Few to no bank conflicts



Bank Conflicts for shared memory

Banks service 32-bit words at a time at addresses mod 64

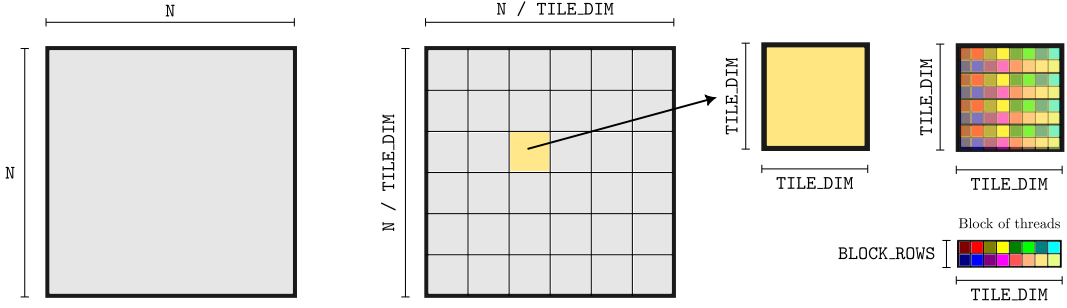
- Bank 0 manages 0x00, 0x40, 0x80, etc., bank 1 manages 0x04, 0x44, 0x84, etc.

- Want to avoid multiple thread access to same bank
  - Usually a problem if many threads access to the same bank
  - Padding if necessary
  - Last thing to worry about for performance

Example: An efficient Matrix Transpose in CUDA

- This exercise is based on https://devblogs.nvidia.com/efficient-matrix-transpose-cuda-cc/
- The code we wish to optimize is a transpose of a matrix of single precision values that operates out-of-place, i.e., the input and output are separate arrays in memory.
- **Data is allocated by row**. For simplicity , we consider square matrices with number of rows multiple of 32.
- All kernels in this study launch blocks of 32×8 threads (`TILE_DIM=32`, `BLOCK_ROWS=8` in the code), and each thread block transposes (or copies) a tile of size 32×32.
- Using a thread block with fewer threads than elements in a tile is advantageous for the matrix transpose because each thread transposes four matrix elements, so much of the index calculation cost is amortized over these elements.

Let's start with matrix copy



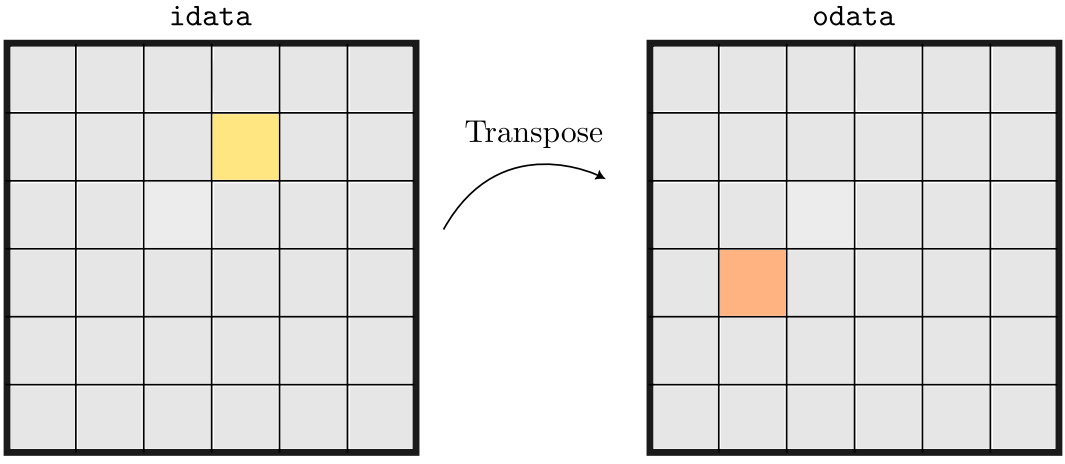
Let's start with matrix copy

```
__global__
void copy(float *odata, const float *idata) {
    int x = blockIdx.x * TILE_DIM + threadIdx.x;
    int y = blockIdx.y * TILE_DIM + threadIdx.y;
    int N = gridDim.x * TILE_DIM;

    for(int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
    odata[(y + j) * N + x] = idata[(y + j) * N + x];
  }
```

