

GPU Programming with OpenMP

Part 3: Memory accessing

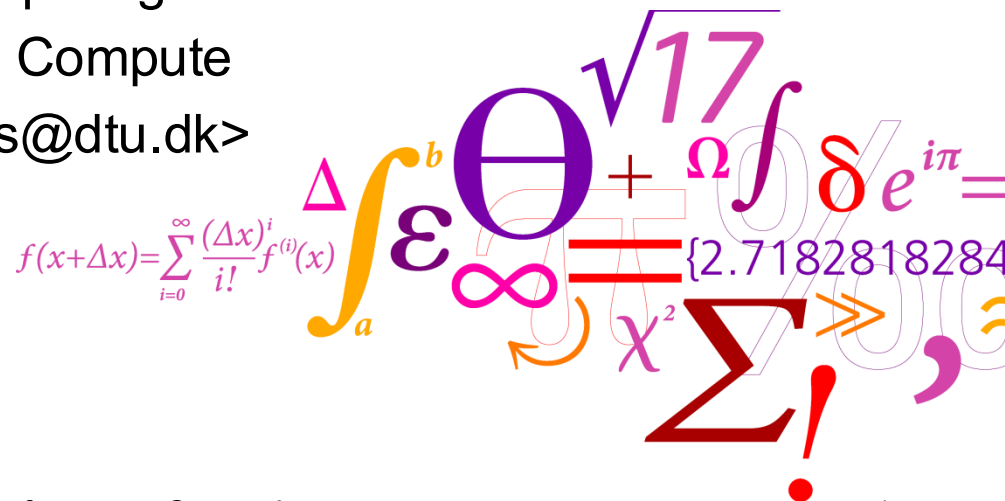


Hans Henrik Brandenburg Sørensen

DTU Computing Center

DTU Compute

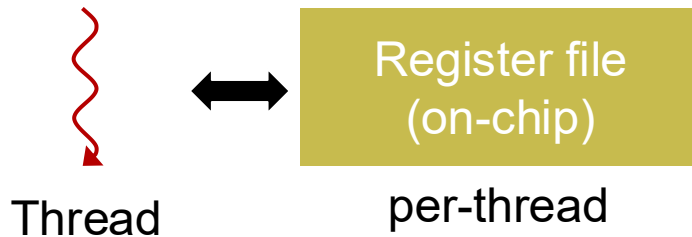
<hhbs@dtu.dk>



Overview

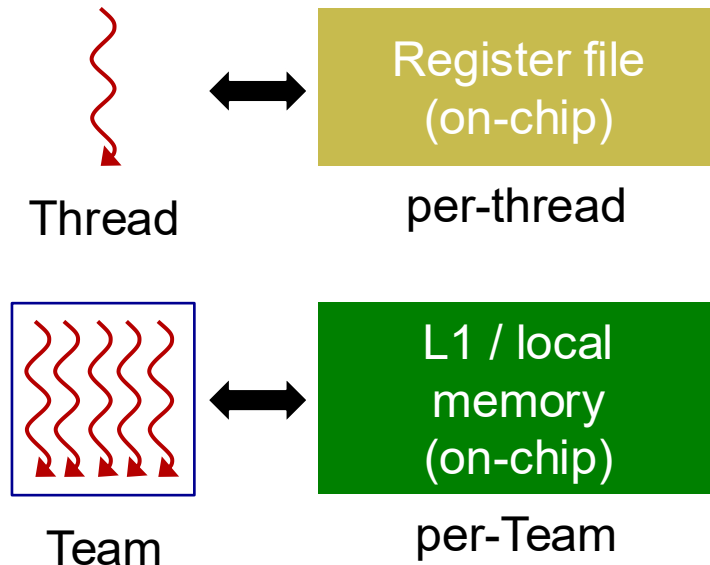
- GPU memory model
- Coalesced memory accesses
 - Adjacent in memory
 - Misaligned memory
 - Strided in memory
- Transpose example (cont.)
- Unified memory

GPU memory model



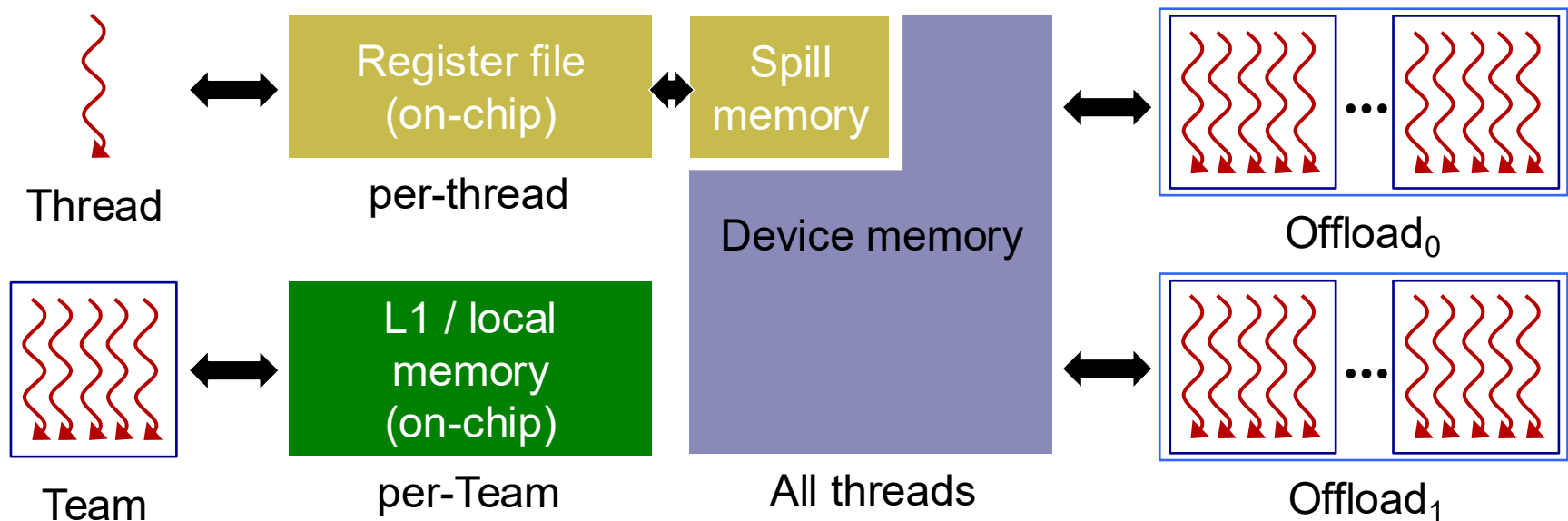
	Register file
Size	256KB / SM
Speed	N/A

