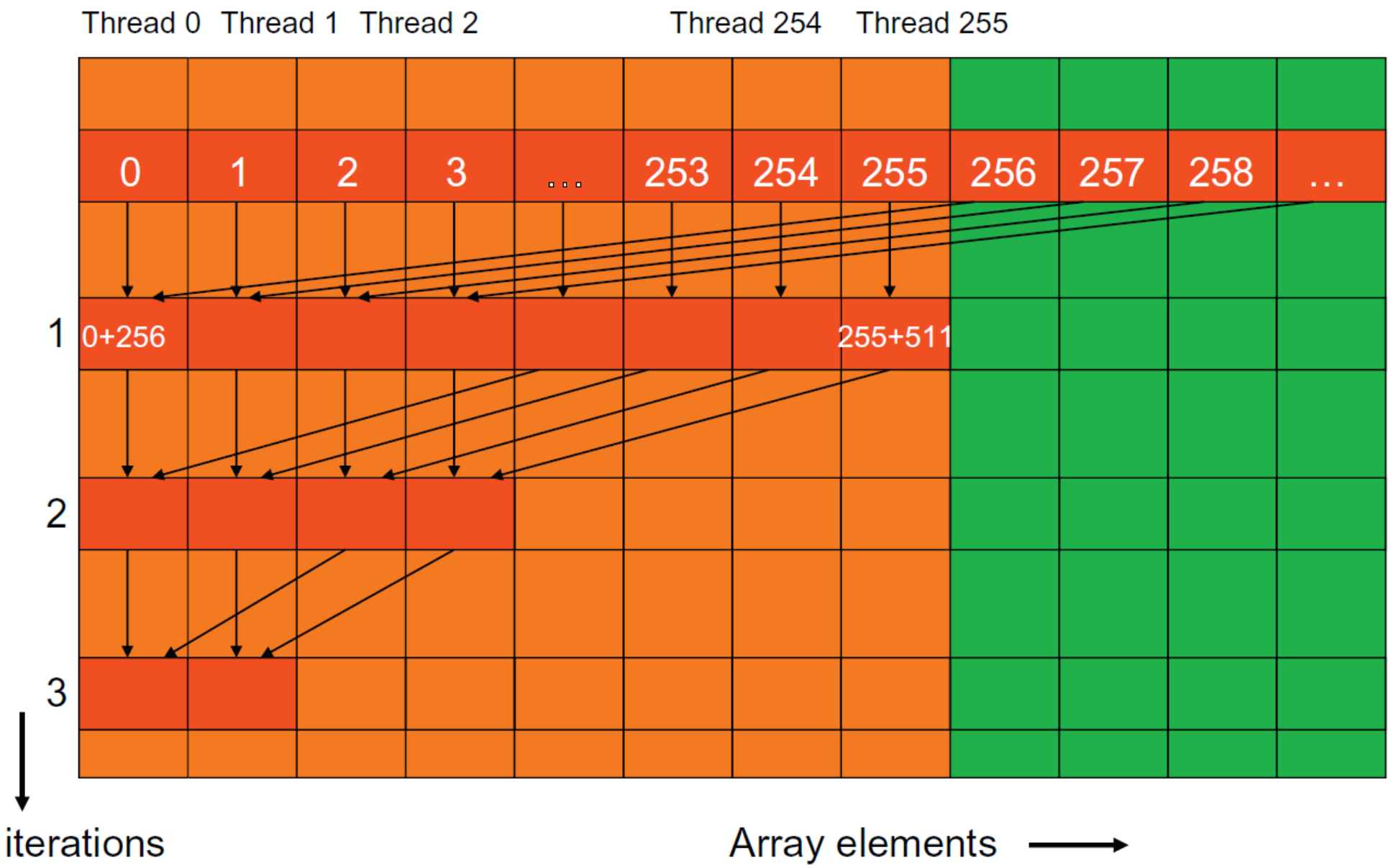
Programmazione di Sistemi ~~Embedded e~~ Multicore

Teacher: Daniele De Sensi

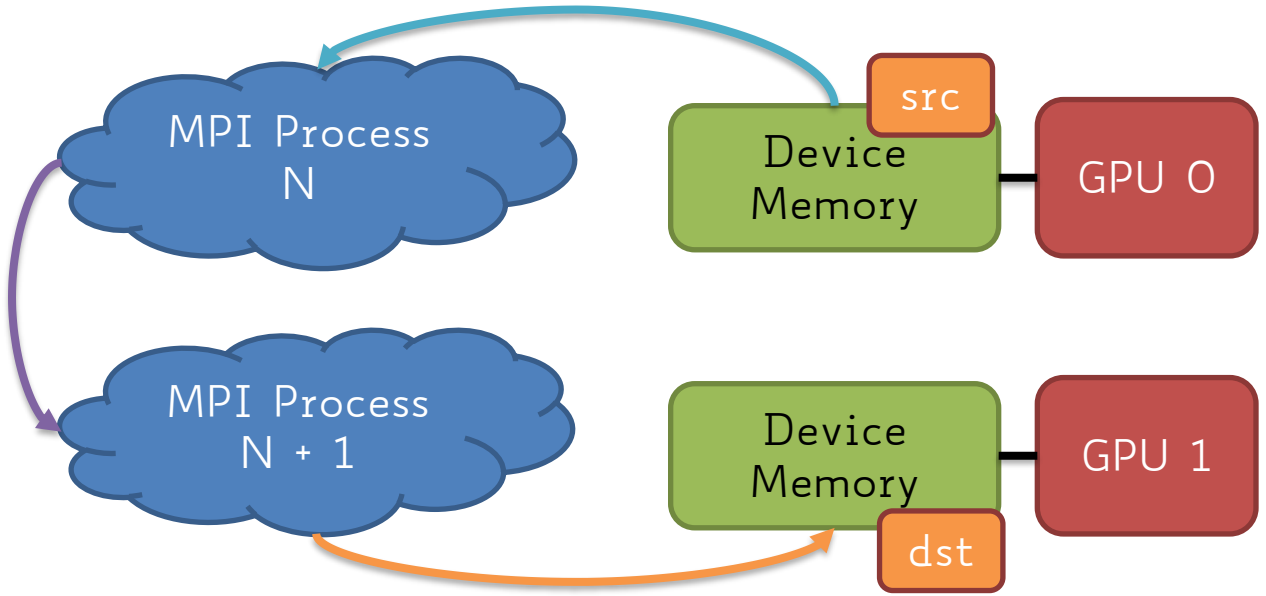
Recap

Better way of doing a reduce



MPI not GPU-Aware

- Source MPI process:
  - cudaMemcpy(tmp, src, cudaMemcpyDeviceToHost)
  - MPI_Send(tmp, …, N+1, …) // send tmp to N+1
- Destination MPI process:
  - MPI_Recv(tmp, …, N, …) // recv tmp from N+1
  - cudaMemcpy(dst, tmp, cudaMemcpyHostToDevice)



Beware!