- Each thread copies `TILE_DIM/BLOCK_ROWS` values
- `TILE_DIM` must be used to compute the indices x, y
- This implementation is used as a reference of bandwidth

Naïve matrix transpose



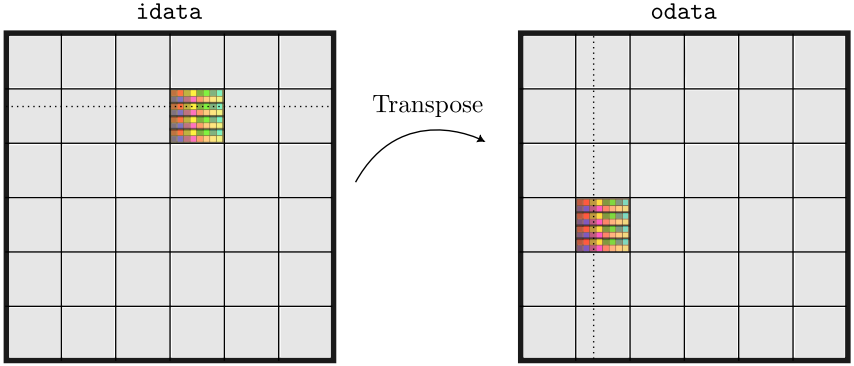
Naïve matrix transpose

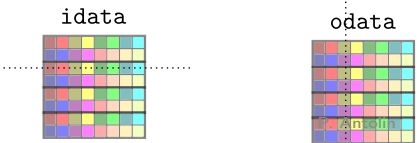
```
__global__
void transposeNaive(float *odata, const float *idata) {
    int x = blockIdx.x * TILE_DIM + threadIdx.x;
    int y = blockIdx.y * TILE_DIM + threadIdx.y;
    int N = gridDim.x * TILE_DIM;

    for(int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
    odata[x * N + (y + j)] = idata[(y + j) * N + x];
  }
```

- The only difference is that the indices for `odata` are swapped.
- The access to `idata` is coalesced while for `odata` is not.

Naïve matrix transpose





Naïve matrix transpose

	Effective Bandwidth (GB/s, ECC enabled)	
Routine	Tesla M2050	Tesla K20c
copy	105.2	136.0
transposeNaive	18.8	55.3

- The `transposeNaive` bandwidth is a fraction of the copy.

Coalesced transpose via shared memory - 1/2

```
__global__ void transposeCoalesced(float *odata, const float *idata) {

    __shared__ float tile[TILE_DIM][TILE_DIM];

    int x = blockIdx.x * TILE_DIM + threadIdx.x;
    int y = blockIdx.y * TILE_DIM + threadIdx.y;
    int N = gridDim.x * TILE_DIM;

    for(int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
       tile[threadIdx.y+j][threadIdx.x] = idata[(y+j) * N + x];

    __syncthreads();

    // ... continues in next slide
```

