

CUDA Performance Tuning

Memory Optimization

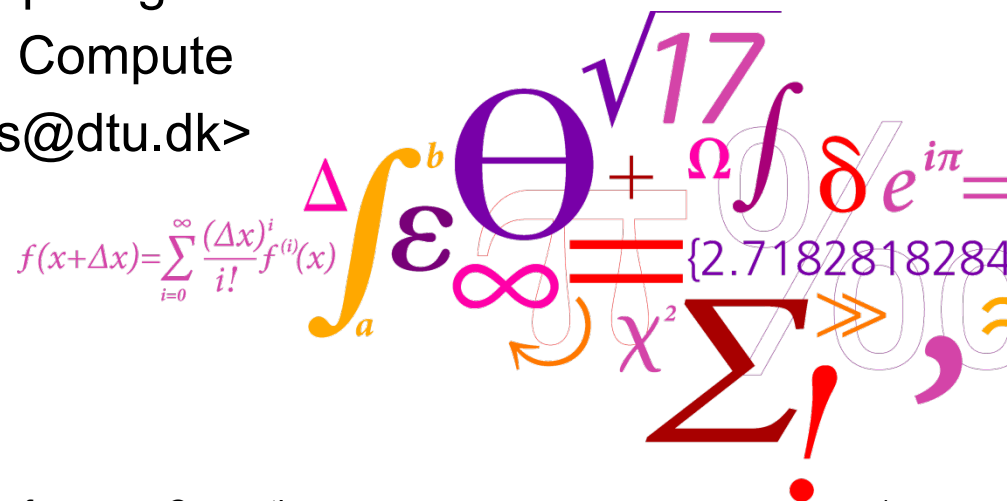


Hans Henrik Brandenborg Sørensen

DTU Computing Center

DTU Compute

<hhbs@dtu.dk>



Overview

- Coalesced memory accesses
 - Adjacent in memory
 - Misaligned memory
 - Strided in memory
- Transpose example (cont.)
- Synchronization
 - Barriers
- Atomic operations
 - Avoiding barriers

Always start here

- ① Minimize memory accesses
- ② Maximize use of fast memory

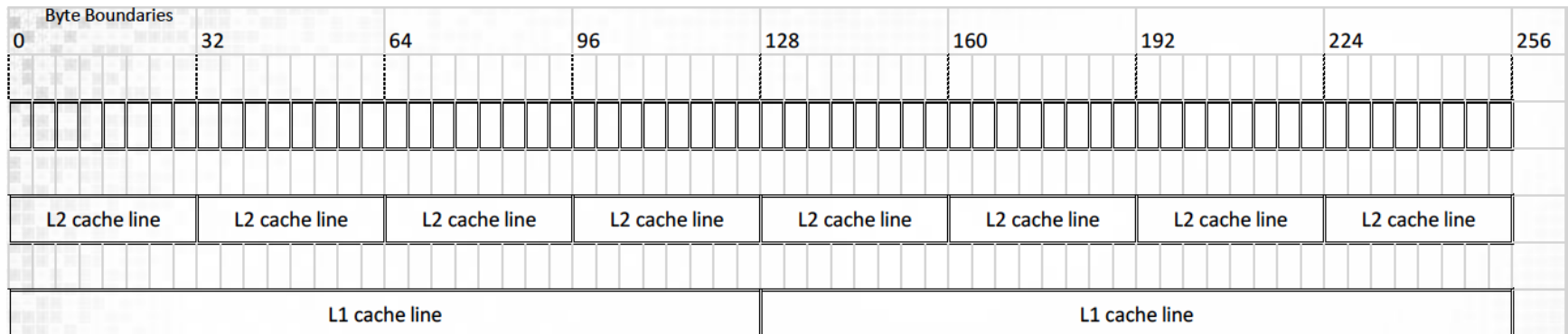
Coalesced memory accesses

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

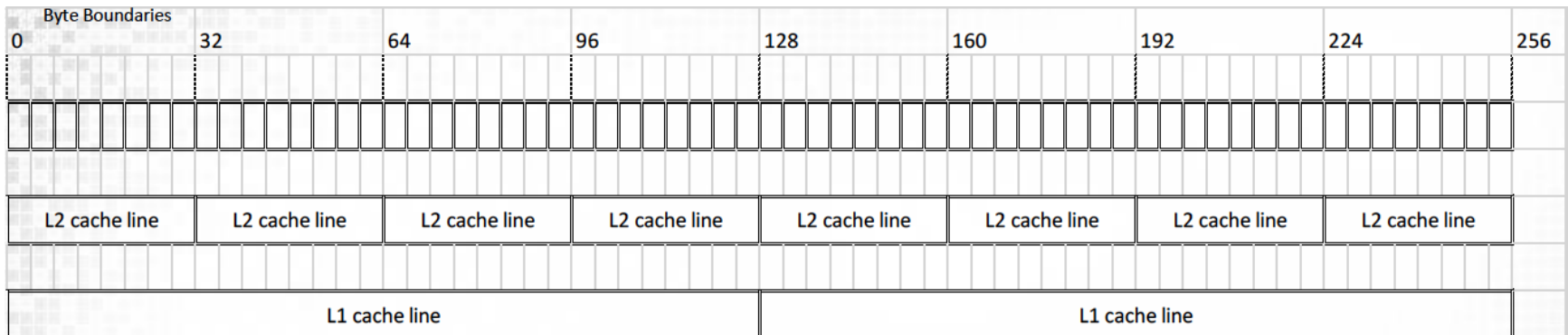
■ Recap from week 1 (now with GPUs)