- On the cluster, each GPU node has 2 GPUs
- Thus, you should run with 2 MPI process per node (each managing one GPU)
- You can specify the GPU you want to run on with **cudaSetDevice** e.g.:

```
if(rank % 2 == 0){
   cudaSetDevice(0);
}else{
   cudaSetDevice(1);
}
```

*CCL: Beyond MPI

- *CCL libraries becoming more and more popular and developed by big companies (NCCL, RCCL, HCCL, etc...)
- Provide a few collectives (mostly those needed for ML training – allreduce, reduce-scatter, allgather, and a few others), and point-to-point
- They do not adhere to any standard, so they are more flexible and can innovate much faster
  - MPI is a huge standard and any change must ensure backward compatibility
  - MPI design is driven by the community through open discussion. Changes take years to be accepted and adopted by the implementations
- Still, there is value in having something standard. What if I have a system deploying both AMD and NVIDIA GPUs? (NCCL and RCCL do not talk to each other)

*CCL: Beyond MPI

In practice, *CCL often provide higher performance than MPI (at least for collectives)

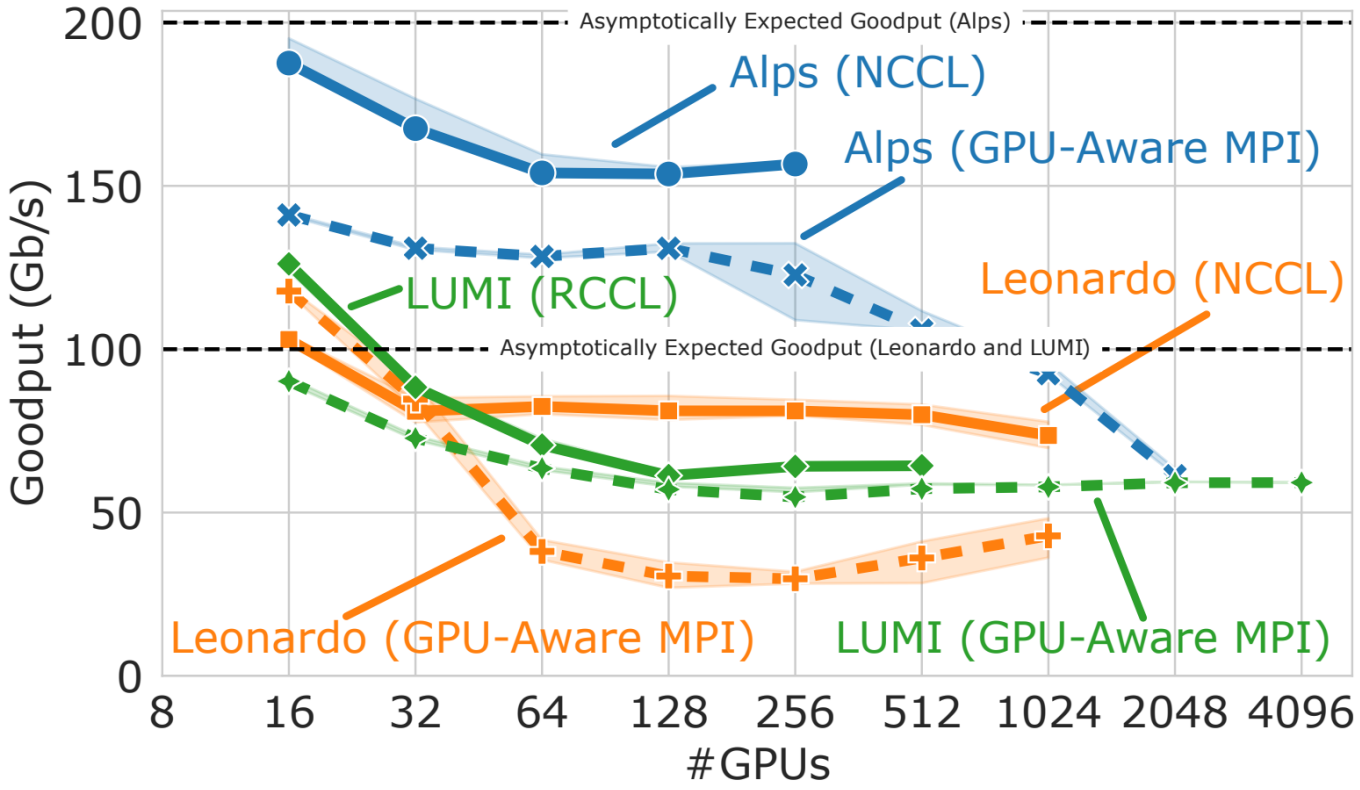
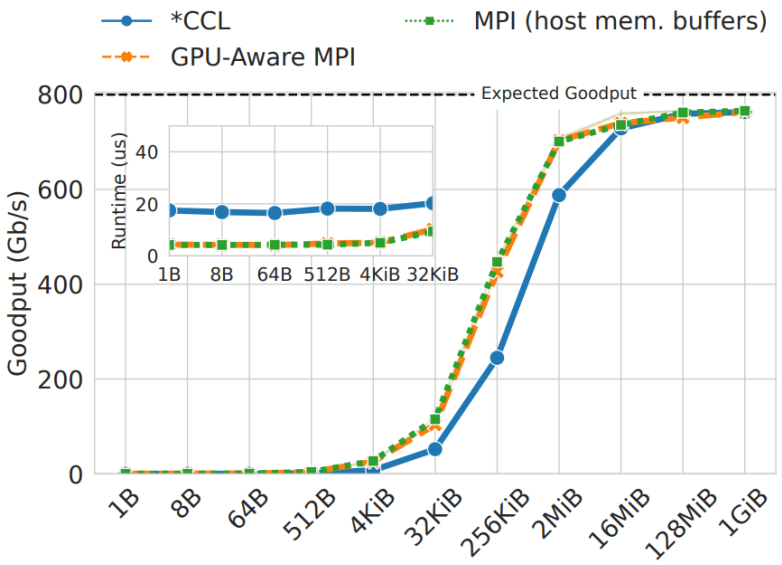


Fig. 9: 2 MiB alltoall scalability.