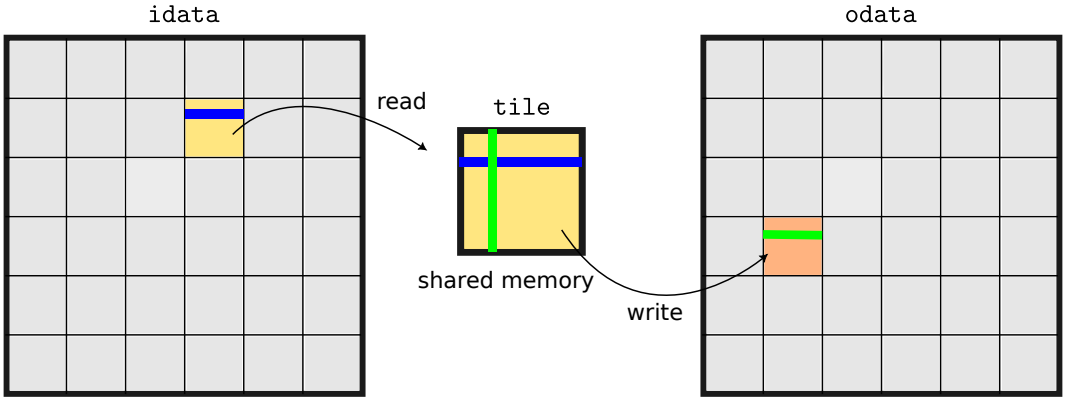
- In the first loop, a warp of threads reads contiguous data from `idata` into rows of the shared memory tile

Coalesced transpose via shared memory - 2/2

```
1   __syncthreads();
2
3   // ... continues from previous slide
4
5   // Transpose block offset
6   x = blockIdx.y * TILE_DIM + threadIdx.x;
7   y = blockIdx.x * TILE_DIM + threadIdx.y;
8
9   for(int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
10  odata[(y+j) * N + x] = tile[threadIdx.x][threadIdx.y+j];
11 }
```

- After recalculating the array indices, a column of the shared memory tile is written to contiguous addresses in `odata`
- Because threads write in different data to `odata` than they read from `idata`, we must use a block-wise barrier synchronization `__syncthreads()`

Coalesced transpose via shared memory



Coalesced transpose via shared memory

	Effective Bandwidth (GB/s, ECC enabled)	
Routine	Tesla M2050	Tesla K20c
copy	105.2	136.0
transposeNaive	18.8	55.3
transposeCoalesced	51.3	97.6

- The results improved a lot but they are still far from copy.
- Copy data to shared memory and synchronization might be responsible for slow down.

Simple copy in shared memory - 1/2

```
__global__ void copySharedMem(float *odata, const float *idata) {

    __shared__ float tile[TILE_DIM * TILE_DIM];

    int x = blockIdx.x * TILE_DIM + threadIdx.x;
    int y = blockIdx.y * TILE_DIM + threadIdx.y;
    int N = gridDim.x * TILE_DIM;

    for(int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
        tile[(threadIdx.y+j) * TILE_DIM + threadId.x] = idata[(y+j)
        ↪  * N + x];

    __syncthreads();
    // ... continues in next slide
```

Simple copy in shared memory - 2/2

```
1   __syncthreads();
2   // ... continues from previous slide
3
4   for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
5       odata[(y+j)*width + x] = tile[(threadIdx.y+j)*TILE_DIM +
        ↪  threadIdx.x];
6 }
```

- The `__syncthreads` is technically not needed (no race conditions)
- Included to mimic the behavior above
- The problem is not the barrier or the thread synchronization

Copy in shared memory

	Effective Bandwidth (GB/s, ECC enabled)	
Routine	Tesla M2050	Tesla K20c
copy	105.2	136.0
copySharedMem	104.6	152.3
transposeNaive	18.8	55.3
transposeCoalesced	51.3	97.6

Shared memory bank conflicts

- For a shared memory tile of 32×32 elements, all elements in a column of data map to the same shared memory bank.
- Worst case scenario: reading a column of data results in a 32-way bank conflict.
- the solution for this is simply to pad the width in the declaration of the shared memory tile, making the tile 33 elements wide rather than 32.

```
1 __shared__ float tile[TILE_DIM][TILE_DIM + 1];
```

	Effective Bandwidth (GB/s, ECC enabled)	
Routine	Tesla M2050	Tesla K20c
copy	105.2	136.0
copySharedMem	104.6	152.3
transposeNaive	18.8	55.3
transposeCoalesced	51.3	97.6
transposeNoBankConflicts	99.5	144.3

Summary

- Best memory management
- Balances memory optimization with parallelism
- Break problem up into a coalesced chucks
- Process data in shared memory, then copy to global
- Avoid bank conflicts!