GPU memory model



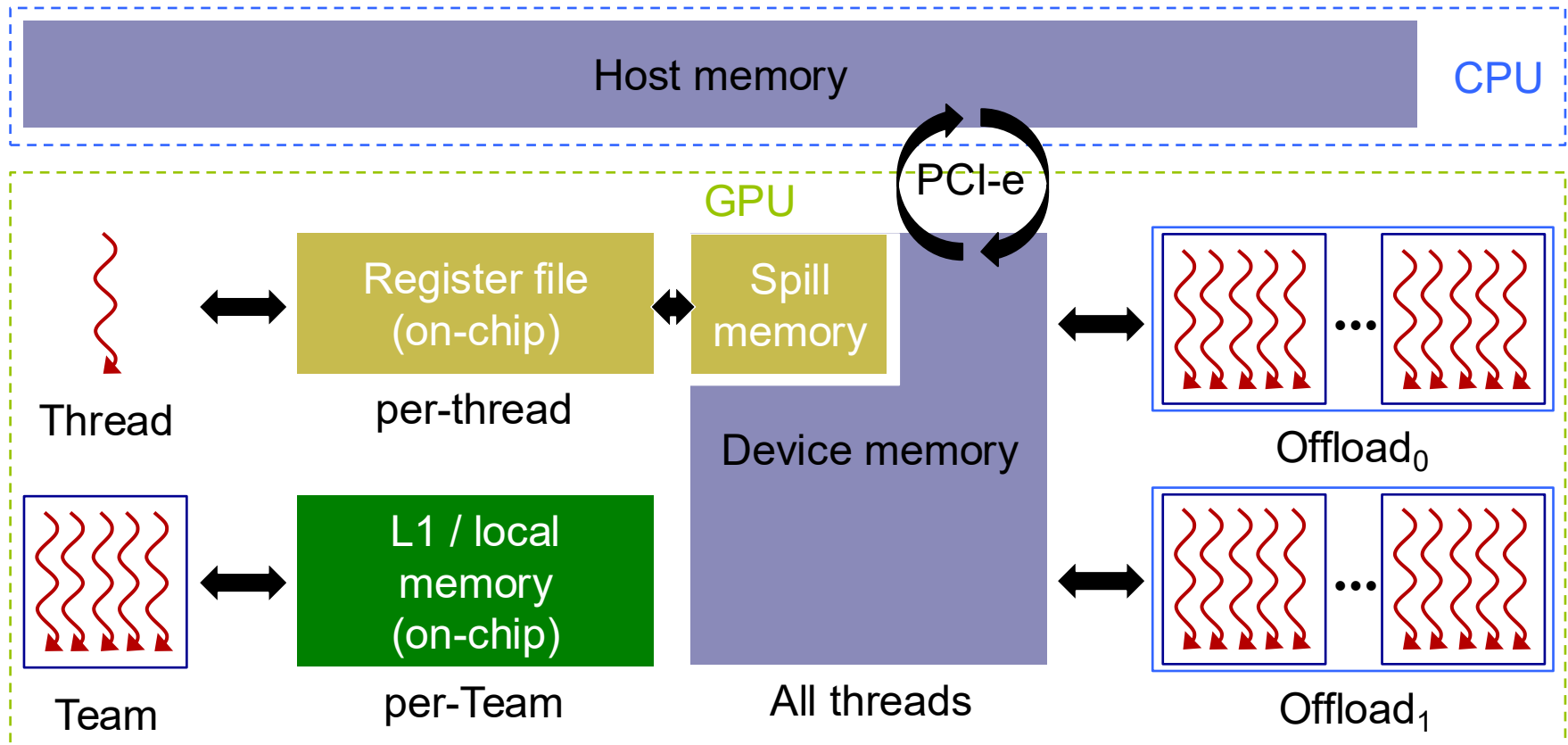
	Register file	L1 / local
Size	256KB / SM	Up to 224 KB
Speed	N/A	>10TB/s aggr.

GPU memory model



	Register file	L1 / local	Device / spill
Size	256KB / SM	Up to 224 KB	16 – 80 GB
Speed	N/A	>10TB/s aggr.	2039 GB/s

GPU memory model



	Register file	L1 / local	Device / spill	CPU "Host"
Size	256KB / SM	Up to 224 KB	16 – 80 GB	192 – 768 GB
Speed	N/A	>10TB/s aggr.	2039 GB/s	< 8 – 32 GB/s

Register memory limitations

■ Hardware limits

Query	Compute Capability		
	1.x (Tesla)	2.x (Fermi)	3.x (Kepler) – 9.x (Hopper)
Max 32-bit registers per thread	128	63	255
Max 32-bit registers per team	8192	32768	65536

- If all registers are used we spill into device memory
 - ❑ Variables are quickly cached in L1 and still fast to use
- Avoid large / complicated code in offload regions!

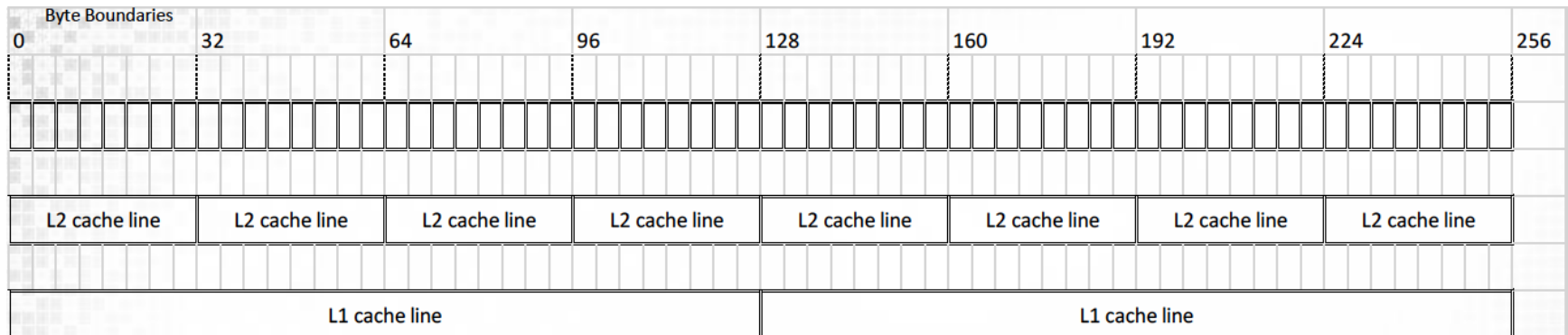
Coalesced memory accesses

Coalesced memory access

“Perhaps the single most important performance consideration in programming for CUDA-capable GPU architectures is the coalescing of global memory accesses.”, CUDA Best Practices, ch. 6.2.1