■ 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 (128 KB per SM)
 - 128 B wide cache line transactions
 - L2 cache (6 MB) (everything comes through here)
 - 32 B wide cache line transactions

Coalesced memory access

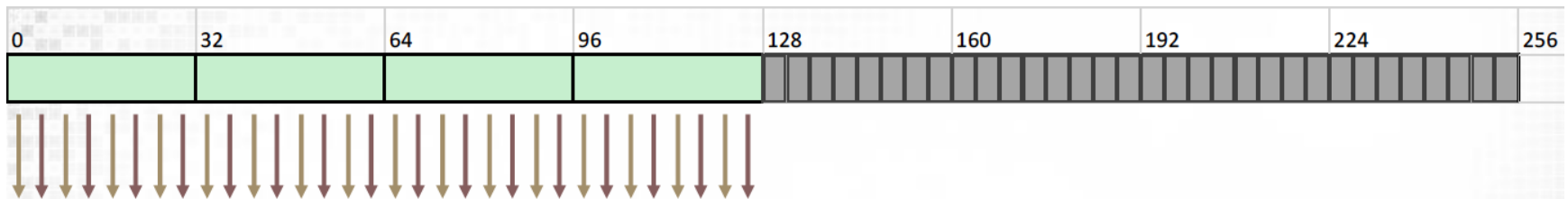
■ Coalesced access

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

- ❑ Coalesced access is where 32 adjacent threads in a warp 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

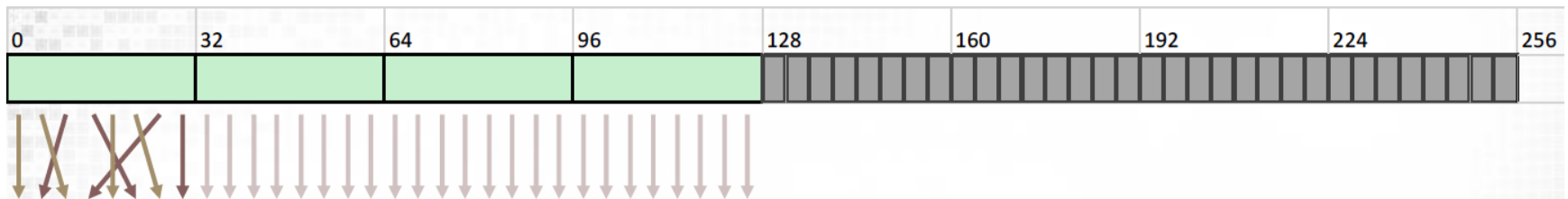


```
__global__ void copy(float *odata, float *idata)
{
    int xid = blockIdx.x * blockDim.x + threadIdx.x;
    odata[xid] = idata[xid];
}
```

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

Coalesced memory access (L2)

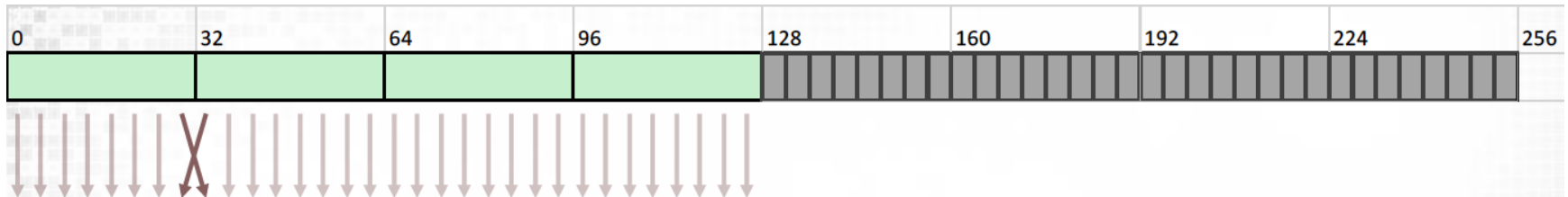
■ Permuted access



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

Coalesced memory access (L2)

