KOÇ UNIVERSITY

Performance Considerations on GPU

Didem Unat

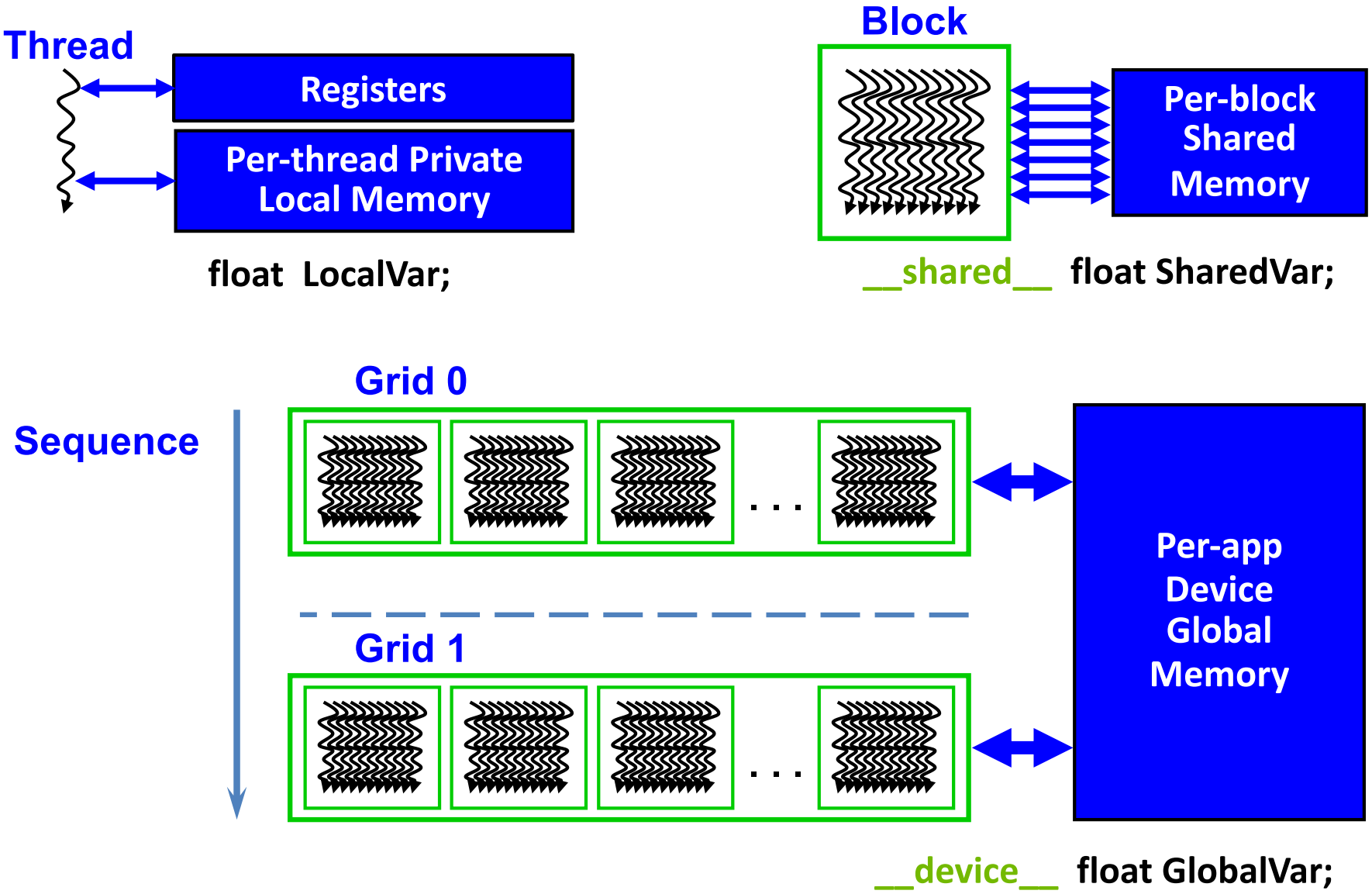
dunat@ku.edu.tr

http://parcorelab.ku.edu.tr

Performance Considerations

- Memory Types
- Enough Parallelism
- Overlap Transfers
- Memory Coalescing
- Shared Memory Bank Conflicts
- Control-Flow Divergence
- Occupancy
- Kernel Launch Overheads

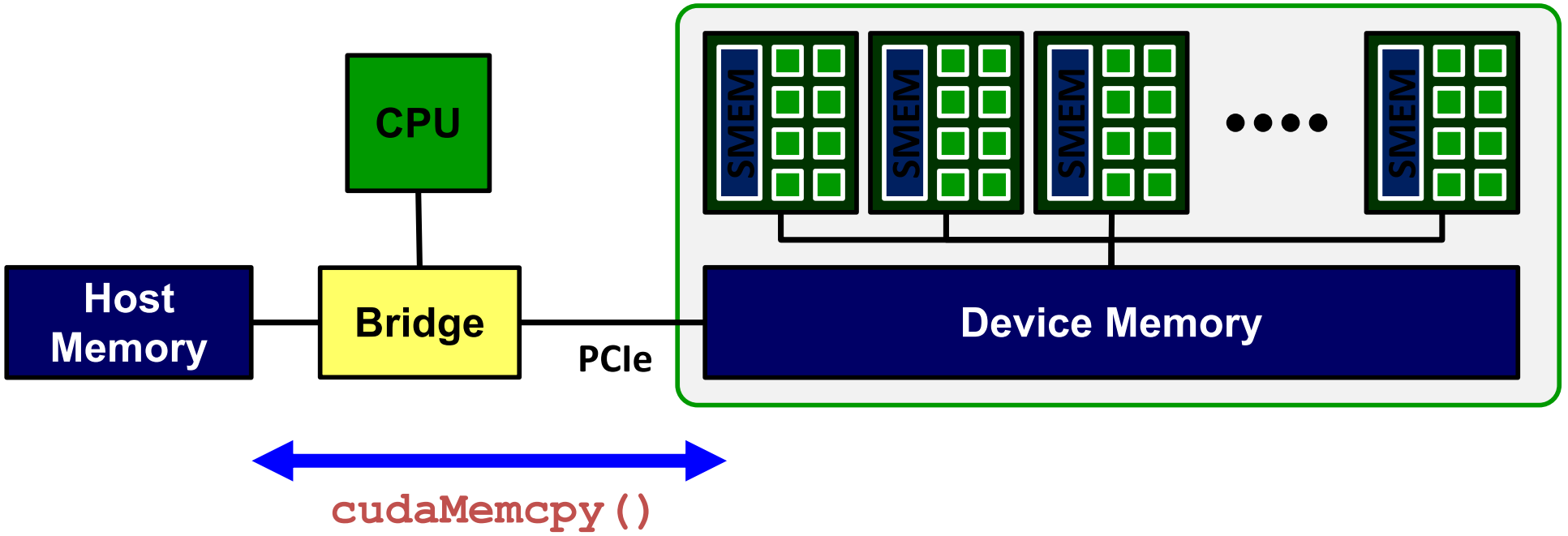
CUDA Parallel Threads and Memory



Expose Massive Parallelism

- Thread block count
  - A thread block executes on one SM
  - Need many blocks to use 10s of SMs
  - SM executes 2 to 8 concurrent blocks efficiently
  - Need many blocks to scale to different GPUs
- Use hundreds of threads per thread block
  - A thread instruction executes on one core
  - Need 128 – 512 threads/SM to use all the cores all the time
  - Use multiple of 32 threads (warp) per thread block
  - Fine-grained data parallelism, vector parallelism, thread parallelism, instruction-level parallelism