Memory hierarchy

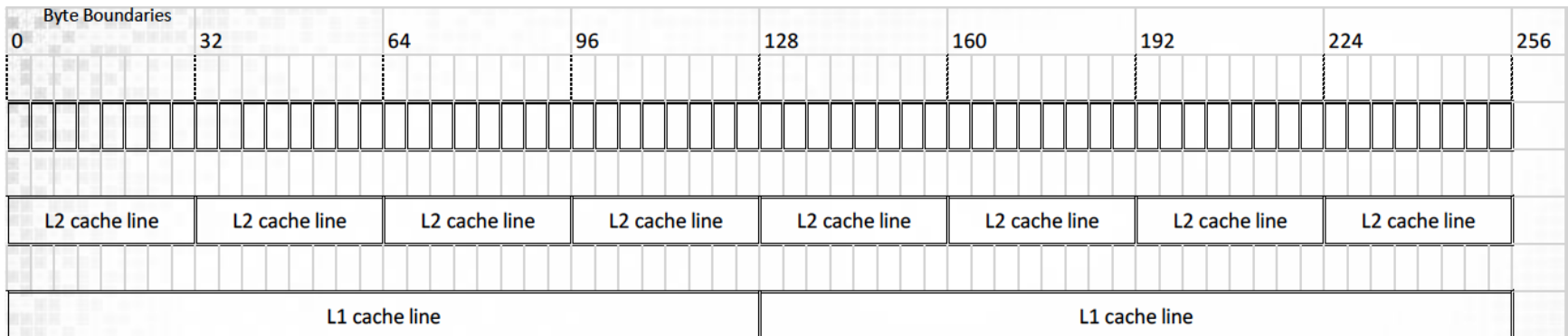
■ Recap from week 1 (now with GPUs)



■ When memory is loaded/stored to global memory through L2 (and L1) it is moved in cache lines

Memory hierarchy

■ Recap from week 1 (now with GPUs)



- When memory is loaded/stored to global memory through L2 (and L1) it is moved in cache lines
 - L1 cache (256 KB per SM)
 - 128 B wide cache line transactions
 - L2 cache (50 MB) (everything comes through here)
 - 32 B wide cache line transactions

Coalesced memory access

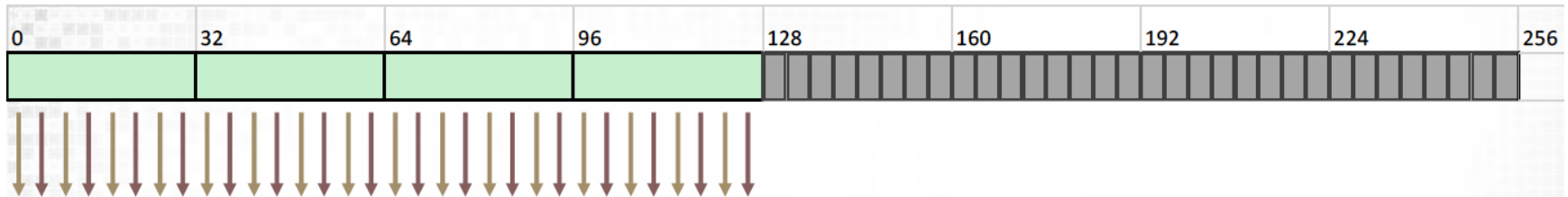
■ Coalesced access

Webster's: **Coalesce** = “to come together so as to form one whole”.

- ❑ Coalesced access is where 32 adjacent threads in a team access sequentially adjacent 4 byte words (e.g. `float` or `int` values)
- ❑ Having coalesced accesses will reduce the number of cache lines moved and optimize memory performance

Coalesced memory access (L2)

■ Aligned and adjacent access

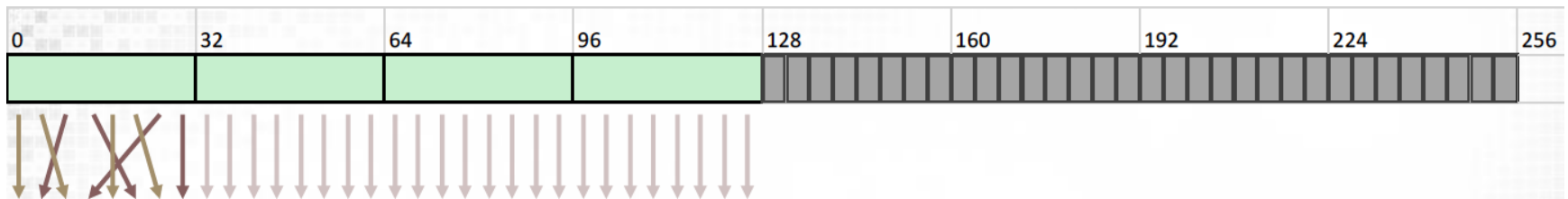


```
void copy(float *dst, float *src, int n) {  
    #pragma omp target teams loop is_device_ptr(dst, src)  
    for (int i = 0; i < n; i++)  
        dst[i] = src[i];  
}
```

- ❑ For a coalesced read/write within a warp, 4 transactions are required
- ❑ 100% memory bus utilization

Coalesced memory access (L2)

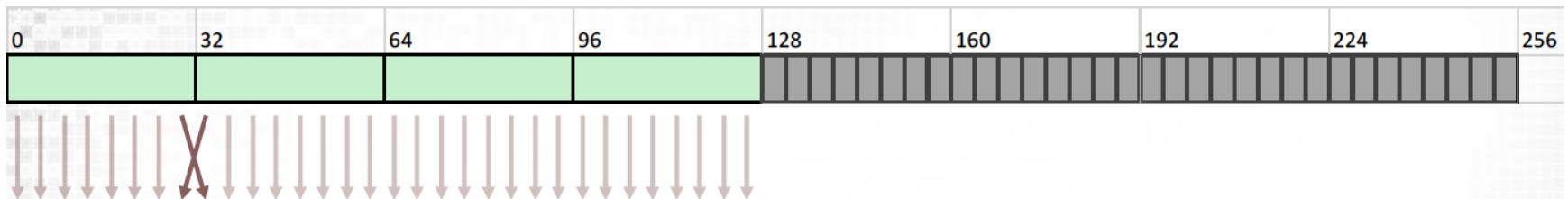
■ Permuted access



- ❑ Within the same cache line accesses can be permuted between threads
- ❑ 100% memory bus utilization

Coalesced memory access (L2)