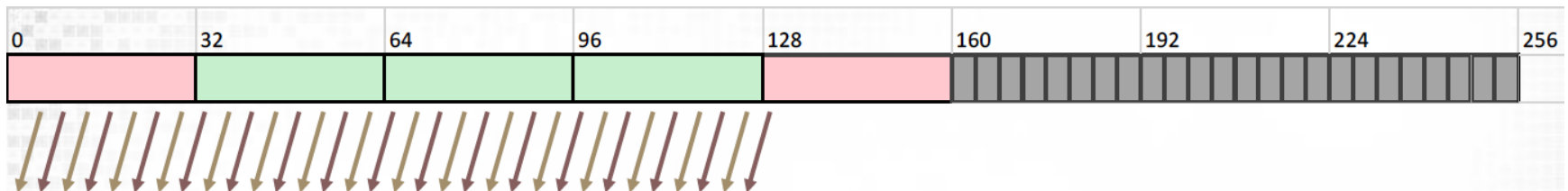
■ Permuted access over 32 byte segments



- ❑ 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

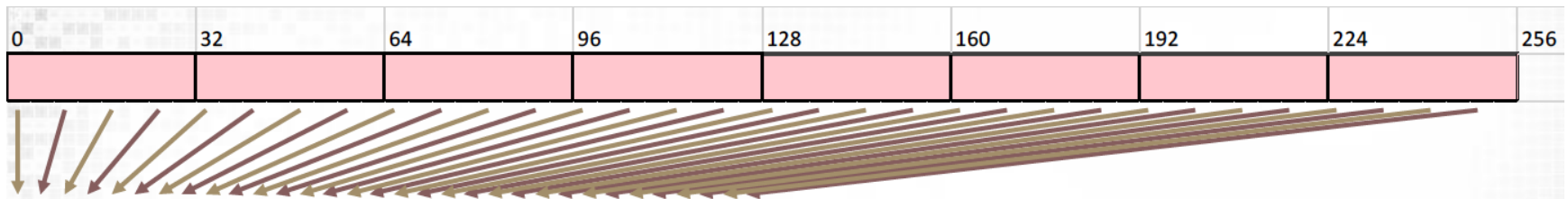


```
__global__ void offsetCopy(float *odata, float* idata, int offset)
{
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
    odata[xid] = idata[xid];
}
```

- ❑ 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)

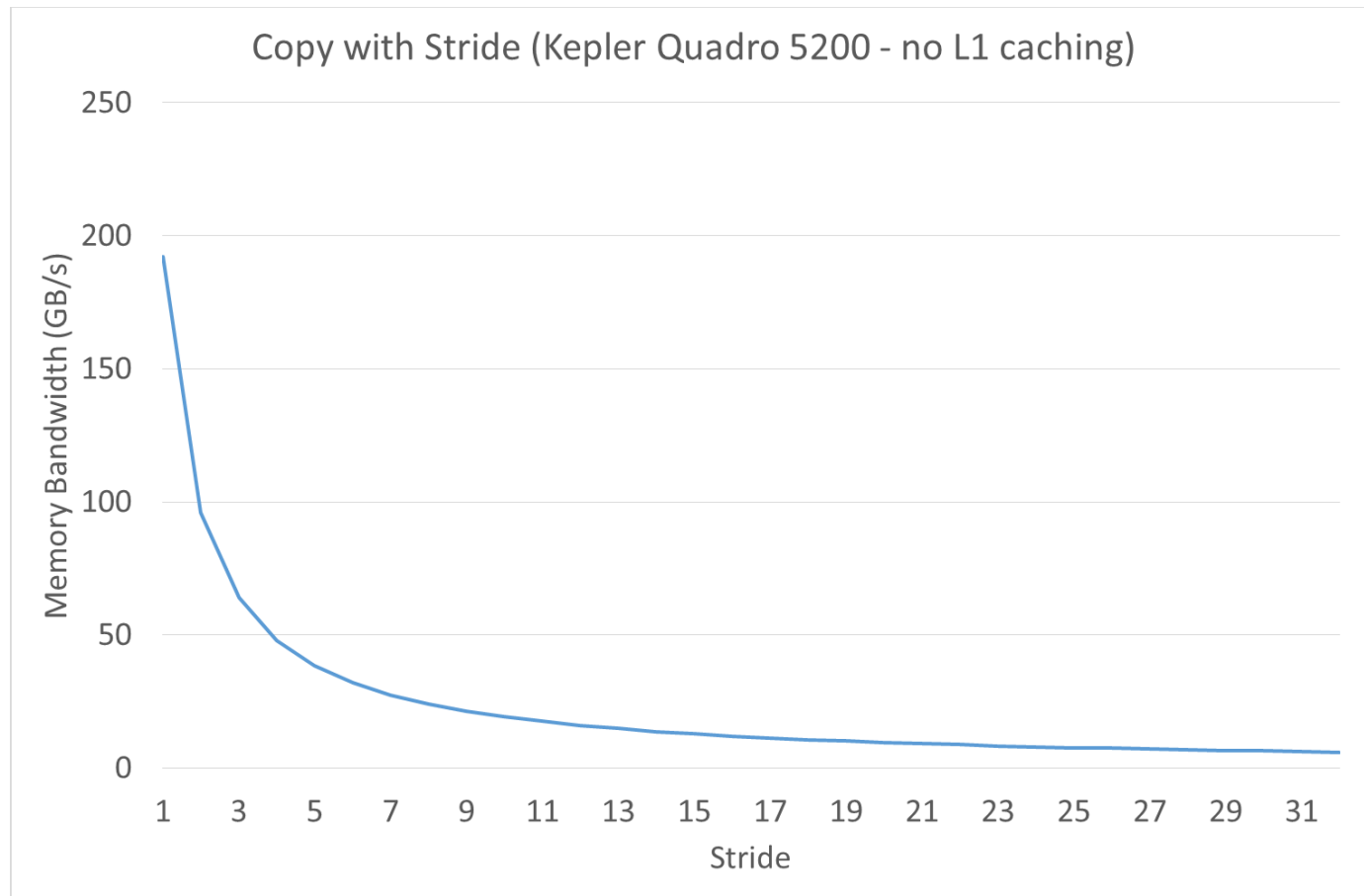
■ Strided access



```
__global__ void strideCopy(float *odata, float* idata, int stride)
{
    int xid = (blockIdx.x * blockDim.x + threadIdx.x) * stride;
    odata[xid] = idata[xid];
}
```