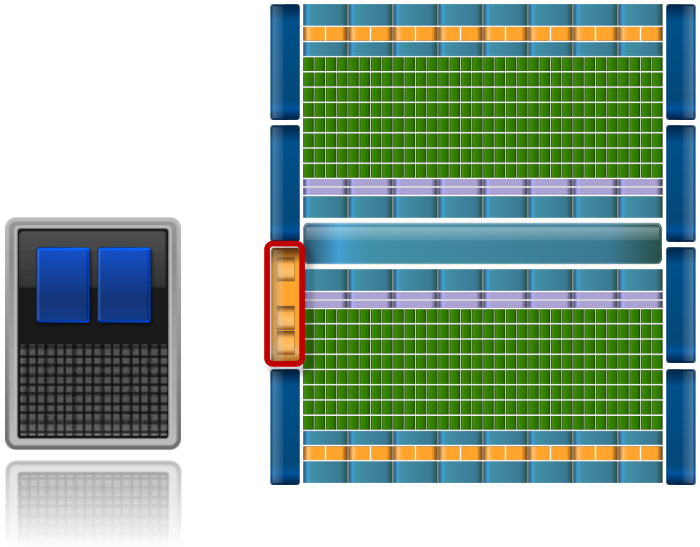
Using cudaMemCpy()



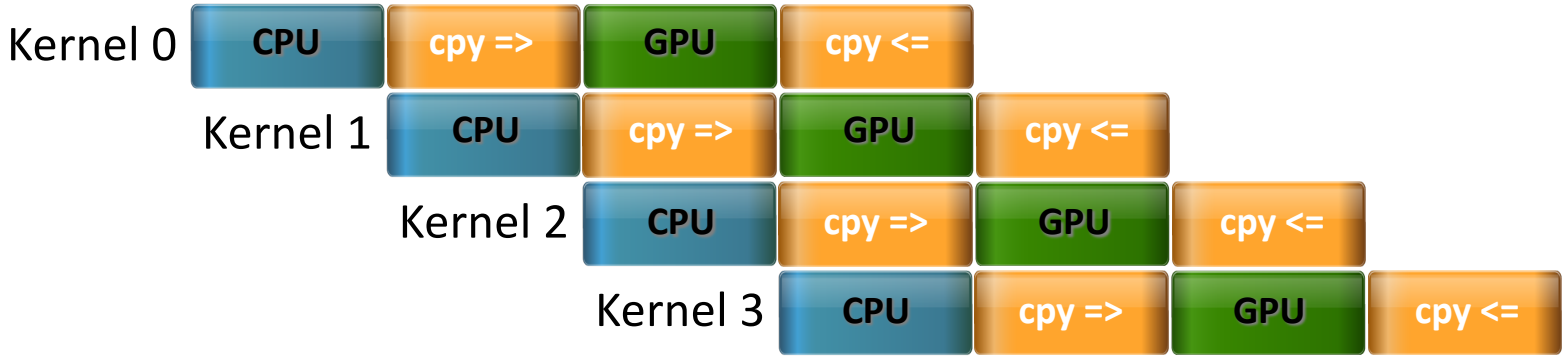
- cudaMemcpy() invokes a DMA copy engine
- Minimize the number of copies
- Use data as long as possible in a given place

Overlap computing & CPU⇔GPU transfers

- cudaMemcpy() invokes data transfer engines
  - CPU→GPU and GPU→CPU data transfers
  - Overlap with CPU and GPU processing



- Pipeline Snapshot:



MEMORY COALESCING

Enable Global Memory Coalescing

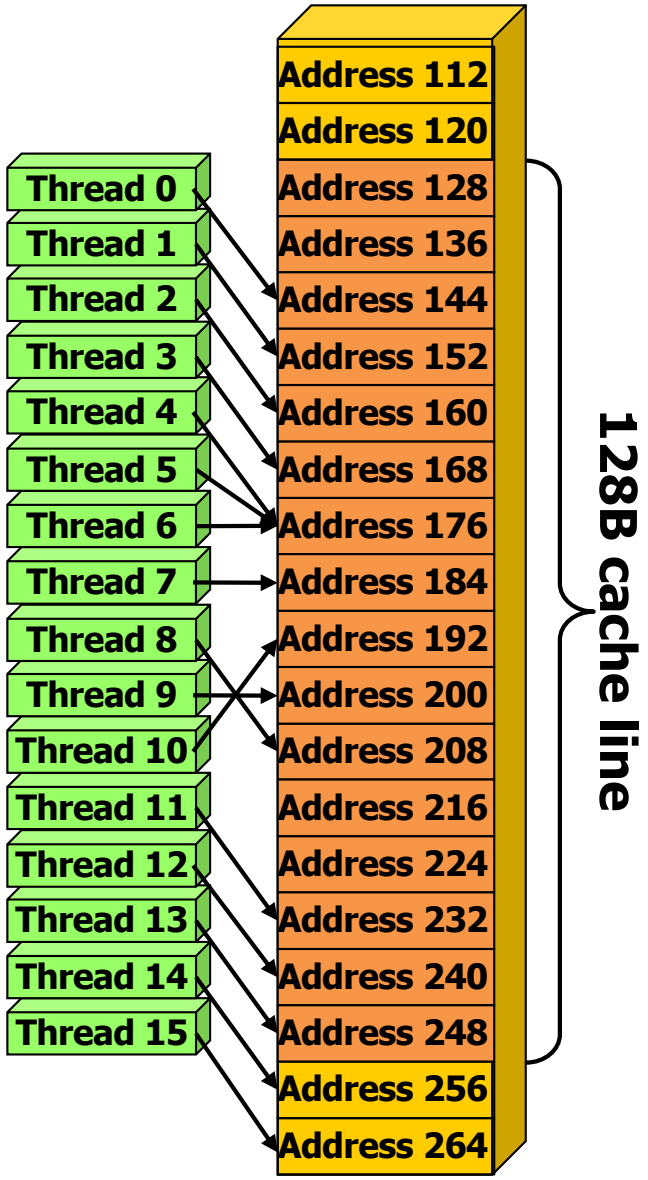
- Individual threads access independent addresses
- A thread loads/stores 1, 2, 4, 8, 16 Bytes per access
- LD.sz / ST.sz; sz = {8, 16, 32, 64, 128} bits per thread

- For 32 parallel threads in a warp, SM load/store units coalesce individual thread accesses into minimum number of 128 Bytes cache line accesses or 32B memory block accesses
- Access serializes to distinct cache lines or memory blocks