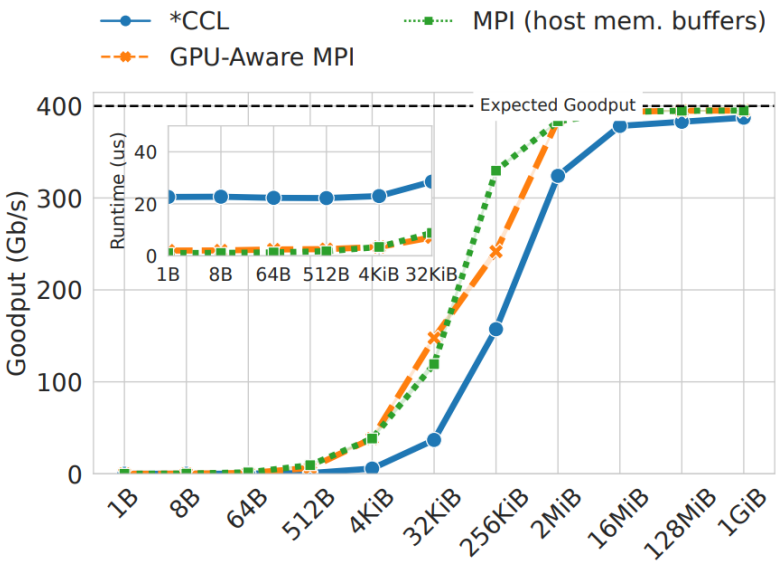
If you want to know more: https://danieledesensi.github.io/assets/pdf/2024_GPUGPU.pdf

*CCL: Beyond MPI

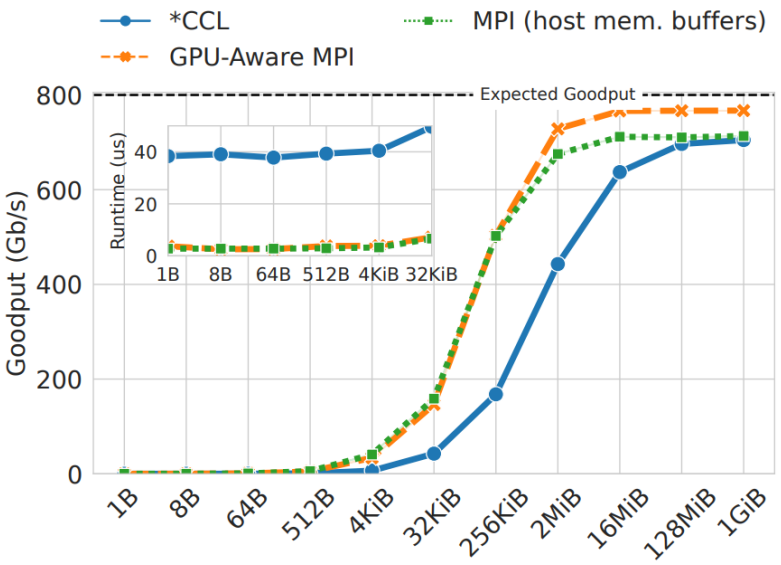
(Not so well on the point-to-point)



(a) Alps



(b) Leonardo



(c) LUMI

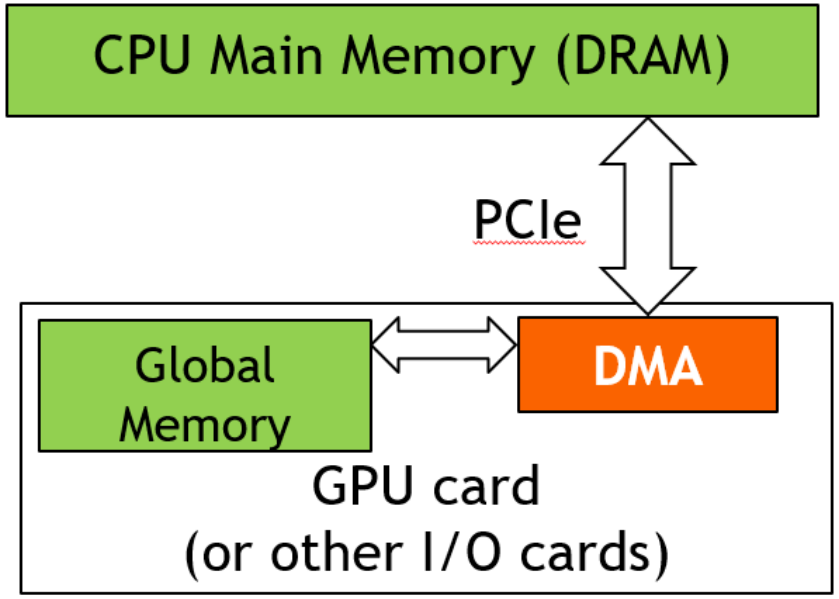
If you want to know more: https://danieledesensi.github.io/assets/pdf/2024_GPUGPU.pdf

Questions?

Pinned Memory

CPU-GPU Transfers using DMA

- DMA (Direct Memory Access) hardware is used by cudaMemcpy() for better efficiency
  - Frees CPU for other tasks
  - Hardware unit specialized to transfer a number of bytes requested by OS between physical memory address space regions (some can be mapped I/O memory locations)
  - Uses system interconnect (e.g., PCIe)



Virtual Memory Management

- Modern systems use virtual memory management
  - Many virtual memory spaces mapped into a single physical memory
  - Virtual addresses (pointer values) are translated into physical addresses
- Not all variables and data structures are always in the physical memory
  - Each virtual address space is divided into pages that are mapped into and out of the physical memory
  - Virtual memory pages can be mapped out of the physical memory (page-out) to make room
  - Whether or not a variable is in the physical memory is checked at address translation time

Data Transfer and Virtual Memory

- DMA uses physical addresses
  - When cudaMemcpy() copies an array, it is implemented as one or more DMA transfers
  - Address is translated and page presence checked for the entire source and destination regions at the beginning of each DMA transfer
  - No address translation for the rest of the same DMA transfer so that high efficiency can be achieved
- The OS could accidentally page-out the data that is being read or written by a DMA and page-in another virtual page into the same physical location

Data Transfer and Virtual Memory

- Pinned memory are virtual memory pages that are specially marked so that they cannot be paged out
- Allocated with a special system API function call
- a.k.a. Page Locked Memory, Locked Pages, etc.
- CPU memory that serve as the source or destination of a DMA transfer must be allocated as pinned memory

CUDA Data Transfer uses pinned memory

- The DMA used by cudaMemcpy() requires that any source or destination in the host memory is allocated as pinned memory
- If a source or destination of a cudaMemcpy() in the host memory is not allocated in pinned memory, it needs to be **first copied** to a pinned memory – extra overhead
- cudaMemcpy() is faster if the host memory source or destination is allocated in pinned memory since no extra copy is needed

Page-locked Memory

- Placing program data in page-locked pinned memory saves extra data transfers but can be detrimental for the efficiency of the host's virtual memory.