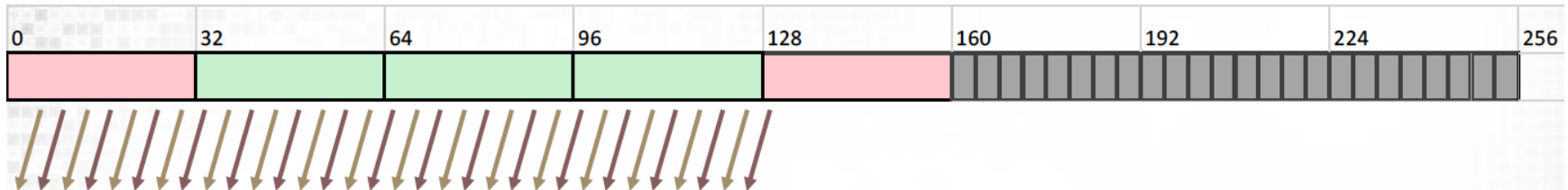
■ Permuted access over 32 byte segments



- ❑ Permuted access within the 128 byte segments is ok!
- ❑ 100% memory bus utilization
- ❑ Must not cross 128 byte boundary

Coalesced memory access (L2)

■ Misaligned access

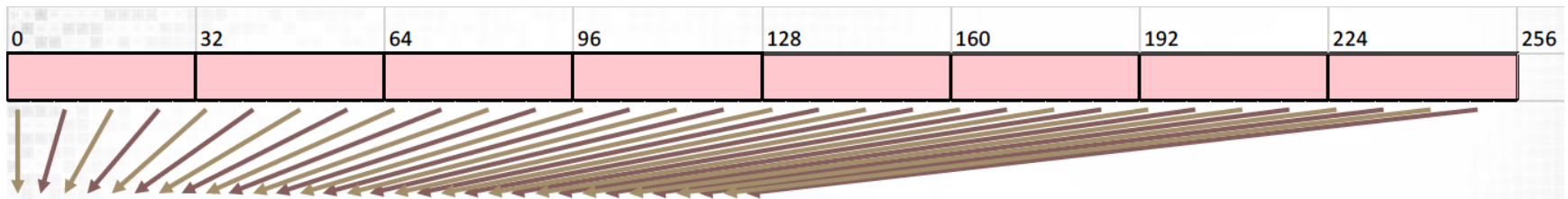


```
void offsetCopy(float *dst, float *src, int n, int offset) {  
    #pragma omp target teams loop is_device_ptr(dst, src)  
    for (int i = offset; i < n; i++)  
        dst[i] = src[i];  
}
```

- ❑ If memory accesses are misaligned (offset) then parts of the cache line might be unused (shown in red)
- ❑ 5 transactions of total 160 bytes of which 128 bytes is required: 80% memory bus utilization

Coalesced memory access (L2)

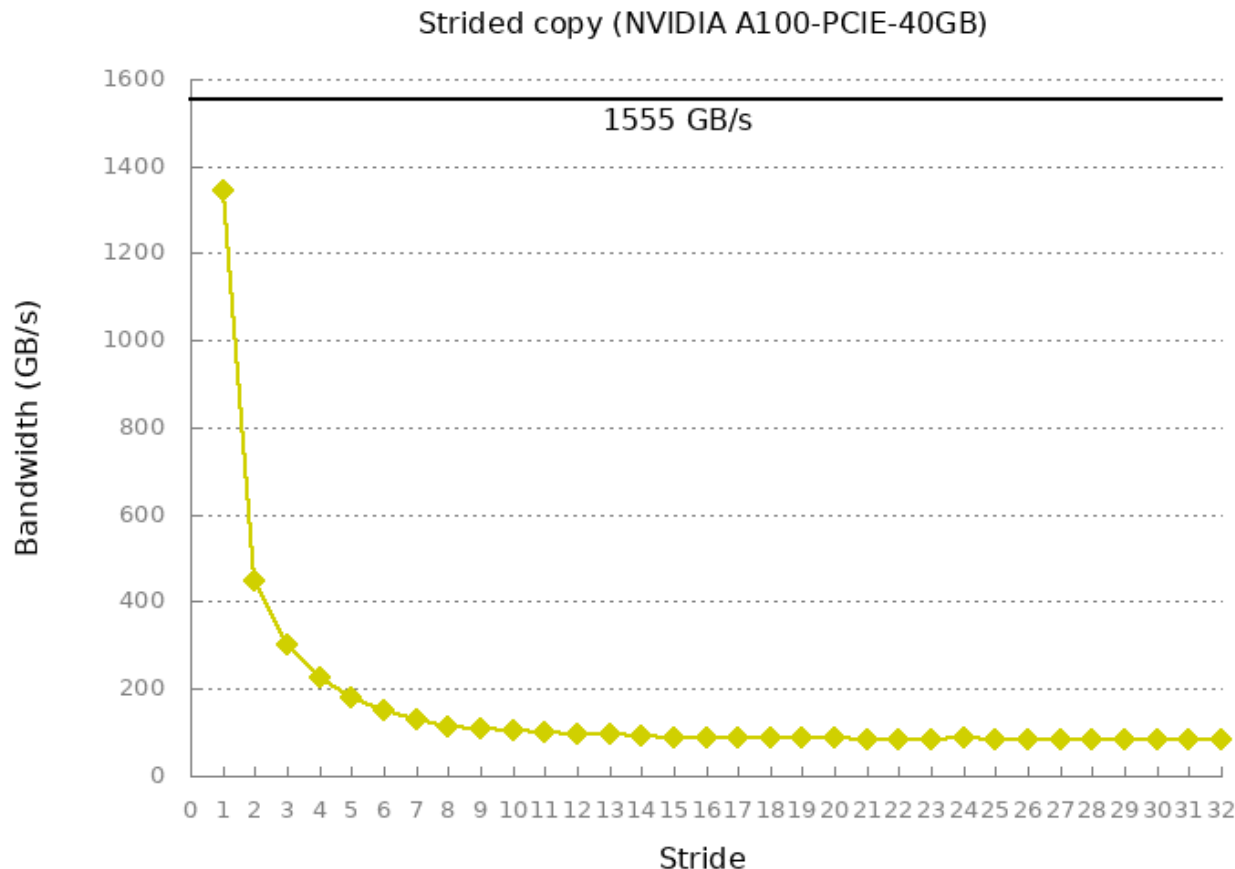
■ Strided access



```
void stridedCopy(float *dst, float *src, int n, int stride) {  
    #pragma omp target teams loop is_device_ptr(dst, src)  
    for (int i = 0; i < n; i += stride)  
        dst[i] = src[i];  
}
```

- ❑ If memory accesses are strided then large parts of the cache line might be unused
- ❑ A stride of 2 causes 8 transactions: 50% memory bus utilization if the interleaved bytes are not used

Coalesced memory access (L2)



- ❑ A stride of >32 causes 32 transactions: ONLY 3.125% bus utilization! This corresponds to random access.