- Use nearby addresses for threads in a warp
- Use unit stride accesses when possible
- Use Structure of Arrays (SoA) to get unit stride

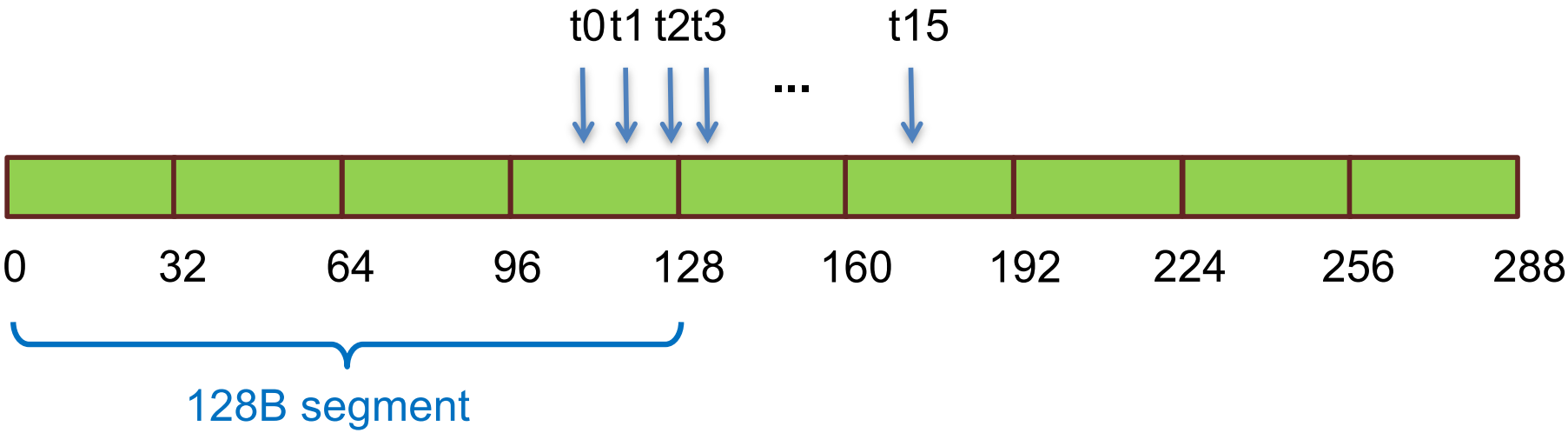
Memory Coalescing

- Off-chip memory is accessed in chunks
  - Even if you read only a single word
  - If you don't use whole chunk, bandwidth is wasted
- Chunks are aligned to multiples of 32/64/128 bytes
  - Unaligned accesses will cost more



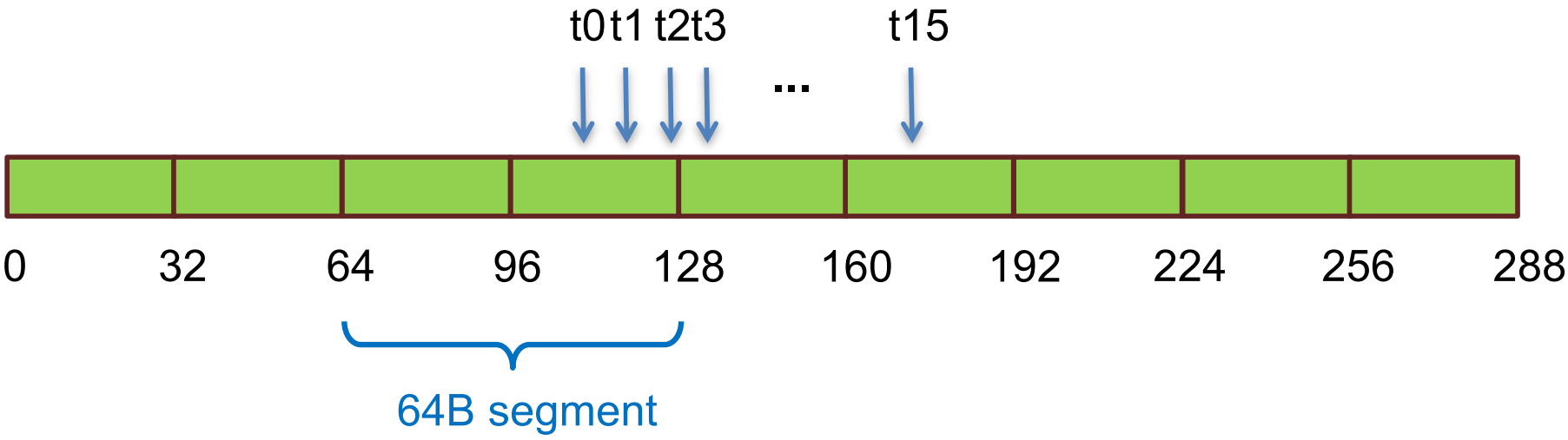
Threads 0-15 access 4-byte words at addresses 116-176

- Thread 0 is lowest active, accesses address 116
- 128-byte segment: 0-127



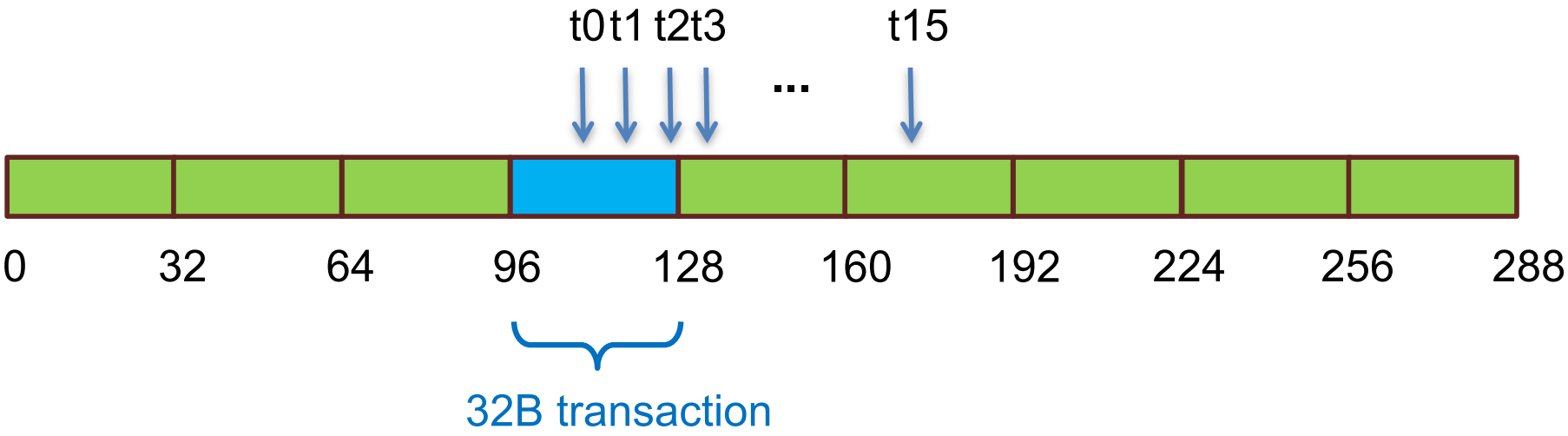
Threads 0-15 access 4-byte words at addresses 116-176

