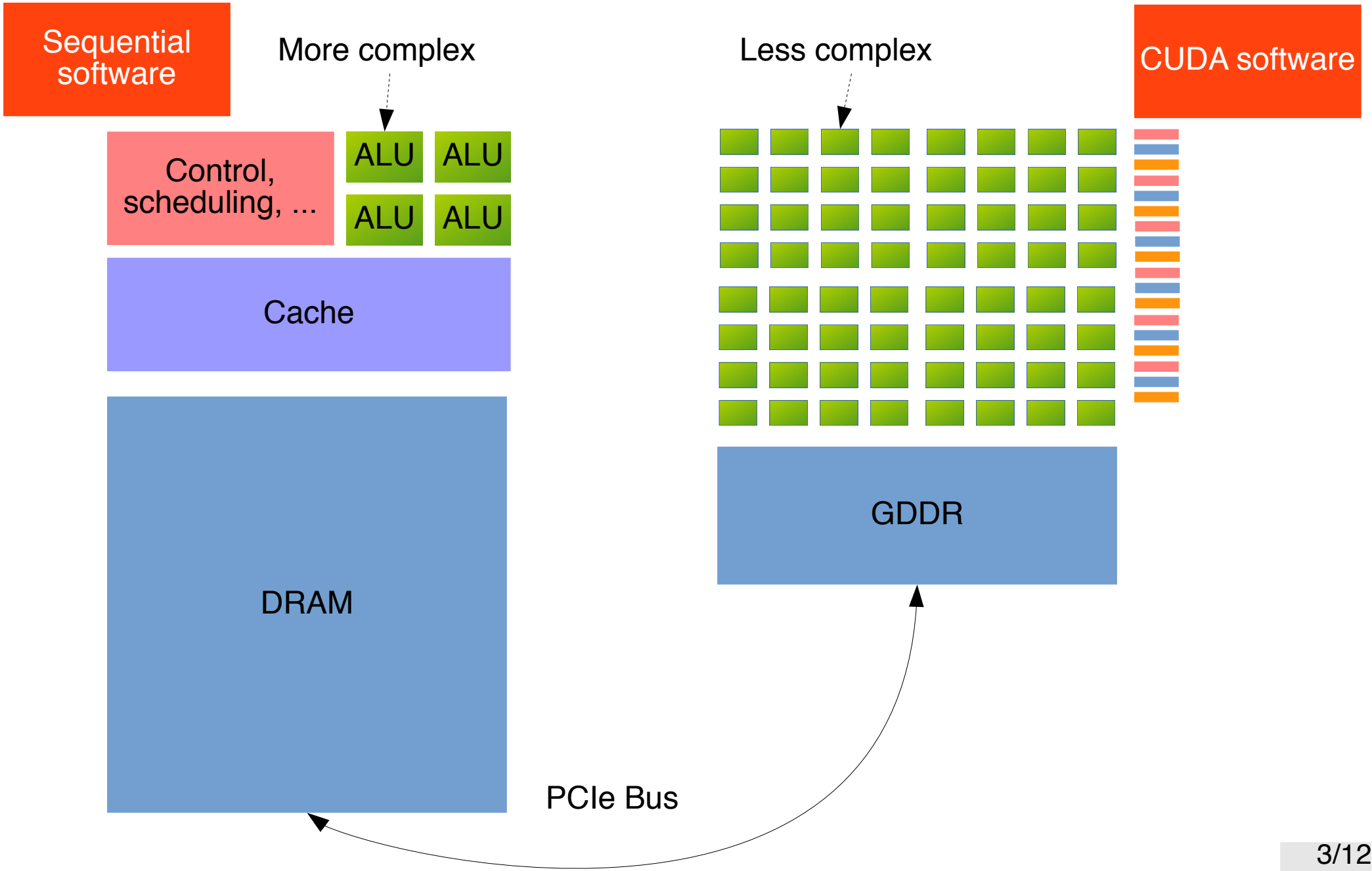
CUDA Programming: Thread Scheduling

Piotr Luszczek

Thread Identification

- POSIX threads
  - `pthread_t tid = pthread_self();`
- MPI
  - `MPI_Comm_rank(comm, &rank);`
  - `MPI_Comm_size(comm, &size);`
- OpenMP
  - `int tid = omp_get_thread_num();`
  - `int all = omp_get_num_threads();`
- CUDA
  - `int blkid = blockIdx.x + (blockIdx.y + blockIdx.z * gridDim.y) * gridDim.x`
  - `int inside_blk_tid = threadIdx.x + (threadIdx.y + threadIdx.z * blockDim.y) * blockDim.x`

GPU Optimization Goal: Target the Hardware



Matrix Summation: CPU

```
float A[n][n], B[n][n], C[n][n];

/* note the order of the loops */
for (int i=0; i<n; ++i)
  for (int j=0; j<n; ++j)
    C[i][j] = A[i][j] + B[i][j];
    // row-major order
```

Matrix Summation: GPU Kernel

```
__global__ void matrix_sum(float *A, float *B,
float *C, int m, int n) {
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;

  if (x < m && y < n) {
    int ij = x + y*m; // column-major order