- Pinned memory can be allocated with:
  - malloc() followed by a call to mlock(). Deallocation is done in the reverse order, i.e. munlock() then free().
  - Or by calling the cudaMallocHost() function. Memory allocated in this fashion has to be deallocated with a call to cudaFreeHost().

Page-locked Memory (cont.)

- The performance gain obtained via pinned memory depends on the size of the data to be transferred.
- The gain can range from 10% to a massive 2.5x.

```
cudaError_t cudaMallocHost(
              void ** ptr,  // Addr. of pointer to pinned
                            //   memory (IN/OUT)
              size_t size); // Size in bytes of request (IN)

cudaError_t cudaFreeHost (void * ptr);
```

Bank Conflicts in Shared Memory

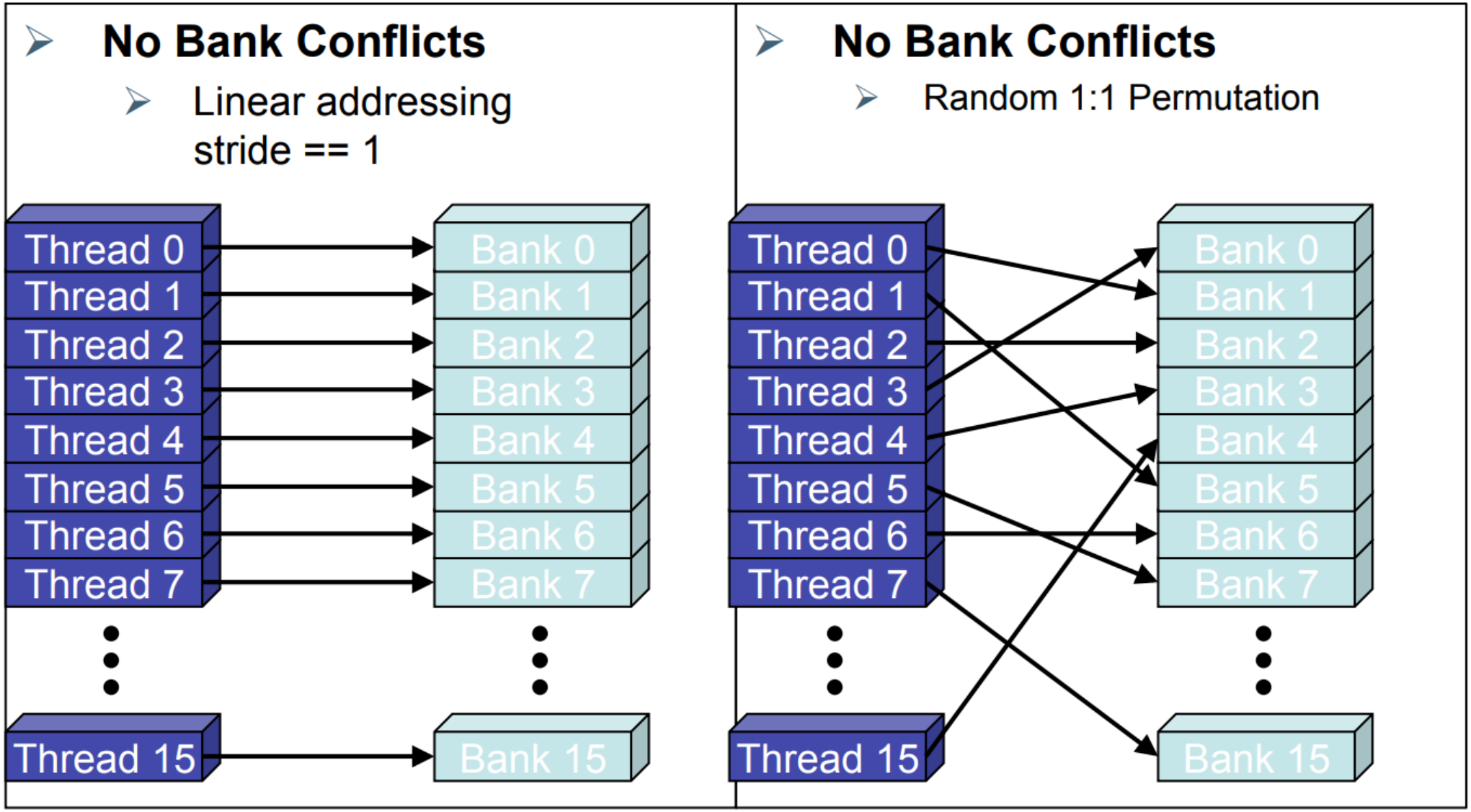
Bank Conflicts in Shared Memory

- Shared memory is split into banks as illustrated below:

Address	Bank 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60
	64	68	72	76	80	84	88	92	96	100	104	108	112	116	120	124
	128	132	136	140	144	148	152	156	160	164	168	172	176	180	184	188