- ❑ 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

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

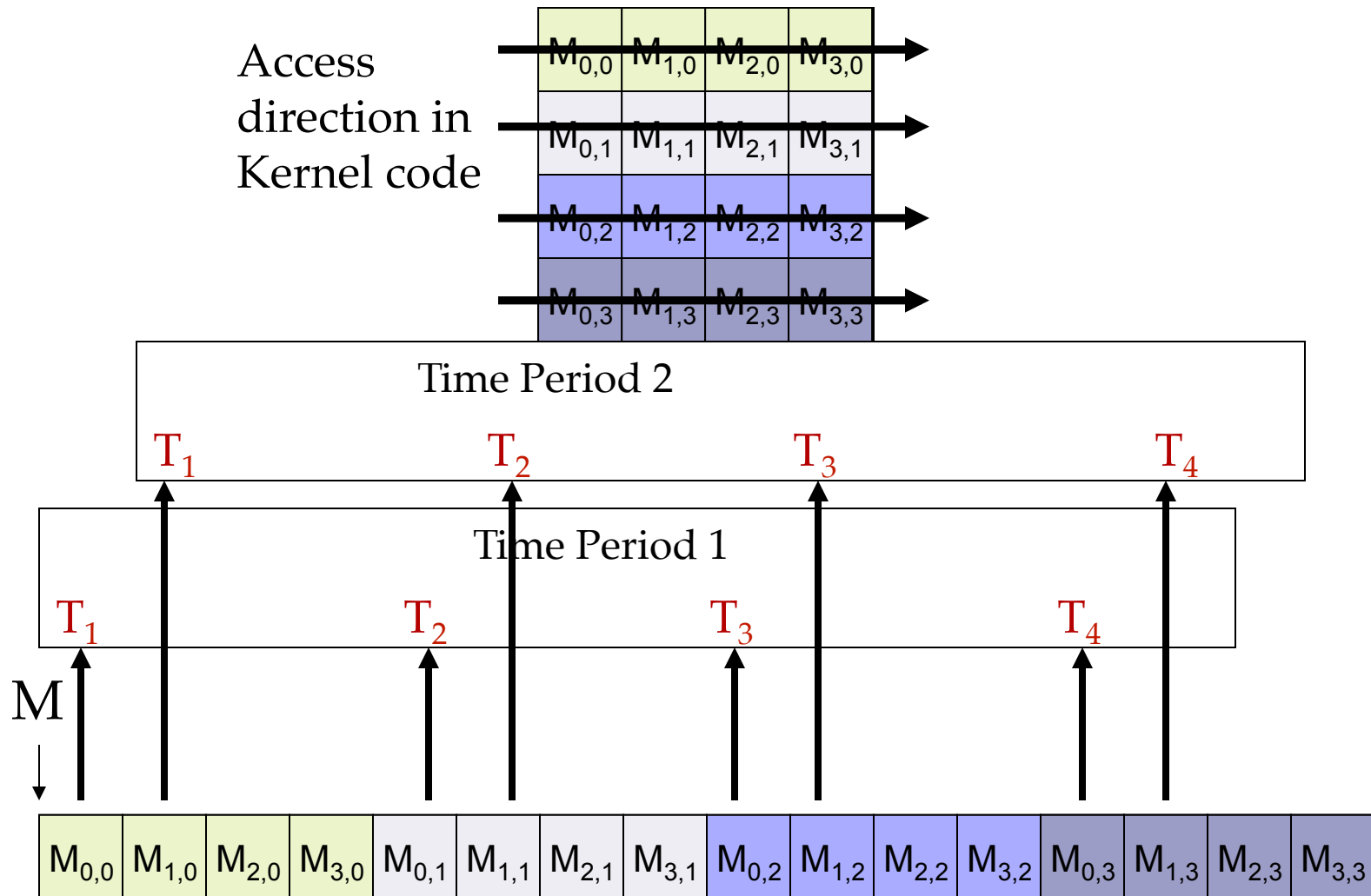
M



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

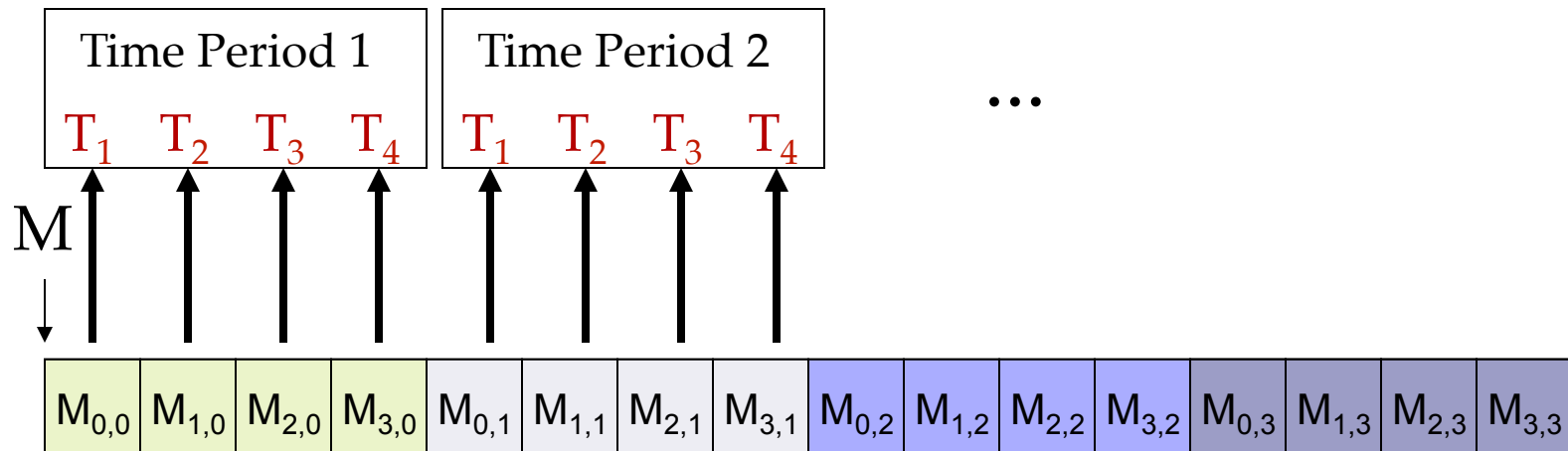
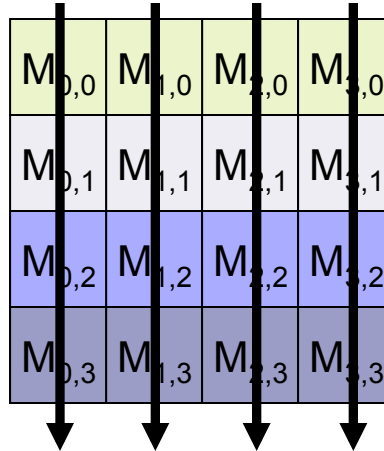
From David Kirk/NVIDIA and Wen-mei W. Hwu