    C[ij] = A[ij] + B[ij];
  }
}
```

Matrix Summation: GPU Launching (Slow!)

```
// optimization: copy data outside of the loop
cudaMemcpy(...);
for (int i=0; i<n; ++i)
  for (int j=0; j<n; ++j) {
    int ij = i + j*n; // column-major order
    matrix_sum<<1,1>>(dA+ij, dB+ij, dC+ij, 1,1);
    // problem: kernel launch overhead 1-10 ms
  }
cudaMemcpy(...);
```

Matrix Summation: GPU Launching (Faster!)

- Kernel launch for every row

```
for (int i=0; i<n; ++i)
  matrix_sum<<1,n>>(dA+i, dB+i, dC+i, 1,n );
```

- Kernel launch for every column

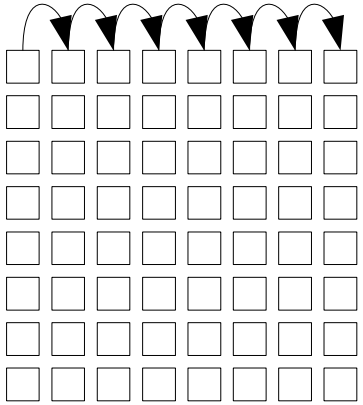
```
for (int j=0; j<n; ++j)
  int k = j*n;
  matrix_sum<<n,1>>(dA+k, dB+k, dC+k, n,1 );
```

- Kernel launch for all rows and columns at once

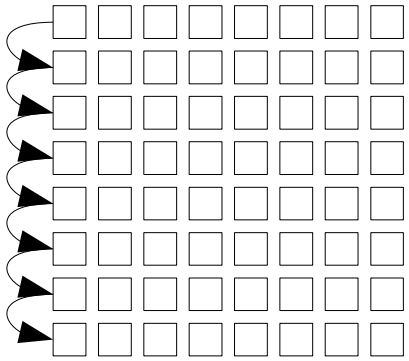
```
matrix_sum<<n,n>>(dA, dB, dC, n,n );
```

  - Single point to incur kernel-launch overhead
  - Might run into hardware limits on threads, thread blocks, and their dimensions