Example: Matrix in C

Row major
storage

$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$
$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$
$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$
$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$

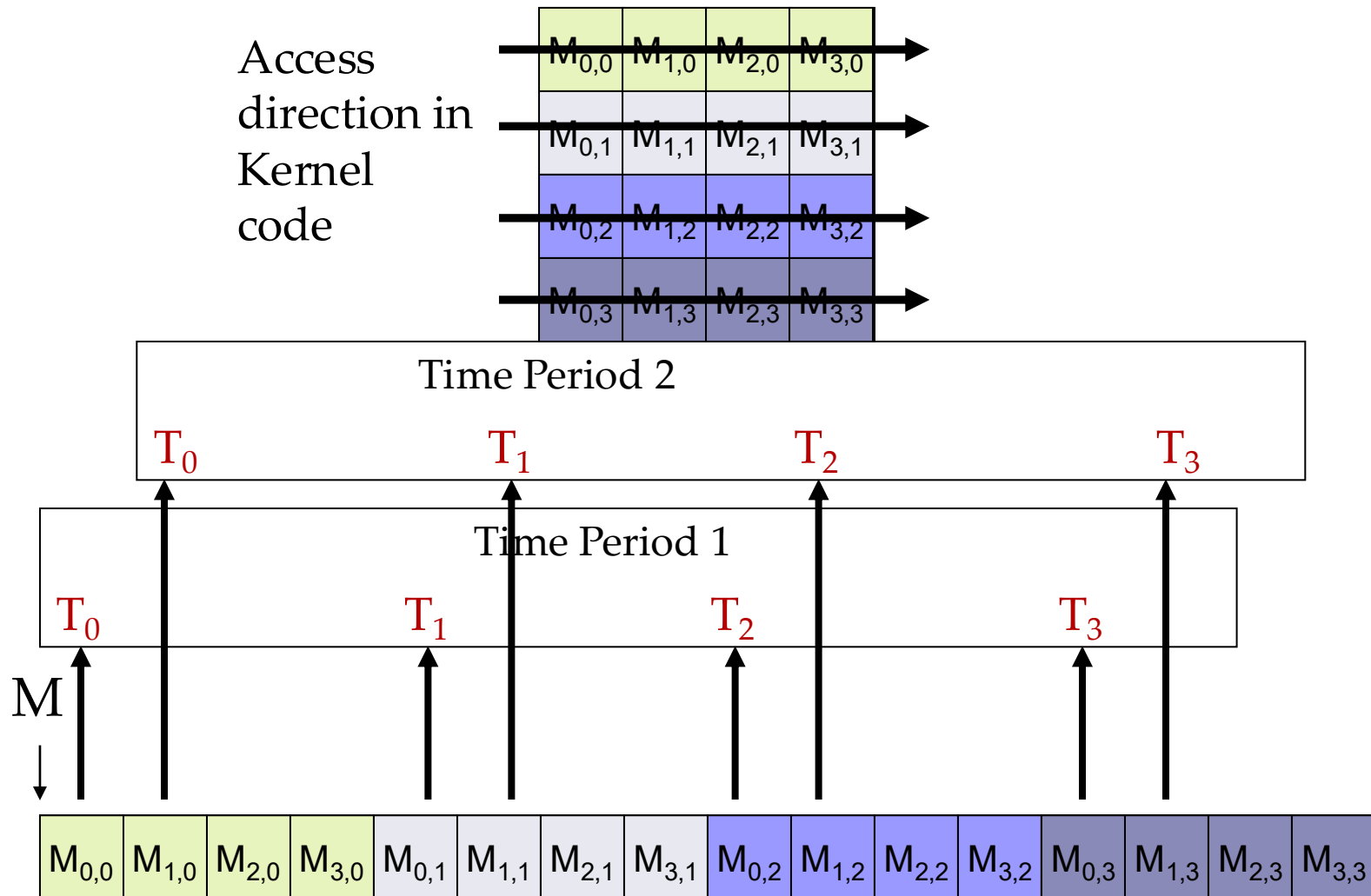
M



$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$	$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$	$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$	$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$
-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------

From David Kirk/NVIDIA and Wen-mei W. Hwu

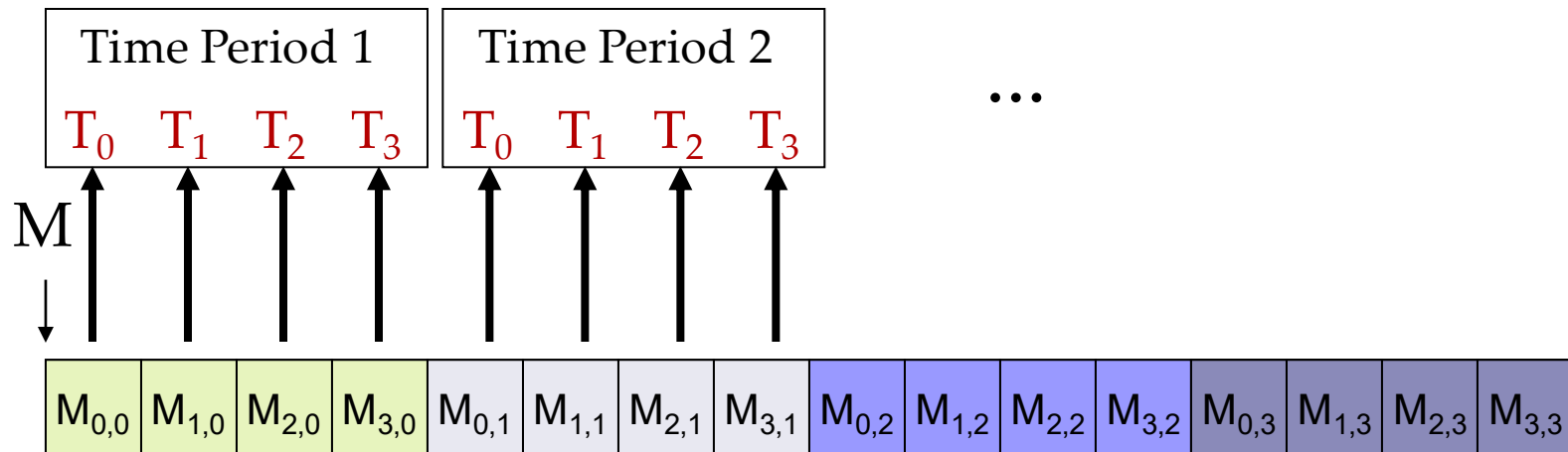
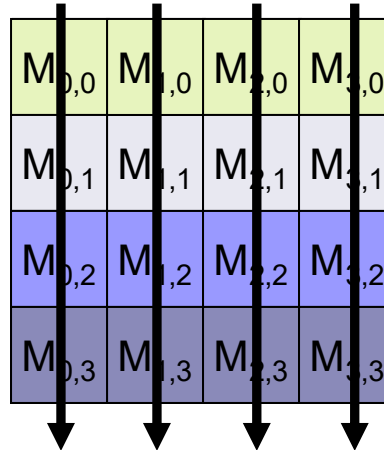
Example: Matrix in C