Example: Matrix in C



Example: Matrix in C

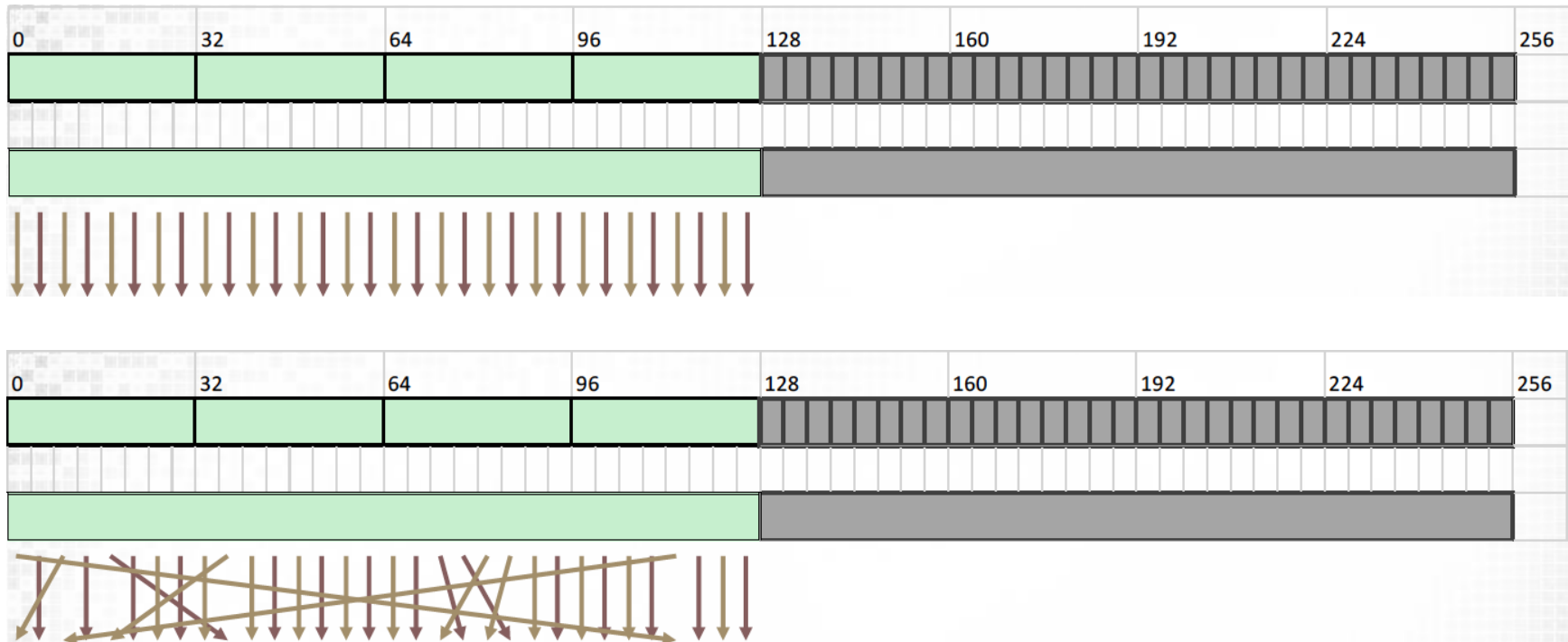
Access
direction in
Kernel code



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

Coalesced memory access (L1)

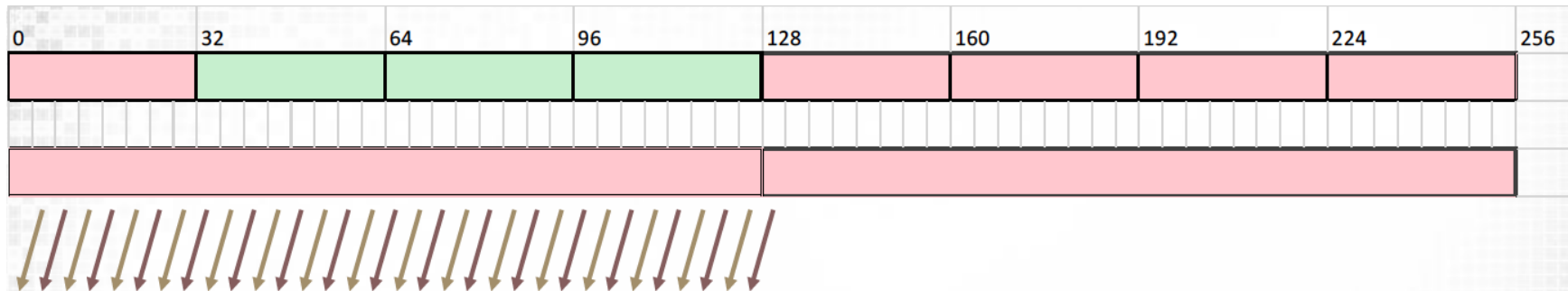
- All accesses fall inside 128 byte cache line