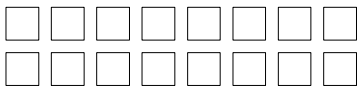
Ordering Matrix Elements in Memory



Row-major order in C/C++



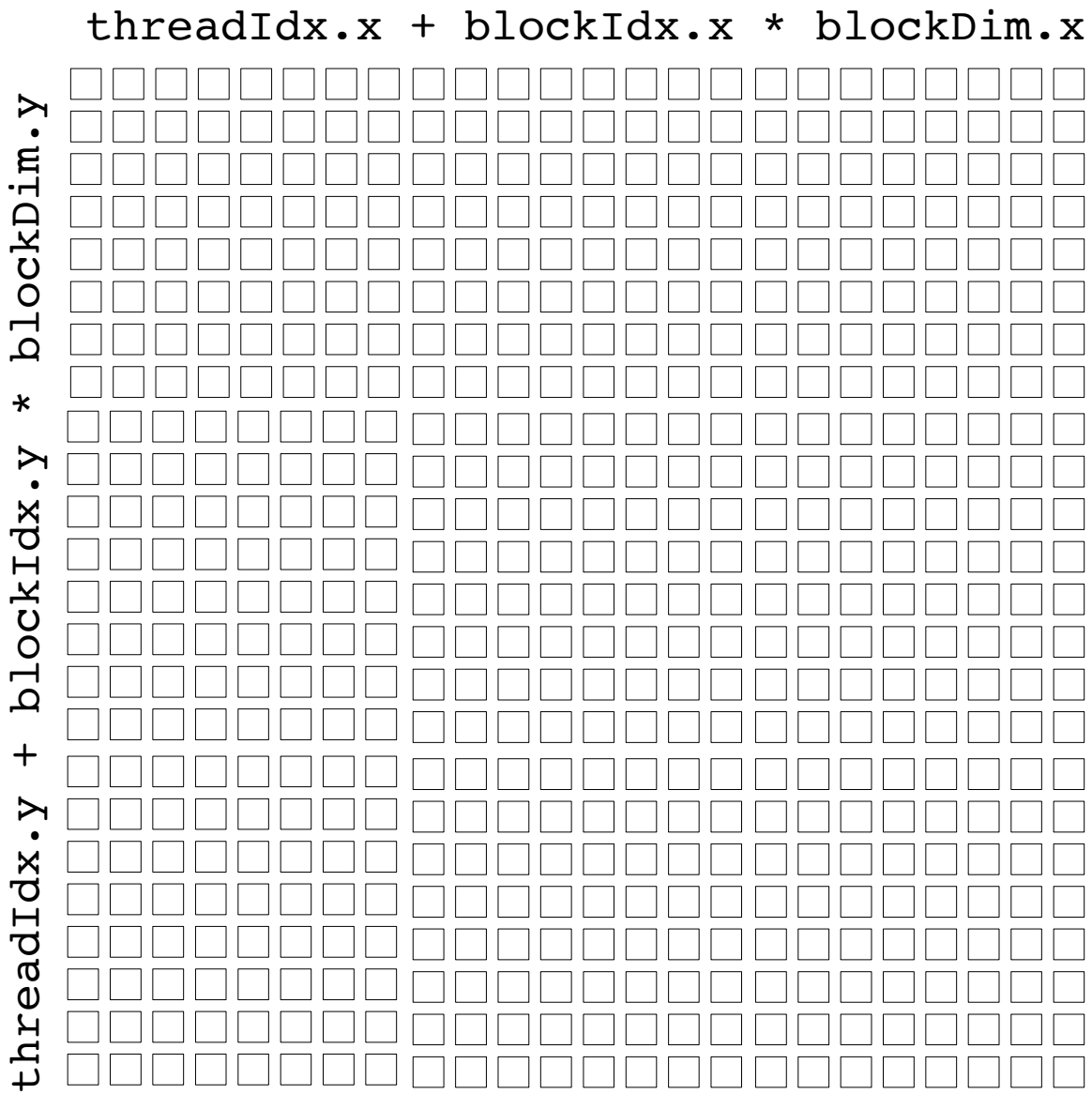
Column-major order in Fortran



```
A + 0x0
A + 0x40
A + 0x80
A + 0xC0
A + 0x100
```

```
A + 0x0
A + 0x4
A + 0x8
A + 0xC
A + 0x10
A + 0x14
```