From David Kirk/NVIDIA and Wen-mei W. Hwu

Example: Matrix in C

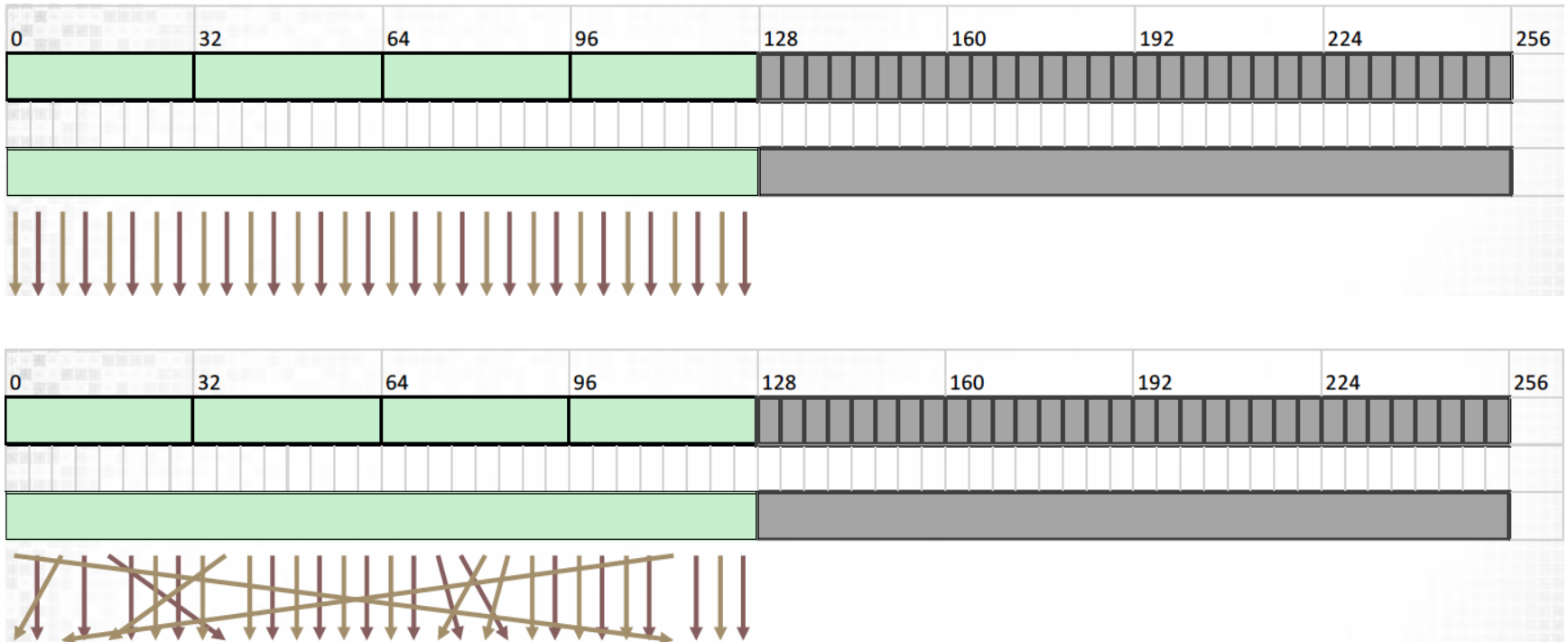
Access
direction in
Kernel
code



From David Kirk/NVIDIA and Wen-mei W. Hwu

Coalesced memory access (L1)

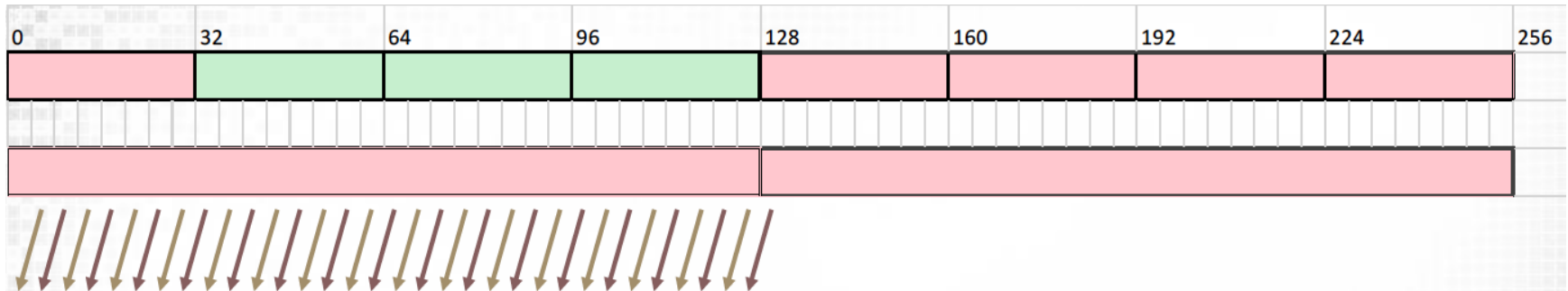
- All accesses fall inside 128 byte cache line



- Single transaction
- 100% memory bus utilization

Coalesced memory access (L1)

■ Misaligned or strided access



- ❑ If memory accesses are misaligned (offset) then parts of the cache line might be unused (shown in red)
- ❑ 2 transactions of total 256 bytes of which 128 bytes is required: 50% memory bus utilization
- ❑ Strided access has the same degradation as for L2

Transpose example

Transpose example

- How well are we doing?

Version	v1 seq	v2 per col	v3 per elm	v4 warp
Time [ms]	12305	4.19	0.78	0.67

Transpose example

■ How well are we doing?