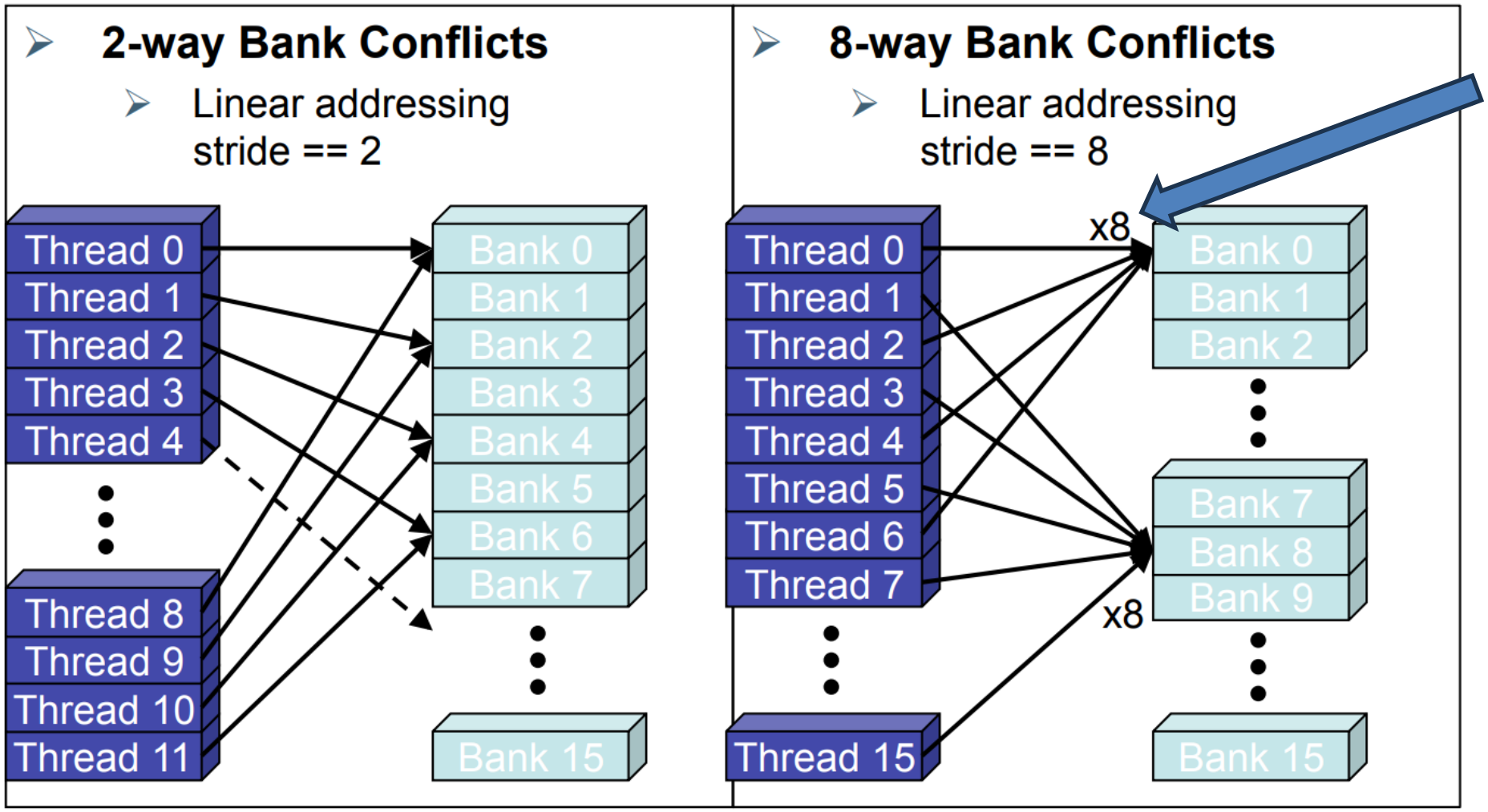
- **Beware:** addresses are interleaved
- Devices of CC 2.0 and above have 32 banks. Earlier devices had 16.
- Each bank can serve one access per cycle
- i.e., if threads access different banks in shared memory, access is instantaneous
- If threads access different data but on the same bank, the access is **serialized**

Examples



https://people.cs.vt.edu/yongcao/teaching/csx984/fall2011/documents/Lecture7_GPU_Memory2.pdf

Examples



Accesses from thread 0,2,4,6,8,10,12,14 all happen on Bank 0. Even if they access different addresses, the accesses are serialized

- One thread waits 1 clock cycle
- One thread waits 2 clock cycles
- ...
- One thread waits 8 clock cycles

https://people.cs.vt.edu/yongcao/teaching/csx984/fall2011/documents/Lecture7_GPU_Memory2.pdf

Take-Home Message

- Threads in a warp/half-warp should avoid accessing at the same time locations in the same shared memory bank

Some extra useful info:

- If all threads access the same memory location, the hardware does a broadcast read (there are no conflicts)
- From CC 2.0, if subsets of threads access the same memory location, the hardware does a multicast read (there are no conflicts)

Global Memory Coalescing

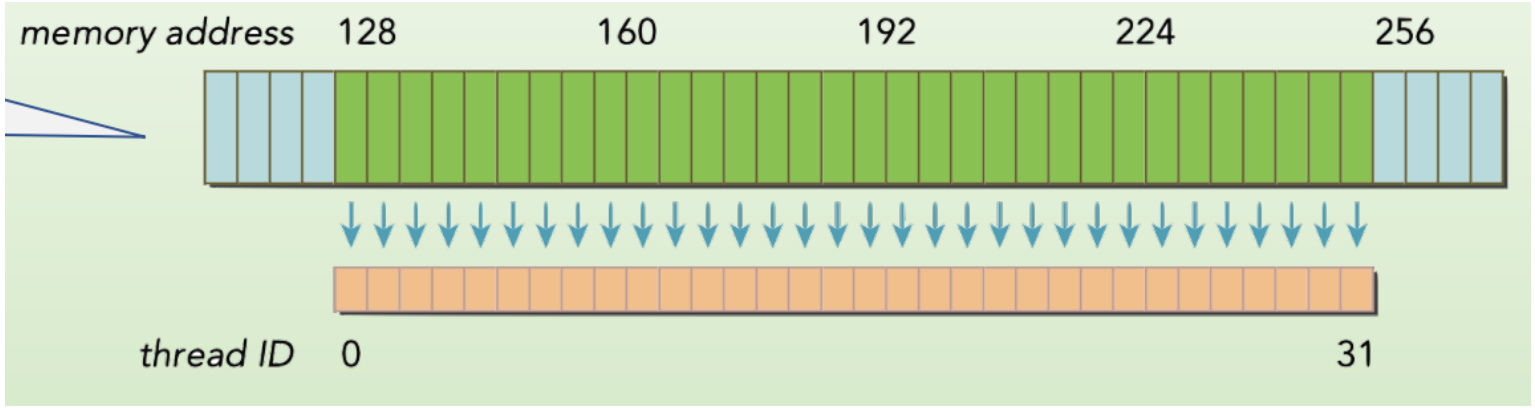
Global Memory Coalescing

- In practice, when a thread accesses a memory location, a *burst* of consecutive locations is actually read (similar to when we load a block of consecutive memory locations into a cache line)
- When all threads in a warp execute a load instruction, the hardware detects if they access consecutive global memory locations
- If this is the case, the hardware **coalesces** all these accesses into a single access
- E.g., for a given load instruction of a warp, if thread O accesses global memory location N2, thread 1 location N+1, thread 2 location N+2, and so on, all these accesses will be coalesced, or combined into a single request for consecutive locations when accessing the DRAMs.
- CUDA devices might impose requirement on the *alignment* of N (e.g., it must be a multiple of 16)