- Thread 0 is lowest active, accesses address 116
- 128-byte segment: 0-127 (reduce to 64B)



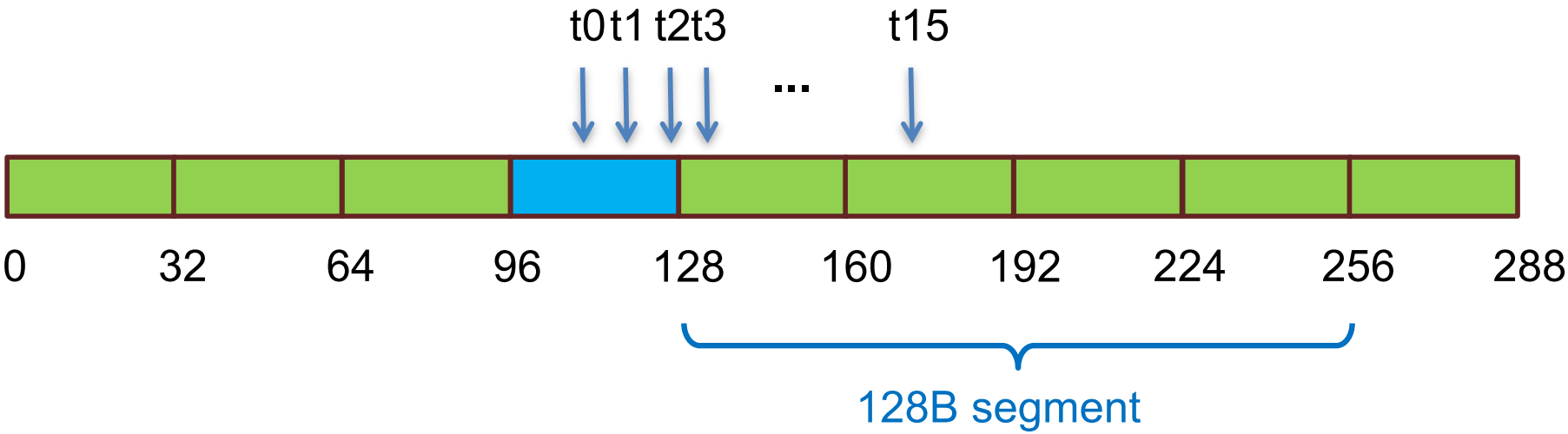
Threads 0-15 access 4-byte words at addresses 116-176

- Thread 0 is lowest active, accesses address 116
- 128-byte segment: 0-127 (reduce to 32B)



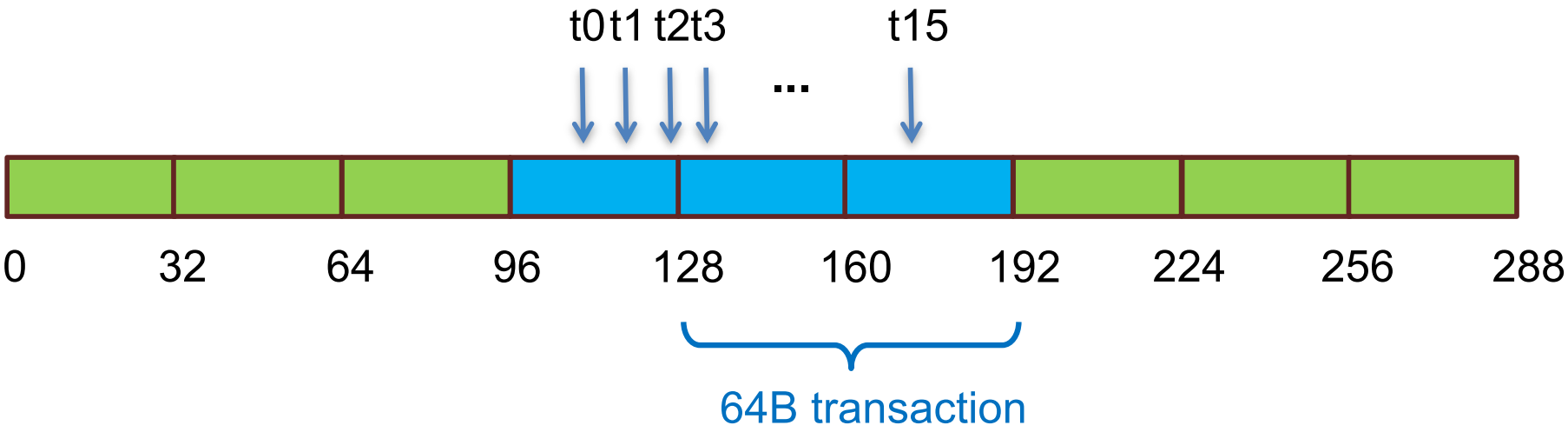
Threads 0-15 access 4-byte words at addresses 116-176

- Thread 3 is lowest active, accesses address 128
- 128-byte segment: 128-255



Threads 0-15 access 4-byte words at addresses 116-176

- Thread 3 is lowest active, accesses address 128
- 128-byte segment: 128-255 (reduce to 64B)



Unit stride accesses coalesce well

```
__global__ void kernel(float* arrayIn,
                       float* arrayOut)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;

  // Stride 1 coalesced load access
  float val = arrayIn[i];

  // Stride 1 coalesced store access
  arrayOut[i] = val + 1;
}
```

Consider the stride of your accesses

```
__global__ void foo(int* input, float3* input2)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  // Stride 1
  int a = input[i];

  // Stride 2, half the bandwidth is wasted
  int b = input[2*i];

  // Stride 3, 2/3 of the bandwidth wasted
  float c = input2[i].x; //assume there is .y and .z
}
```