- 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 serial	v2 per row	v3 per elm
Time [ms]	2522	4.38	1.53

Transpose example

- How well are we doing?
 - ❑ Theoretical peak bandwidth = 288 GB/s
 - ❑ Achieved bandwidth = $2 * N^2 * 8 / 10^9 / \text{runtime}$
 - ❑ Memory utilization: $100 \% * (\text{Achieved} / \text{Peak})$

Version	v1 serial	v2 per row	v3 per elm
Time [ms]	2522	4.38	1.53
Memory utilization	< 0.1 %	10.5 %	30.1 %

Transpose example (v3 per elm)

■ Why are we not doing better?

```
// Kernel to be launched with one thread per element of A
__global__
void transpose_per_elm(double *A, double *At)
{
    // 2D thread indices defining row and col of element
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;

    At[j + i*N] = A[i + j*N]; // At(j,i)=A(i,j)
}
```

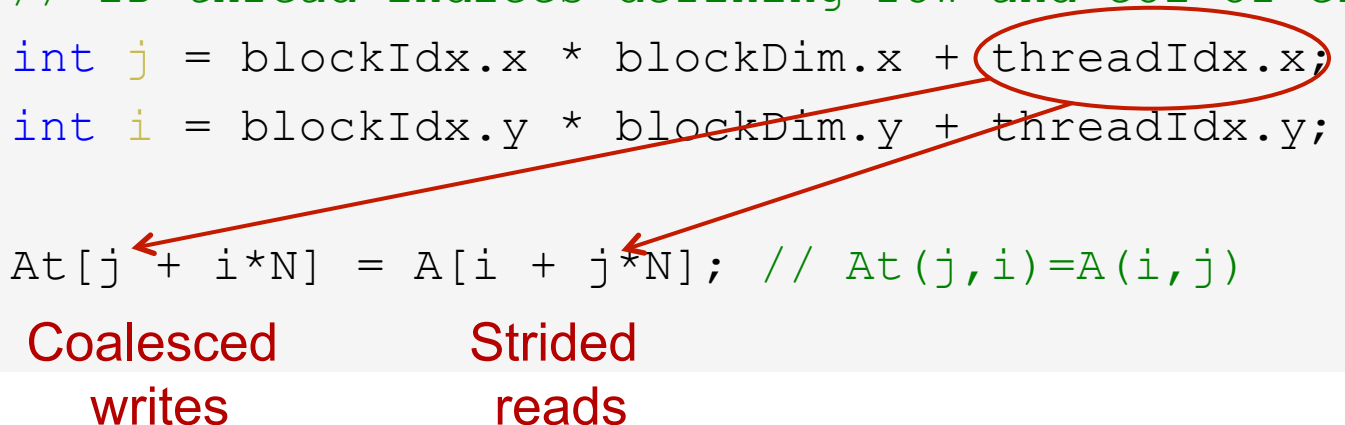
Transpose example (v3 per elm)

■ Why are we not doing better?

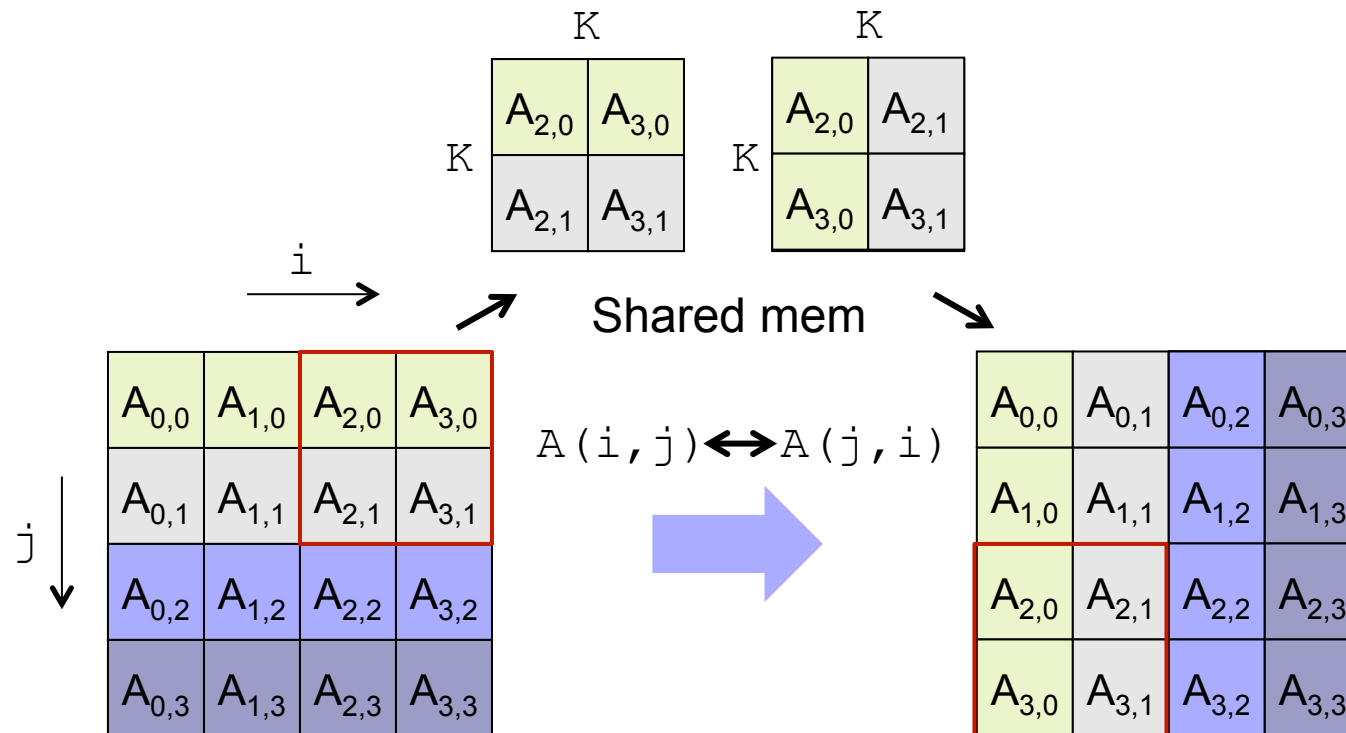
```
// Kernel to be launched with one thread per row of A
__global__
void transpose_per_elm(double *A, double *At)
{
    // 2D thread indices defining row and col of element
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;

    At[j + i*N] = A[i + j*N]; // At(j,i)=A(i,j)
}
```