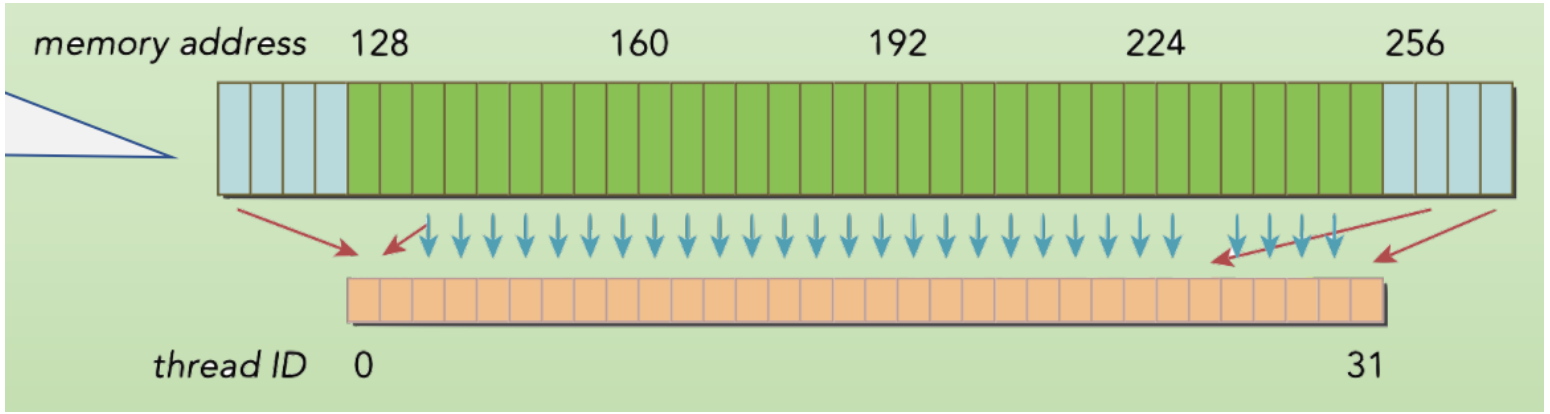
Global Memory Coalescing

- Intuition: if multiple threads in a warp access locations which are close to each other, a single memory **transaction** is issued, reading a **burst** of elements
- Aligned access: the first address of the transaction is a multiple of the cache granularity (usually, 32 bytes for the L2 cache and 128 bytes for the L1)
- Coalesced access: all the 32 threads in a warp access a contiguous memory burst

Accesses are aligned and coalesced
(one single transaction)



Accesses are not aligned and not coalesced
(3 transactions needed)



http://gpu.di.unimi.it/slides/lezione6.pdf

Questions?

Global Memory Accesses

Two types of load:

- Cached loads:
  - Used by default for devices that have L1 caches
  - Check L1, if not present, check L2, if not present, check global memory
  - Load granularity: 128-byte line
- Non-cached loads:
  - If the device does not have the L1 cache
  - Or if it has the L1 cache and you compile with `-Xptxas -dlcm=cg`
  - Check L2, if not present, check global memory
  - Load granularity: 32-byte line

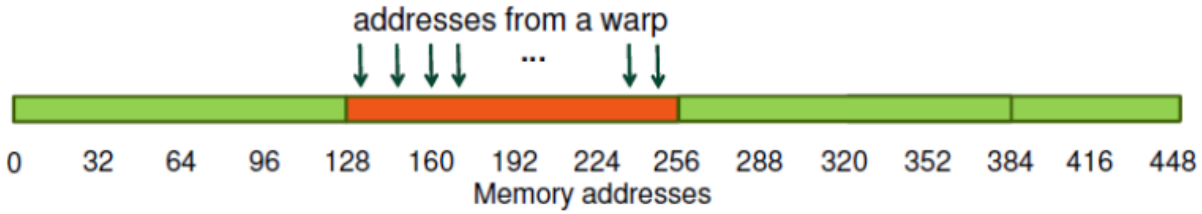
For stores:

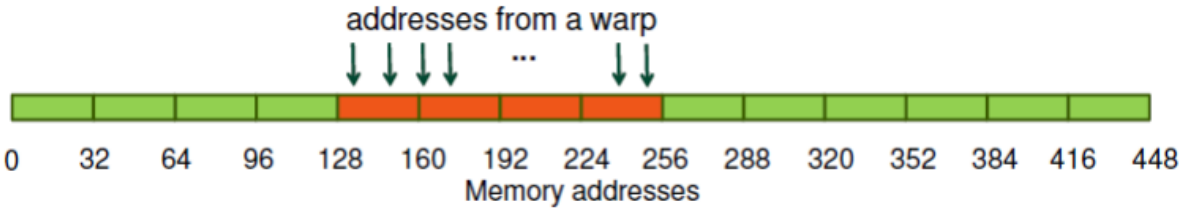
- Invalidate L1, write-back to L2
- (that's why we do not have false sharing)

https://hpc-forge.cineca.it/files/CoursesDev/public/2013/Introduction%20to%20GPGPU%20and%20CUDA%20programming/Roma/CUDA_Day2.pdf

Why should we disable the L1 cache?

Warp requests 32 aligned, consecutive 4-byte words (128 bytes)	
Caching Load	**Non-caching Load**
Addresses fall within 1 cache-line	Addresses fall within 4 segments
128 bytes move across the bus	128 bytes move across the bus
Bus utilization: **100%**	Bus utilization: **100%**

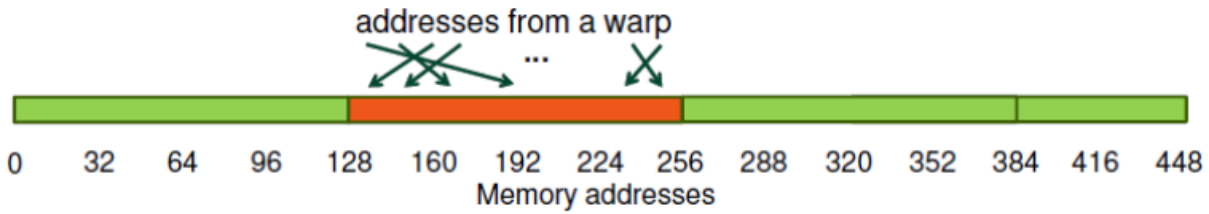


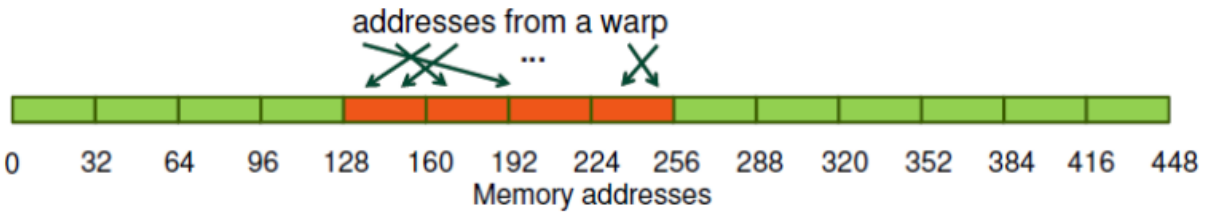


https://hpc-forge.cineca.it/files/CoursesDev/public/2013/Introduction%20to%20GPGPU%20and%20CUDA%20programming/Roma/CUDA_Day2.pdf

Why should we disable the L1 cache?