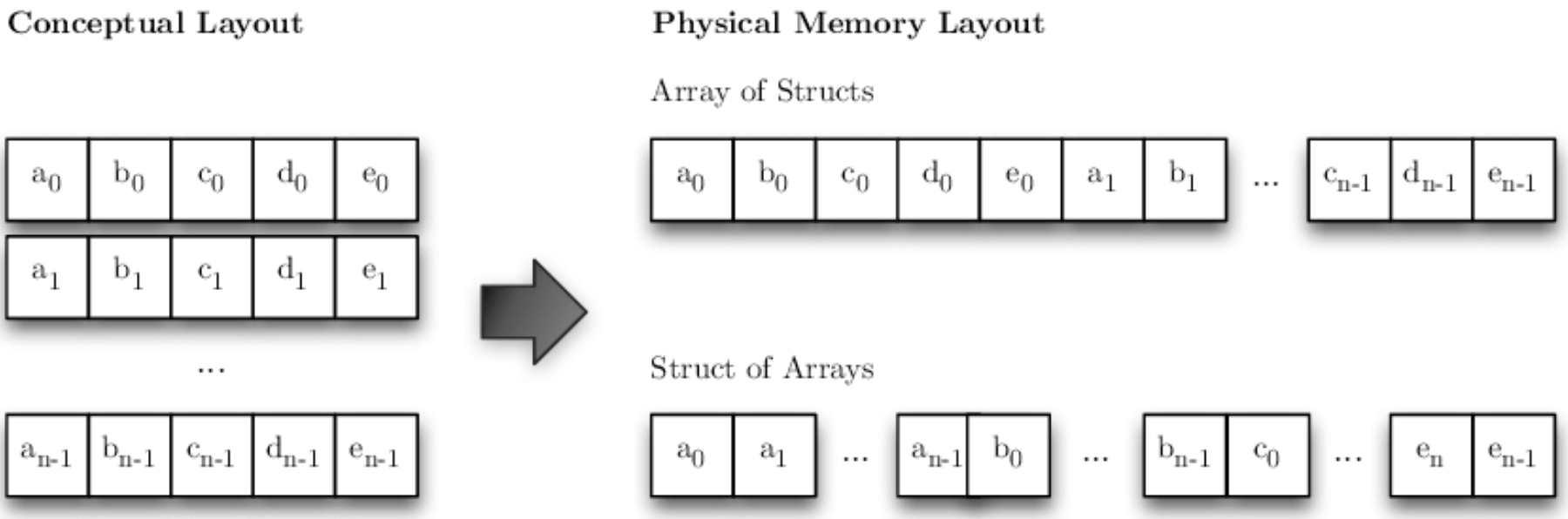
Example: Array of Structures (AoS)

```
struct record
{
  int key;
  int value;
  int flag;
};

record  *d_records;
cudaMalloc((void**)&d_records, ...);
```

Example: Structure of Arrays (SoA)

```
struct SoA
{
  int  * keys;
  int  * values;
  int  * flags;
};

SoA d_SoA_data;
cudaMalloc((void**)&d_SoA_data.keys, ...);
cudaMalloc((void**)&d_SoA_data.values, ...);
cudaMalloc((void**)&d_SoA_data.flags, ...);
```

SoA vs AoS



- Usually struct of arrays are better for GPUs
  - Because a thread is assigned one element
- On CPU, depending on the computation requirements, one can use array of structs
  - Because a thread is assigned a larger chunk size

Example: SoA vs. AoS

```
__global__ void bar(record *AoS_data, SoA SoA_data)
{

  int i = blockDim.x * blockIdx.x+ threadIdx.x;

  // AoS wastes bandwidth
  int key = AoS_data[i].key;

  // SoA efficient use of bandwidth
  int key_better = SoA_data.keys[i];
}
```

Memory Coalescing

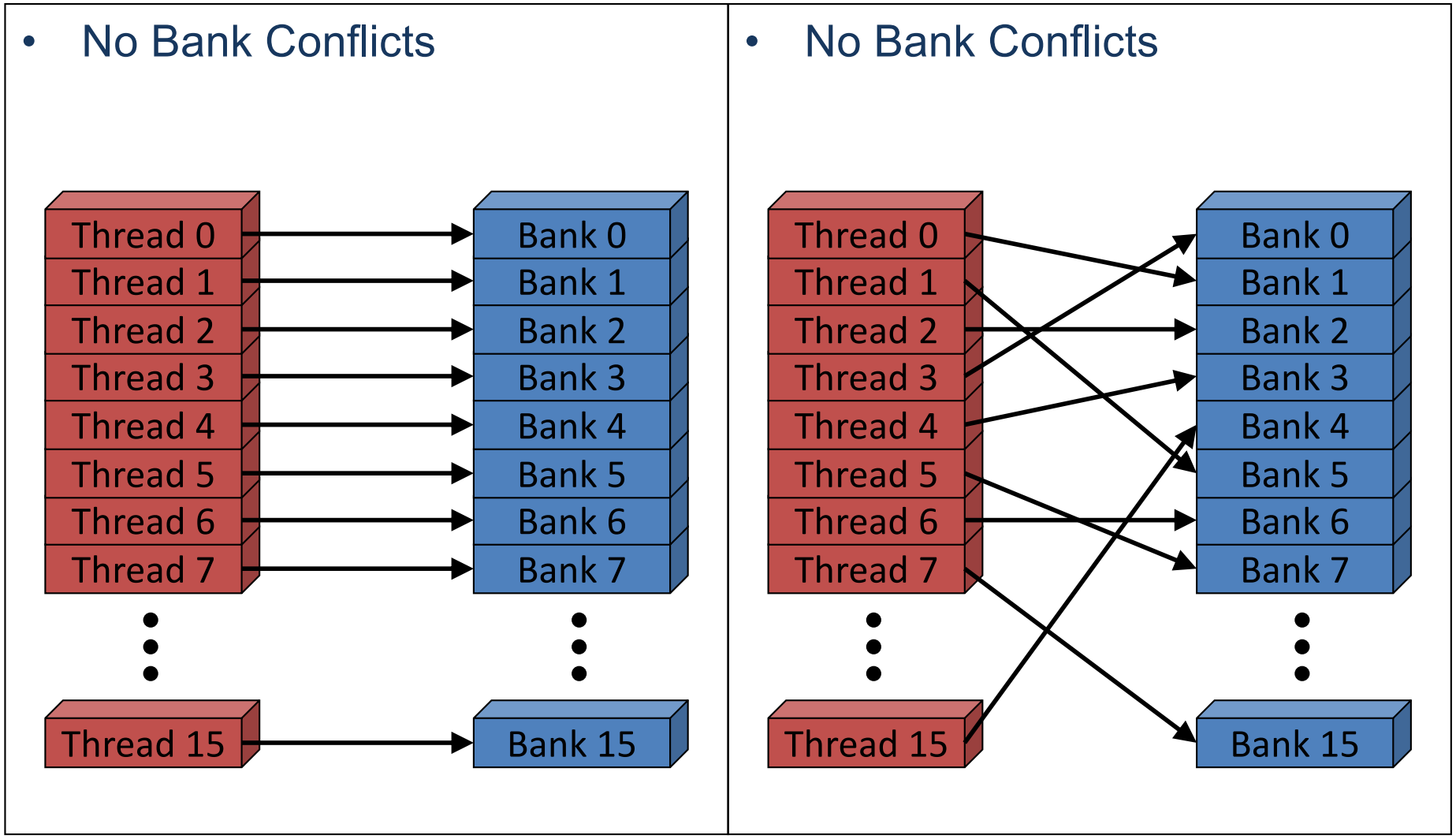
- Structure of array is often better than array of structures on the GPU
  - Very clear win on regular, stride 1 access patterns
  - Unpredictable or irregular access patterns are case-by-case