- ❑ Theoretical peak bandwidth = 2039 GB/s
- ❑ Achieved bandwidth = $2 * N^2 * 8 / 10^9 / \text{runtime}$
- ❑ Memory utilization: $100 \% * (\text{Achieved} / \text{Peak})$

Version	v1 seq	v2 per col	v3 per elm	v4 warp
Time [ms]	12305	4.19	0.78	0.67
Memory utilization	< 0.01 %	9.97 %	53.6 %	62.3 %

Transpose example (v4 warp)

■ Why are we not doing better?

```
// OpenMP offload transpose using one warp per col
void transpose_warp_per_col(double **A, double **At)
{
    #pragma omp target teams loop \
        num_teams(N) thread_limit(32)
    for(int i = 0; i < N; i++) {
        #pragma omp loop bind(parallel)
        for(int j = 0; j < N; j++)
            At[i][j] = A[j][i];
    }
}
```

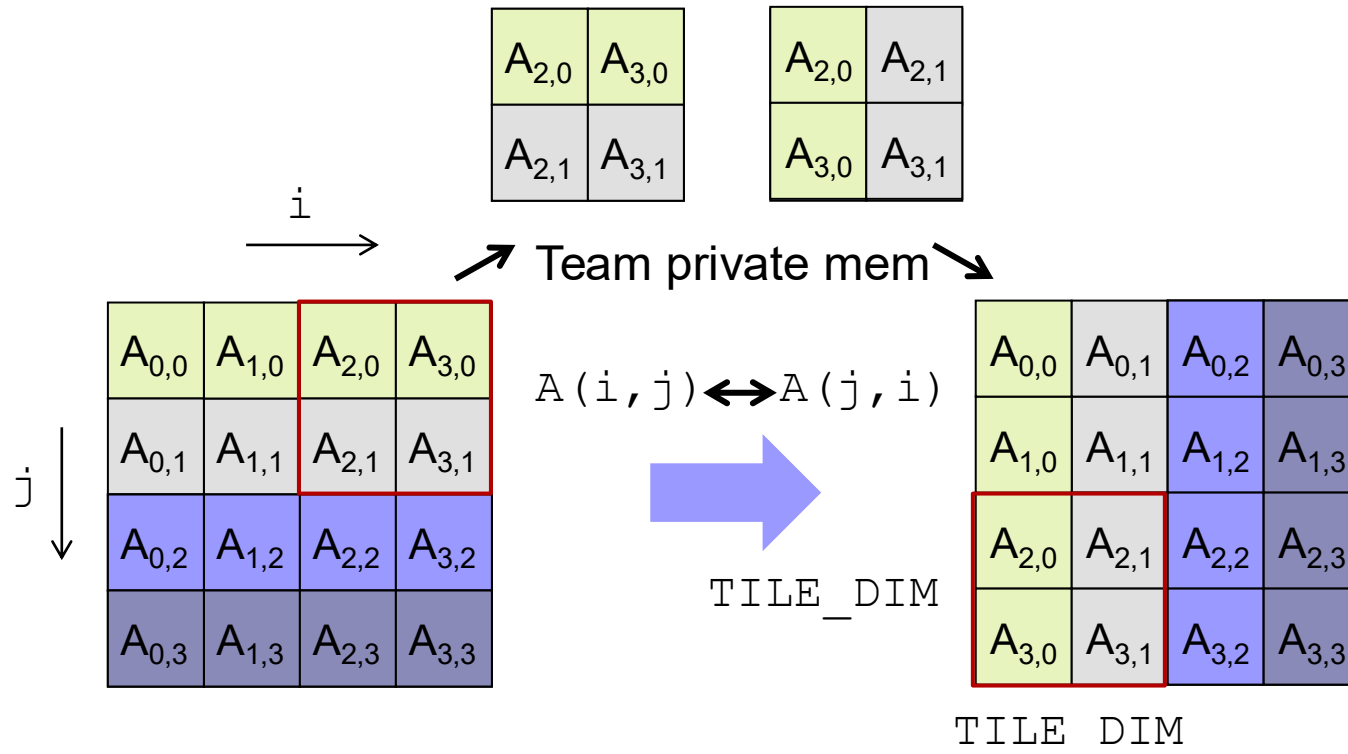
Transpose example (v4 warp)

■ Why are we not doing better?

```
// OpenMP offload transpose using one warp per col
void transpose_warp_per_col(double **A, double **At)
{
    #pragma omp target teams loop \
        num_teams(N) thread_limit(32)
    for(int i = 0; i < N; i++) {
        #pragma omp loop bind(parallel)
        for(int j = 0; j < N; j++)
            At[i][j] = A[j][i];
    }
}
```

Coalesced writes Strided reads

Transpose example (v5 tiled)



- We read a tile from A into team private memory (coalesced) and write from team private memory to A^T (coalesced). Strides are in private memory.

Transpose example (v5 tiled)

```
// OpenMP offload transpose using one warp per tile and team local memory (TILE_DIM is 16)
void transpose_tiled(double **A, double **At)
{
    #pragma omp target teams loop bind(teams) collapse(2) is_device_ptr(A, At) \
        num_teams((N/TILE_DIM)*(N/TILE_DIM)) thread_limit(32)
    for (int ytile = 0; ytile < N; ytile += TILE_DIM)
        for (int xtile = 0; xtile < N; xtile += TILE_DIM) {
            double tile[TILE_DIM][TILE_DIM+1];
            #pragma omp allocate(tile) allocator(omp_pteam_mem_alloc)
            #pragma omp loop bind(parallel) collapse(2)
            for (int y = 0; y < TILE_DIM; y++)
                for (int x = 0; x < TILE_DIM; x++) {
                    int jt = y + xtile;
                    int it = x + ytile;
                    tile[x][y] = A[jt][it];
                }
            #pragma omp loop bind(parallel) collapse(2)
            for (int y = 0; y < TILE_DIM; y++)
                for (int x = 0; x < TILE_DIM; x++) {
                    int j = x + xtile;
                    int i = y + ytile;
                    At[i][j] = tile[y][x];
                }
        }
}
```

Transpose example (v5 tiled)

```
// OpenMP offload transpose using one warp per tile and team local memory (TILE_DIM is 16)
void transpose_tiled(double **A, double **At)
{
    #pragma omp target teams loop bind(teams) collapse(2) is_device_ptr(A, At) \
        num_teams((N/TILE_DIM)*(N/TILE_DIM)) thread_limit(32)
    for (int ytile = 0; ytile < N; ytile += TILE_DIM)
        for (int xtile = 0; xtile < N; xtile += TILE_DIM) {
            double tile[TILE_DIM][TILE_DIM+1];
            #pragma omp allocate(tile) allocator(omp_pteam_mem_alloc)
            #pragma omp loop bind(parallel) collapse(2)
            for (int y = 0; y < TILE_DIM; y++)
                for (int x = 0; x < TILE_DIM; x++) {
                    int jt = y + xtile;
                    int it = x + ytile;
                    tile[x][y] = A[jt][it];
                }
            #pragma omp loop bind(parallel) collapse(2)
            for (int y = 0; y < TILE_DIM; y++)
                for (int x = 0; x < TILE_DIM; x++) {
                    int j = x + xtile;
                    int i = y + ytile;
                    At[i][j] = tile[y][x];
                }
        }
}
```