Warp requests 32 aligned, permuted 4-byte words (128 bytes)	
Caching Load	**Non-caching Load**
Addresses fall within 1 cache-line	Addresses fall within 4 segments
128 bytes move across the bus	128 bytes move across the bus
Bus utilization: **100%**	Bus utilization: **100%**

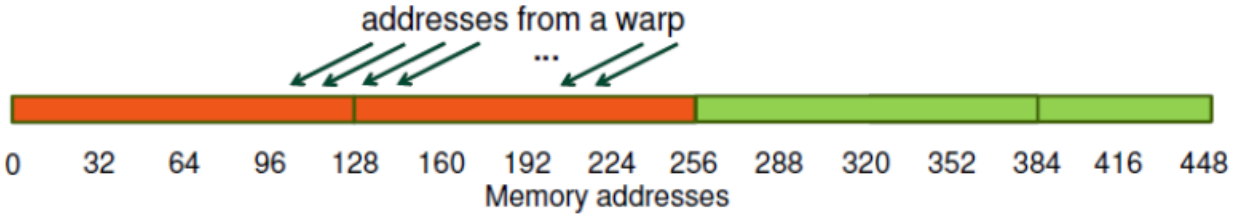


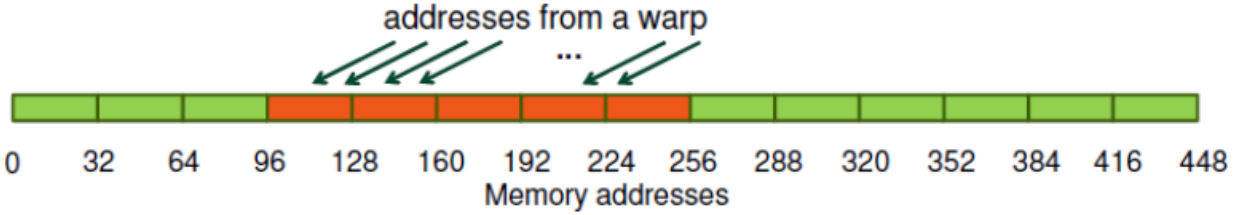


https://hpc-forge.cineca.it/files/CoursesDev/public/2013/Introduction%20to%20GPGPU%20and%20CUDA%20programming/Roma/CUDA_Day2.pdf

Why should we disable the L1 cache?

Warp requests 32 misaligned, consecutive 4-byte words (128 bytes)	
Caching Load	**Non-caching Load**
Addresses fall within 2 cache-lines	Addresses fall within at most 5 segments
256 bytes move across the bus	160 bytes move across the bus
Bus utilization: **50%**	Bus utilization: **at least 80%**

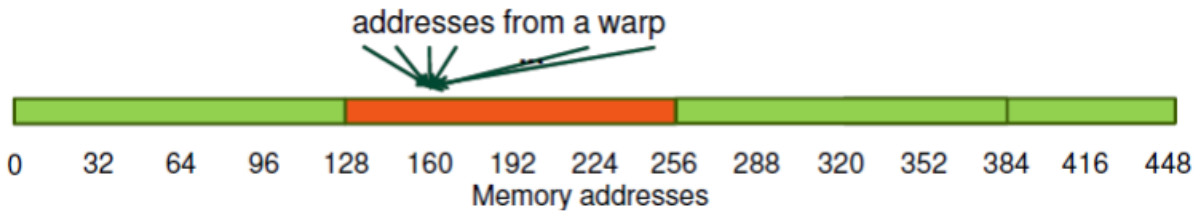


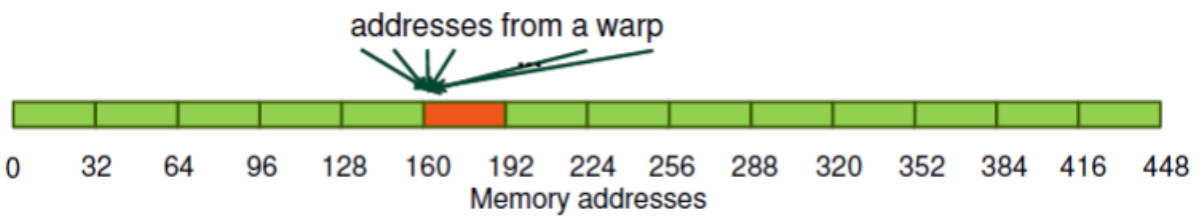


https://hpc-forge.cineca.it/files/CoursesDev/public/2013/Introduction%20to%20GPGPU%20and%20CUDA%20programming/Roma/CUDA_Day2.pdf

Why should we disable the L1 cache?

All threads in a warp request the same 4-byte word (4 bytes)	
Caching Load	**Non-caching Load**
Addresses fall within 1 cache-line	Addresses fall within 1 segments
128 bytes move across the bus	32 bytes move across the bus
Bus utilization: **3.125%**	Bus utilization: **12.5%**

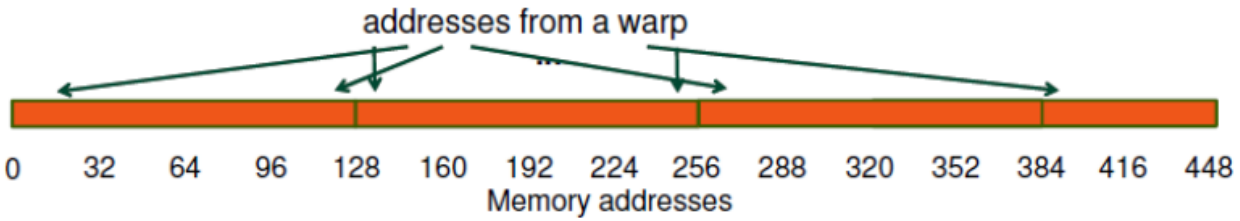


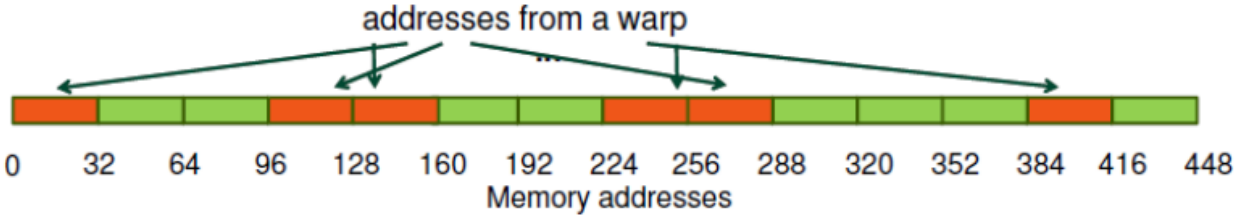


https://hpc-forge.cineca.it/files/CoursesDev/public/2013/Introduction%20to%20GPGPU%20and%20CUDA%20programming/Roma/CUDA_Day2.pdf

Why should we disable the L1 cache?