SHARED MEMORY BANK CONFLICTS

Shared Memory

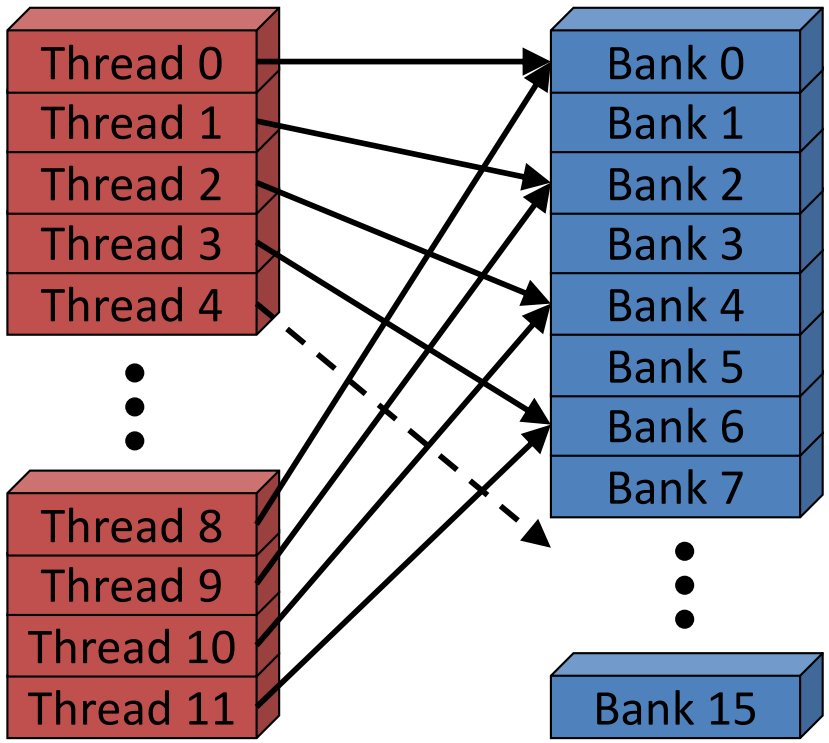
- Shared memory is banked
  - Only matters for threads within a warp
  - Full performance with some restrictions
  - Threads can each access different banks
  - Or can all access the same value
- Consecutive words are in different banks
- If two or more threads access the same bank but different value, get bank conflicts

Bank Addressing Examples

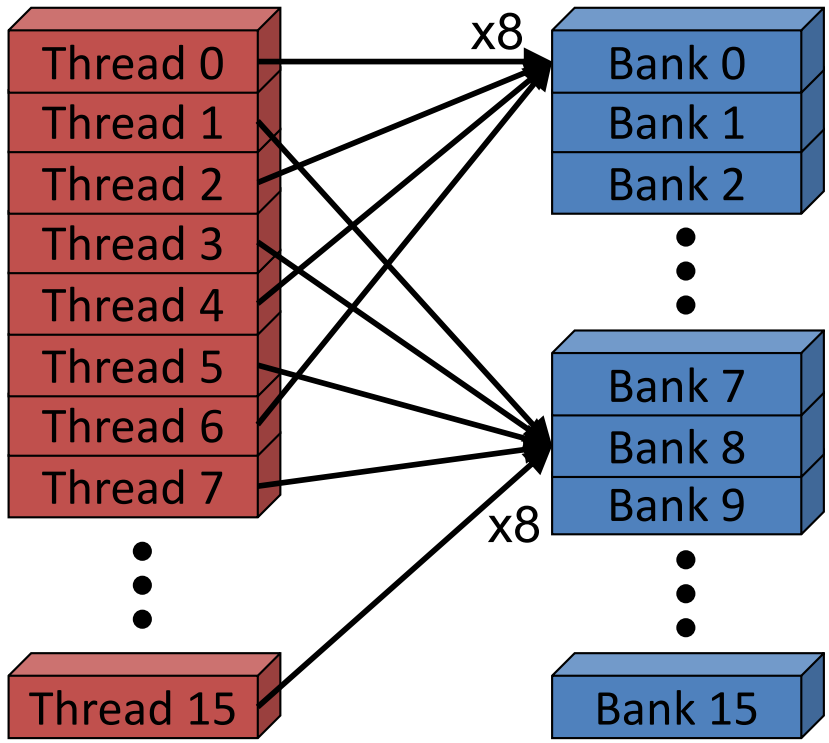


Bank Addressing Examples

- 2-way Bank Conflicts



- 8-way Bank Conflicts



Trick to Assess Impact On Performance

- Change all Shared memory (SMEM) reads to the same value
  - All broadcasts = no conflicts
  - Will show how much performance could be improved by eliminating bank conflicts
- The same doesn't work for SMEM writes
  - So, replace SMEM array indices with threadIdx.x
  - Can also be done to the reads

CONTROL FLOW DIVERGENCE

Control Flow

- Instructions are issued per 32 threads (warp)
- Divergent branches:
  - Threads within a single warp take different paths
    - if-else, …
  - Different execution paths within a warp are serialized
- Different warps can execute different code with no impact on performance

Control Flow

- Avoid diverging within a warp
  - Example with divergence:

```
if (threadIdx.x > 2) {...}
else {...}
```

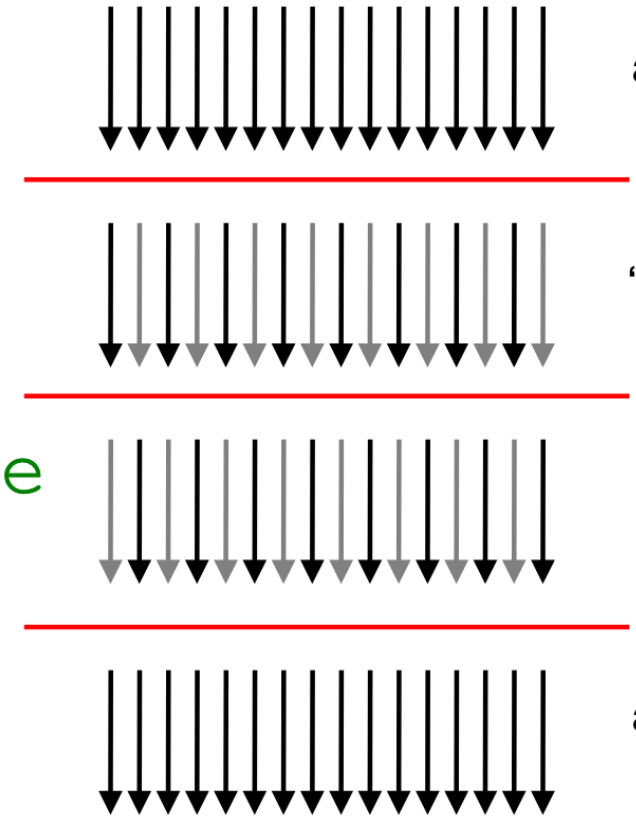
    Branch granularity < warp size
  - Example without divergence:

```
if (threadIdx.x / WARP_SIZE > 2)
{...}
else {...}
```