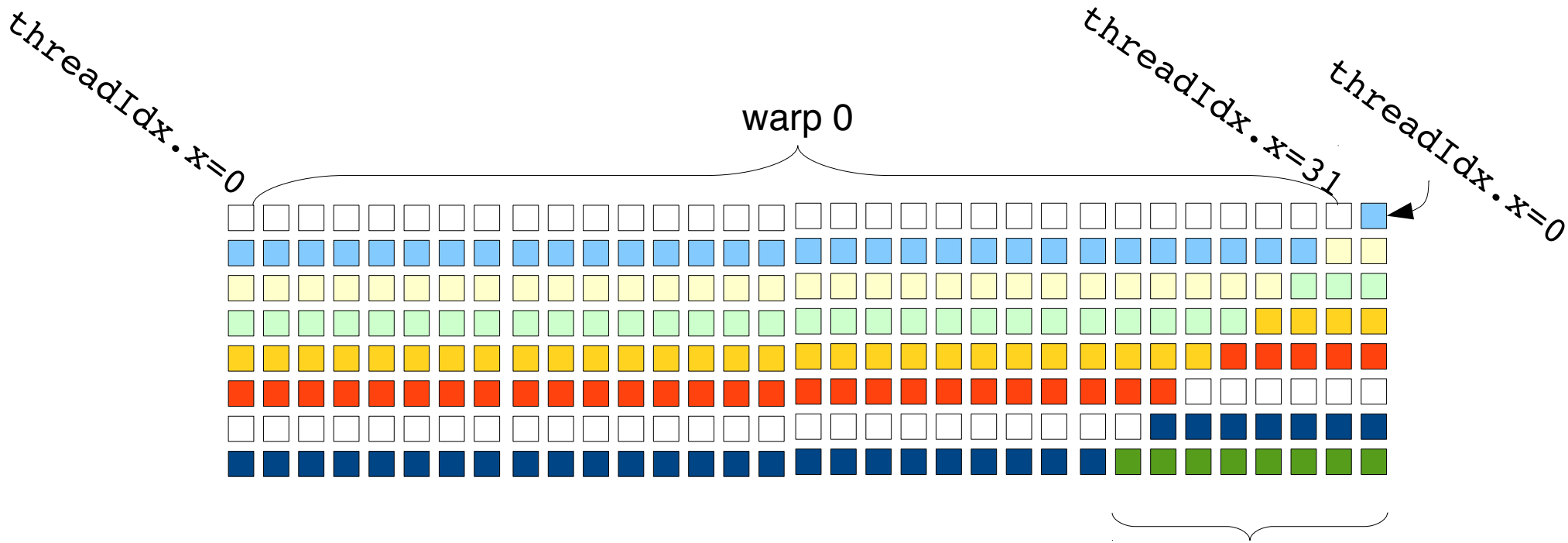
Mapping Threads to Matrix Elements



Scheduling Thread on a GPU

- Programming model for GPUs is SIMT
  - Many threads (ideally) execute the same instruction on different data
  - Performance drops quickly if threads don't execute the same instruction
- Basic unit of scheduling is called a warp
  - The size of warp is (and has been) 32 threads
  - If one of the threads in a warp stalls then entire warp is de-scheduled and another warp is picked
  - Threads are assigned to warp with x-dimension changing fastest
- Some operations can only be performed on half-warp
  - Some GPU cards only have 16 load/store units per SM
  - Each half-warp in a full warp will be scheduled to issue a load one after the other

Mapping Threads In a Block to Warps



Remaining threads in the block will be mapped to an incomplete warp.

GPU Warp Scheduling Basics

- GPU has at least one warp scheduler per SM
- The scheduler picks an eligible warp and executes all threads in the warp
- If any of the threads in the executing warp stalls (uncached memory read) the schedule makes it inactive
- If there are no eligible warps left then GPU idles
- Context switch between warps is fast
  - About 1 or 2 cycles (1 nano-second on 1 GHz GPU)
  - The whole thread block has resources allocated on an SM by the compiler ahead of time