Warp requests 32 scattered 4-byte words (128 bytes)	
Caching Load	**Non-caching Load**
Addresses fall within N cache-lines	Addresses fall within N segments
N*128 bytes move across the bus	N*32 bytes move across the bus
Bus utilization: **128 / (N*128)**	Bus utilization: **128 / (N*32)**





https://hpc-forge.cineca.it/files/CoursesDev/public/2013/Introduction%20to%20GPGPU%20and%20CUDA%20programming/Roma/CUDA_Day2.pdf

Summing up

If you have non-coalesced or non-aligned memory accesses, it might be worth considering disabling the L1 cache

https://hpc-forge.cineca.it/files/CoursesDev/public/2013/Introduction%20to%20GPGPU%20and%20CUDA%20programming/Roma/CUDA_Day2.pdf

Questions?

Importance of data structure organization for coalesced accesses

Array of Struct (AoS) vs. Struct of Array (SoA)

- Better to use an array, in which each element is a struct, or a struct, in which each element is an array?

```
struct innerStruct {
    float x;
    float y;
};
. . .
struct innerStruct AoS[N];
```

```
struct innerArray {
    float x[N];
    float y[N];
};
. . .
struct innerArray SoA;
```

- How is the data organized in memory?
- Cache locality?
- Alignment and coalesced accessed?

http://gpu.di.unimi.it/slides/lezione6.pdf

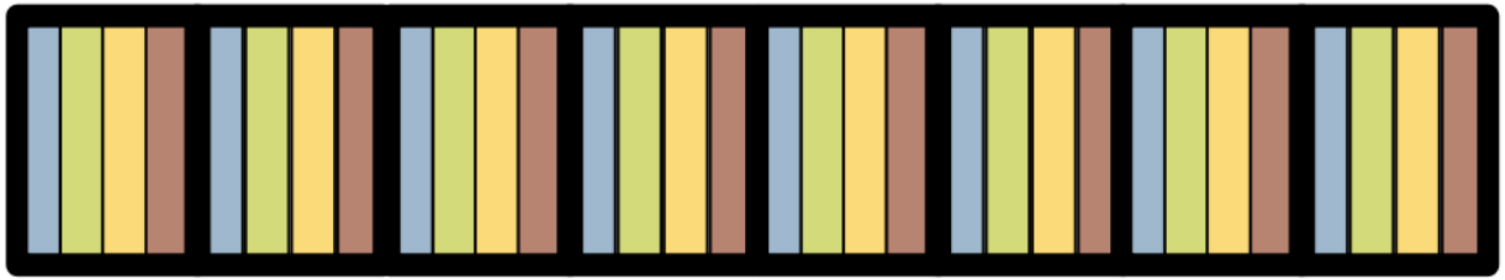
Array of Struct (AoS) vs. Struct of Array (SoA)

Structure of Arrays (SoA)

```
struct foo{
 float a[8];
 float b[8];
 float c[8];
 int d[8];
} A;
```

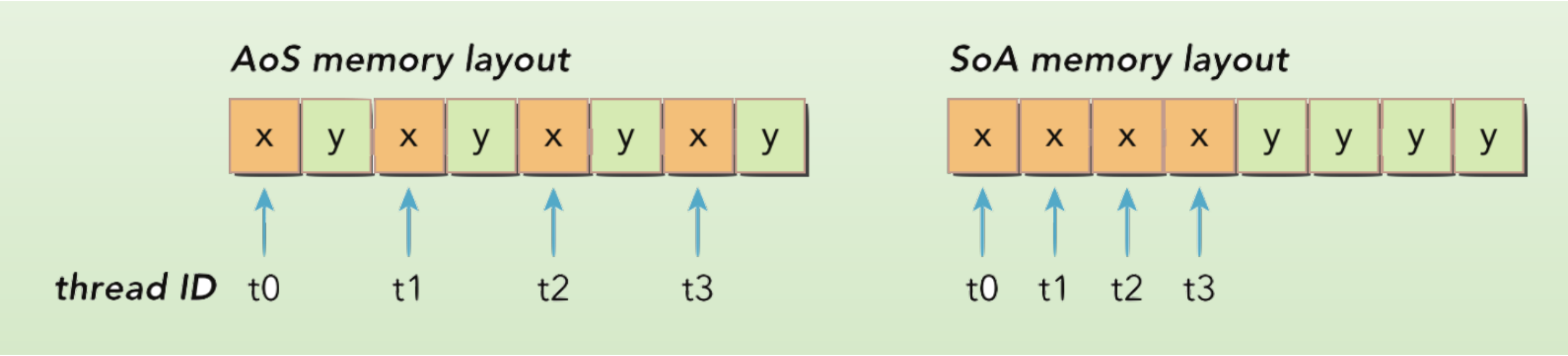
Array of Structures (AoS)

```
struct foo{
 float a;
 float b;
 float c;
 int d;
} A[8];
```



http://gpu.di.unimi.it/slides/lezione6.pdf

Array of Struct (AoS) vs. Struct of Array (SoA)



- AoS would waste space in the cache due to unneeded y values
- With SoA, we only bring in bursts of data we need (i.e., burst only containing 'x' values)
- SoA enables coalesced accesses
- SoA might also require less space (AoS might have padding after each struct)

http://gpu.di.unimi.it/slides/lezione6.pdf

CUDA topics we did not cover

CUDA topics we did not cover

- Asynchronous CUDA operations: streams & events
- Unified memory
- Cooperative groups
- Dynamic parallelism
- …

What Next?

What next?

We barely scratched the surface, I hope to leave you with more questions than answers :)

- How do collectives work?
- How data actually moves on the network? (Not through TCP!)
- How nodes in a cluster are actually connected?
- How GPUs on a node are actually connected?
- How do these new emerging architectures work? (Cerebras, Tenstorrent, TPUs, etc...)
- How can compilers optimize the code?
- Tensor cores on the GPUs
- How are ML models actually trained on 100,000 GPUs?
- ... and much more ...

How to learn more about this topic?

- Internship
- Join the Student Cluster Competition team preparation sessions
- CINECA offers free training courses
- Prof. Pontarelli course on «Architectures for AI» -- for the BSc in «Scienze Matematiche per l'Intelligenza Artificiale»
- Master Degree in Computer Science:
  - Big Data Compute (the syllabus might change next year) - 6 CFUs
  - «Attività Formativa Complementare» (AFC) - 6 CFUs
  - Advanced Architectures - 6 CFUs
  - I can suggest elective courses to be taken from other MSc


Happy Holidays