    Branch granularity is a whole multiple of warp size

Divergence within warp

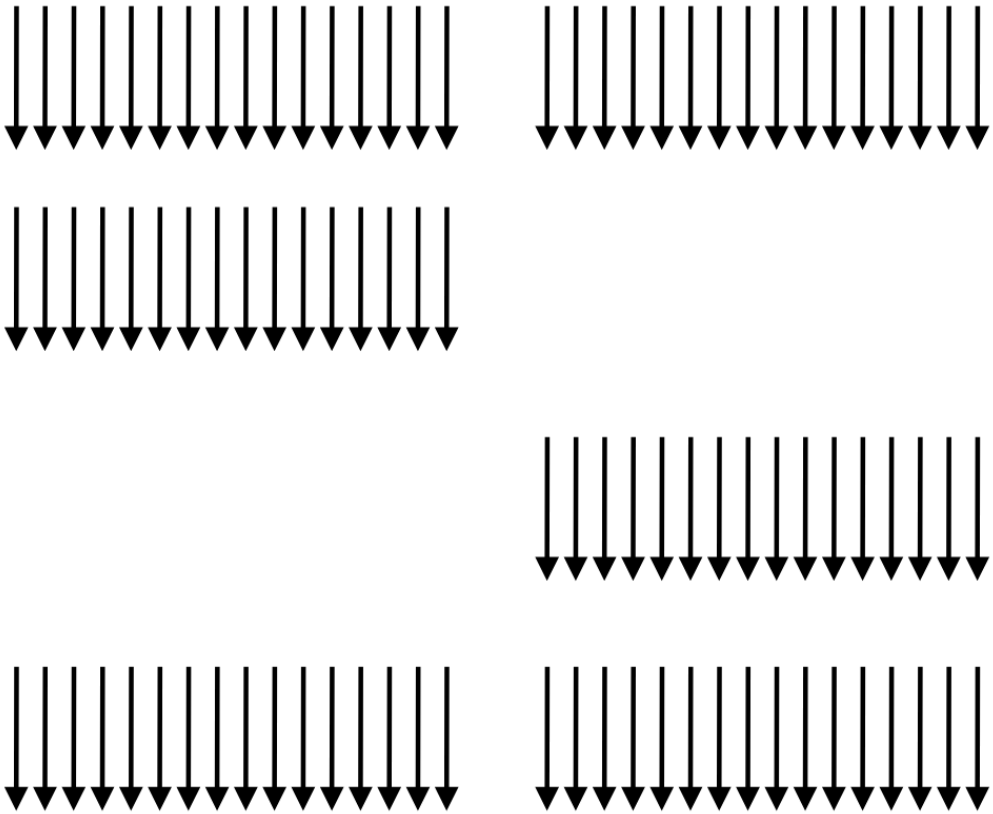
```
// ...
if(condition) {
   // do smth
   // ...
} else {
   // do smth else
   // ...
}
// ...
```



divergence *within* warp => performance penalty

Divergence between warps

```
// ...
if(condition) {
   // do smth
   // ...
} else {
   // do smth else
   // ...
}
// ...
```



divergence *between* warps => no penalty

Example: Divergent Iteration

```
__global__ void per_thread_sum(int *indices,
                               float *data,
                               float *sums)
{
  ...
  // number of loop iterations is data
  // dependent
  for(int j=indices[i];j<indices[i+1]; j++)
  {
    sum += data[j];
  }
  sums[i] = sum;
}
```

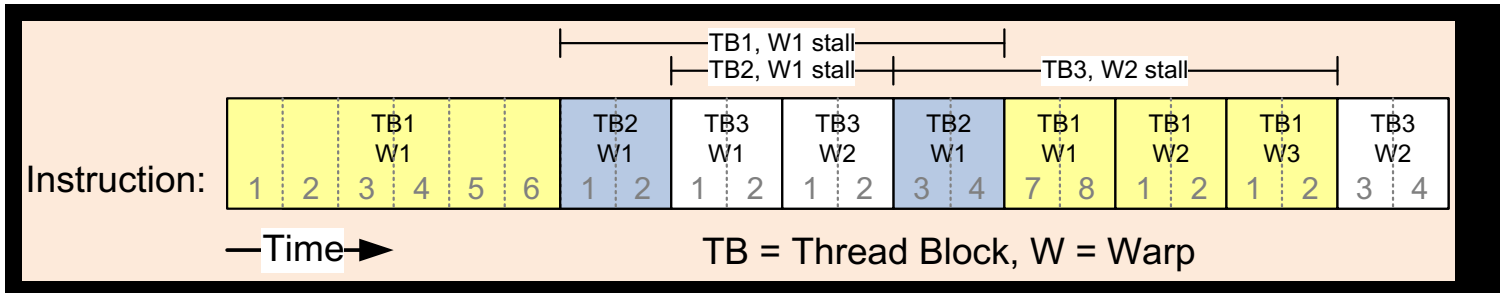
Iteration Divergence

- A single thread can drag a whole warp with it for a long time
- Know your data patterns
- If data is unpredictable, try to flatten peaks by letting threads work on multiple data items

OCCUPANCY

Reminder: Thread Scheduling

- SM implements zero-overhead warp scheduling
  - At any time, only one of the warps is executed by SM *
  - Warps whose next instruction has its inputs ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a warp execute the same instruction when selected