Coalesced writes **Strided reads**



Transpose example (v4 smem)



- We read a block from A into shared mem (coalesced) and write from shared mem to A^t (coalesced)

Transpose example (v4 smem)

```
// Kernel to be launched with one thread per element of A
__global__
void transpose_smem(double *A, double *At)
{
    // 2D thread indices defining row and col of elements
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int jt = blockIdx.x * blockDim.x + threadIdx.y;
    int it = blockIdx.y * blockDim.y + threadIdx.x;

    __shared__ double smem[K][K];
    smem[threadIdx.y][threadIdx.x] = A[it + jt*N];
    __syncthreads();

    At[j + i*N] = smem[threadIdx.x][threadIdx.y];
}
```

Transpose example (v4 smem)

```
$ nvprof --print-gpu-summary ./transpose_gpu
==23164== Profiling application: ./transpose_gpu
==23164== Profiling result:
Time(%)      Time      Calls      Avg      Min      Max      Name
67.06%    98.992ms      100    989.92us  986.83us  992.46us  transpose_smem(double*, double*)
18.81%    27.771ms        1    27.771ms  27.771ms  27.771ms  [CUDA memcpy DtoH]
13.90%    20.523ms        1    20.523ms  20.523ms  20.523ms  [CUDA memcpy HtoD]
 0.22%     331.49us        1    331.49us  331.49us  331.49us  [CUDA memset]
$
```

Version	v1 serial	v2 per row	v3 per elm	v4 smem
Time [ms]	2522	4.38	1.53	0.98
Memory utilization	< 0.1 %	10.5 %	30.1 %	47.0 %

Synchronization

Synchronization

- How do CUDA threads communicate?

Synchronization

- How do CUDA threads communicate?
- Like OpenMP, you have to share data “by hand”
 - ❑ A. Inter-block communication through shared memory
 - ❑ B. Inter-grid communication through global memory

Synchronization

- How do CUDA threads communicate?
- Like OpenMP, you have to share data “by hand”
 - ❑ A. Inter-block communication through shared memory
 - ❑ B. Inter-grid communication through global memory
- Race conditions are handled by synchronization
 - ❑ A. call `__syncthreads()` to create a barrier per block
 - See also next slide...
 - ❑ B. `cudaDeviceSynchronize()` + two kernel launches

Synchronization

- How do CUDA threads communicate?
- Like OpenMP, you have to share data “by hand”
 - ❑ A. Inter-block communication through shared memory
 - ❑ B. Inter-grid communication through global memory
- Race conditions are handled by synchronization
 - ❑ A. call `__syncthreads()` to create a barrier per block
 - See also next slide...
 - ❑ B. `cudaDeviceSynchronize()` + two kernel launches
- What about “inter-warp” communication?
 - ❑ Barrier synchronization is not needed, why?

Be careful

- Can `__syncthreads()` cause a thread to hang?
 - E.g., usage inside conditional code

```
if (someFunc())  
{  
    __syncthreads();  
}  
// ...
```

Be careful

- Can `__syncthreads()` cause a thread to hang?
 - E.g., usage inside conditional code

```
if (someFunc())  
{  
    __syncthreads();  
}  
// ...
```

- Yes!
 - ...but not if the conditional **evaluates identically** across the entire thread block, otherwise the code execution is likely to hang or produce unintended side effects

Atomic operations

Thread synchronization (cont'd)

■ Atomic functions

- ❑ An atomic function performs a read-modify-write atomic operation on one 32-bit or 64-bit word residing in shared or **global memory**
- ❑ Can be used to avoid race conditions **over blocks**

// Arithmetic

atomicAdd()

atomicSub()

atomicExch()

atomicMin()

atomicMax()

atomicAdd()

atomicDec()

atomicCAS()