Not currently
supported in nvc++

Transpose example (v5 tiled 'acc')

```
// OpenACC offload transpose using one warp per tile and team local memory (TILE_DIM is 16)
void transpose_tiled(double **A, double **At)
{
    #pragma acc parallel loop gang collapse(2) deviceptr(A, At) \
        num_gangs((N/TILE_DIM)*(N/TILE_DIM)) vector_length(32)
    for (int ytile = 0; ytile < N; ytile += TILE_DIM)
        for (int xtile = 0; xtile < N; xtile += TILE_DIM) {
            double tile[TILE_DIM][TILE_DIM+1];
            #pragma acc cache(tile)
            #pragma acc loop vector collapse(2)
            for (int y = 0; y < TILE_DIM; y++)
                for (int x = 0; x < TILE_DIM; x++) {
                    int jt = y + xtile;
                    int it = x + ytile;
                    tile[x][y] = A[jt][it];
                }
            #pragma acc loop vector collapse(2)
            for (int y = 0; y < TILE_DIM; y++)
                for (int x = 0; x < TILE_DIM; x++) {
                    int j = x + xtile;
                    int i = y + ytile;
                    At[i][j] = tile[y][x];
                }
        }
}
```


Transpose example (v5 tiled)

```
$ nvc++ -fast -Msafeptr -Minfo -mp=gpu -gpu=cc90 -acc -c -o transpose.o transpose.cpp
...
```

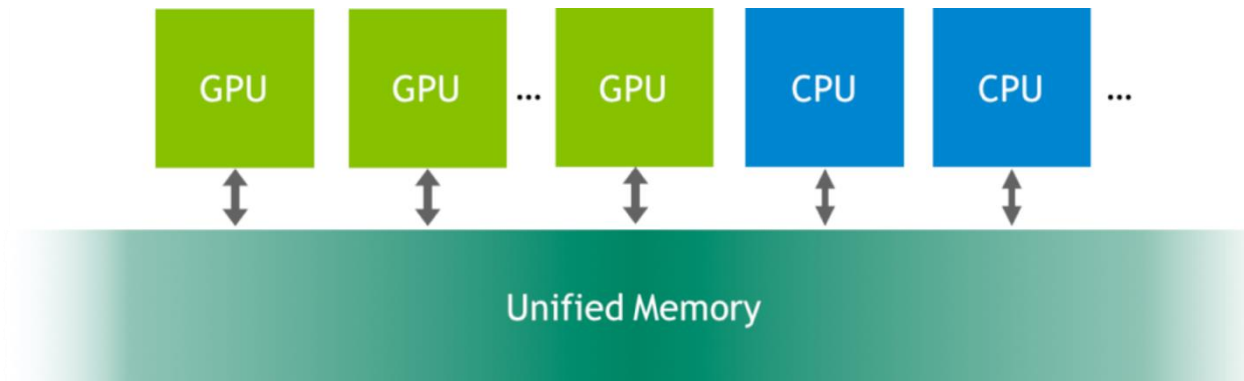
```
transpose_tiled(double **, double **, long):
  103, Generating NVIDIA GPU code
    109, #pragma acc loop gang(207936) collapse(2) /* blockIdx.x */
    110, /* blockIdx.x collapsed */
    114, #pragma acc loop vector(32) collapse(2) /* threadIdx.x */
    115, /* threadIdx.x collapsed */
    121, #pragma acc loop vector(32) collapse(2) /* threadIdx.x */
    122, /* threadIdx.x collapsed */
  103, CUDA shared memory used for tile
  109, Loop not vectorized/parallelized: too deeply nested
  114, Loop is parallelizable
      Loop interchange produces reordered loop nest: 115,114
      Loop not vectorized: unprofitable for target
      Loop unrolled 4 times
```

Time (%)	Total Time (ns)	Instances	Avg (ns)	..	Name
99.7	48,681,626	100	486,816.3	..	transpose_tiled_101(double **, double **, long)
0.3	152,257	2	76,128.5		nvkernel__Zl3malloc_2d_deviiPPd_F1L45_2

Version	v1 seq	v2 per col	v3 per elm	v4 warp	v5 tiled acc
Time [ms]	12305	4.19	0.78	0.67	0.49
Memory utilization	< 0.01 %	9.97 %	53.6 %	62.3 %	85.2 %

Unified shared memory

Unified shared memory



- Runtime creates a pool of managed memory that is shared between the CPU and GPU
- Managed memory is accessible on CPU and GPU from program using the same unique C pointer
- A true “virtual memory” divided into pages
- Data automatically migrates to CPU and GPU

Unified shared memory

■ Advantages

- ❑ Ease of programming
- ❑ Data is migrated on demand – no memCpys
 - Offers the performance of local data on the GPU
 - while providing the ease of use of globally shared data
- ❑ Very efficient with complex data structures (e.g. linked lists, structures with pointers, ...)

■ Disadvantages

- ❑ The physical location of data is invisible to the programmer and may be changed at any time
- ❑ Carefully tuned OpenMP programs that efficiently overlap execution with data transfers may perform better

OpenMP offload syntax

■ Syntax C/C++:

```
#pragma omp requires [clause]
```

■ Clause can be

- ☐ ~~reverse_offload~~
- ☐ unified_address
- ☐ unified_shared_memory
- ☐ ~~atomic_default_mem_order(seq_cst | acq_rel | relaxed)~~
- ☐ ~~dynamic_allocators~~

Not currently
supported in nvc++

- ## ■ Standalone declarative directive that specifies the features that a compiler must provide in order for the code to compile and execute correctly

Unified shared memory

■ `unified_address`

- ❑ A pointer will always refer to the same location in memory from all devices accessible through OpenMP
- ❑ Makes the `is_device_ptr` clause not necessary for device pointers to be translated in target regions

■ `unified_shared_memory`

- ❑ Memory in the device data environment of any device visible to OpenMP, including but not limited to the host, is considered part of the device data environment of all devices
- ❑ Makes the `map` clause optional on target constructs and the `declare target` directive optional

Unified shared memory

```
#pragma omp requires unified_shared_memory

// No data directives or maps needed for pointers a, b, c
#pragma omp target teams distribute parallel for
for (int i = 0; i < N; i++)
    c[i] = a[i] + b[i];
}
```

- ❑ **nvc++:** “The `requires` directive has limited support. The requirement clause `unified_address` is accepted; `unified_shared_memory` is accepted in unified memory mode” (`-gpu=managed`)”
- ❑ **Note:** Using managed memory (`-gpu=managed`) makes it impossible to measure transfer times and execution times separately (don’t use it in this course!)

End of lecture