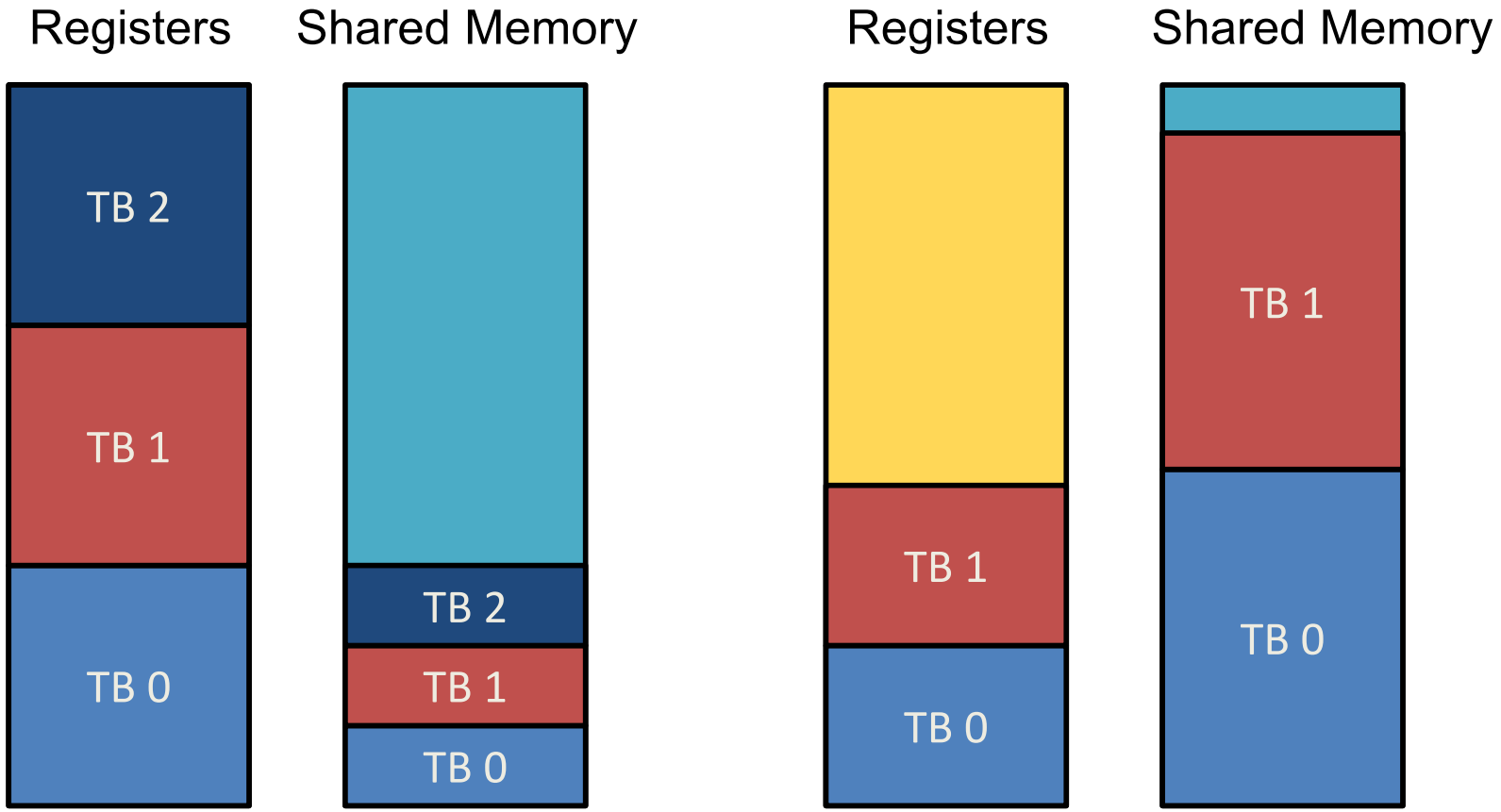
Thread Scheduling

- What happens if all warps are stalled?
  - No instruction issued → performance lost
- Most common reason for stalling?
  - Waiting on global memory
- If your code reads global memory every couple of instructions
  - You should try to maximize occupancy

What determines occupancy?

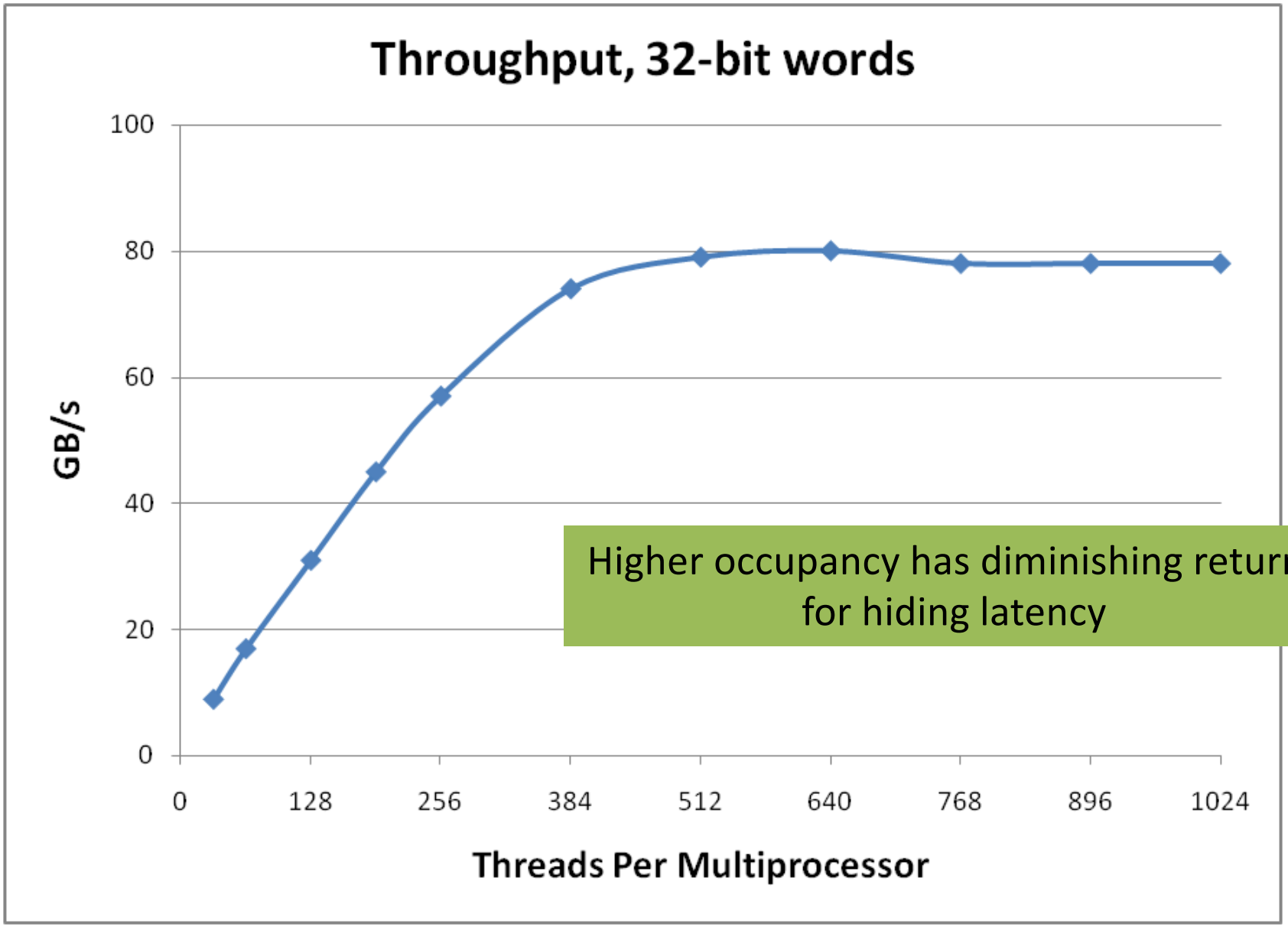
- Register usage per thread & shared memory per thread block

Resource Limits



- Pool of registers and shared memory per SM
  - Each thread block grabs registers & shared memory
  - If one or the other is fully utilized -> no more active thread blocks

Hiding Latency with more threads



How do you know what you're using?

- Use `nvcc -Xptxas -v` to get register and shared memory usage
- Or `--ptxas-options=-v`

Example:

```
ptxas info    : Used 14 registers, 2088 bytes smem, 48 bytes cmem[1]
```

KERNEL LAUNCH OVERHEAD

Kernel Launch Overhead

- Kernel launches aren't free
  - A null kernel launch will take non-trivial time
  - Actual number changes with HW generations and driver software, so we can't give you one number
- Independent kernel launches are cheaper than dependent kernel launches
  - Dependent launch: Some read back to the cpu
- If you are launching lots of small kernels you will lose substantial performance due to this effect

Kernel Launch Overheads

- If you are reading back data to the CPU for control decisions, consider doing it on the GPU
- Even though the GPU is slow at serial tasks, can do surprising amounts of work before you used up kernel launch overhead

- https://www.fz-juelich.de/SharedDocs/Downloads/IAS/JSC/EN/slides/cuda/06-cuda-performance-opt.pdf?__blob=publicationFiles