// Bitwise

atomicAnd()

atomicOr()

atomicXor()

```
atomicAdd(&A[i], sum);
```

Selected memory topics

Global/constant memory allocation

■ Static allocation of global/constant memory

- ❑ Must be declared outside of a function body

- `__device__` / `__constant__`

- ❑ Host can access with CUDA runtime functions

- `cudaGetSymbolAddress()`

- `cudaGetSymbolSize()`

- `cudaMemcpyToSymbol()`

- `cudaMemcpyFromSymbol()`

This is the only way to allocate constant memory!

```
__constant__ double constData[256];  
double data[256];  
cudaMemcpyToSymbol(constData, data, sizeof(data));  
cudaMemcpyFromSymbol(data, constData, sizeof(data));
```

Asynchronous data transfer

■ Overlapping of data transfer and computation

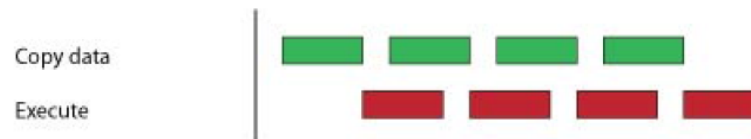
❑ `cudaMemcpyAsync()` and **streams** (advanced!)

❑ `cudaMallocHost()`

Transfer followed by computation

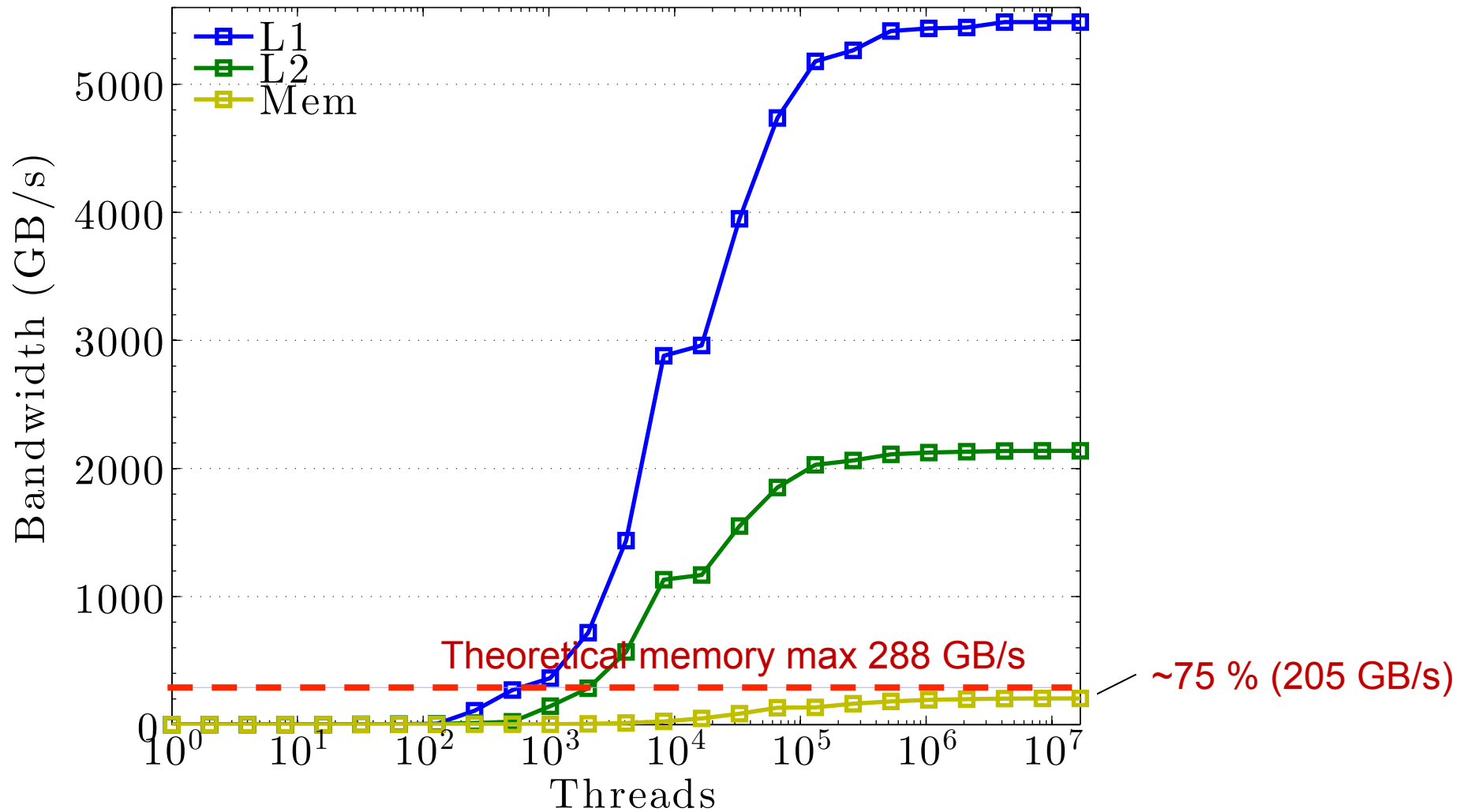


Overlapping transfer and computation



Blocking for caches

Nvidia Tesla K40c @ 0.745GHz



- Wrap up exercises 1-3
- Do the exercise 4 (matrix-vector multiplication)
- Next presentation “CUDA Performance Tuning Control Flow” at 13.00 (Tuesday)!

End of lecture