# OPTIMIZATIONS

Memory components, loop unrolling, thread shuffle, ninja functions, memory access…

# General *strategy*


APOD

# 1. Assessment

- Identify bottlenecks
- High priority : parallelizable loop structures with lots of computation. Even in already host-parallelized sections.

# 2. Parallelize

- Cuda libraries
- Kernels
- OpenACC

# 3. Optimization

## 3.1 Grid Level

- Concurrency, multiple kernels at the same time

- Overlapping computation and data transfers…

# 3. Optimization

## 3.2 Kernel Level

- Memory coalescence & efficient access patterns
- Reduce memory access latency
- Reduce wrap divergence

# 4. Deployment

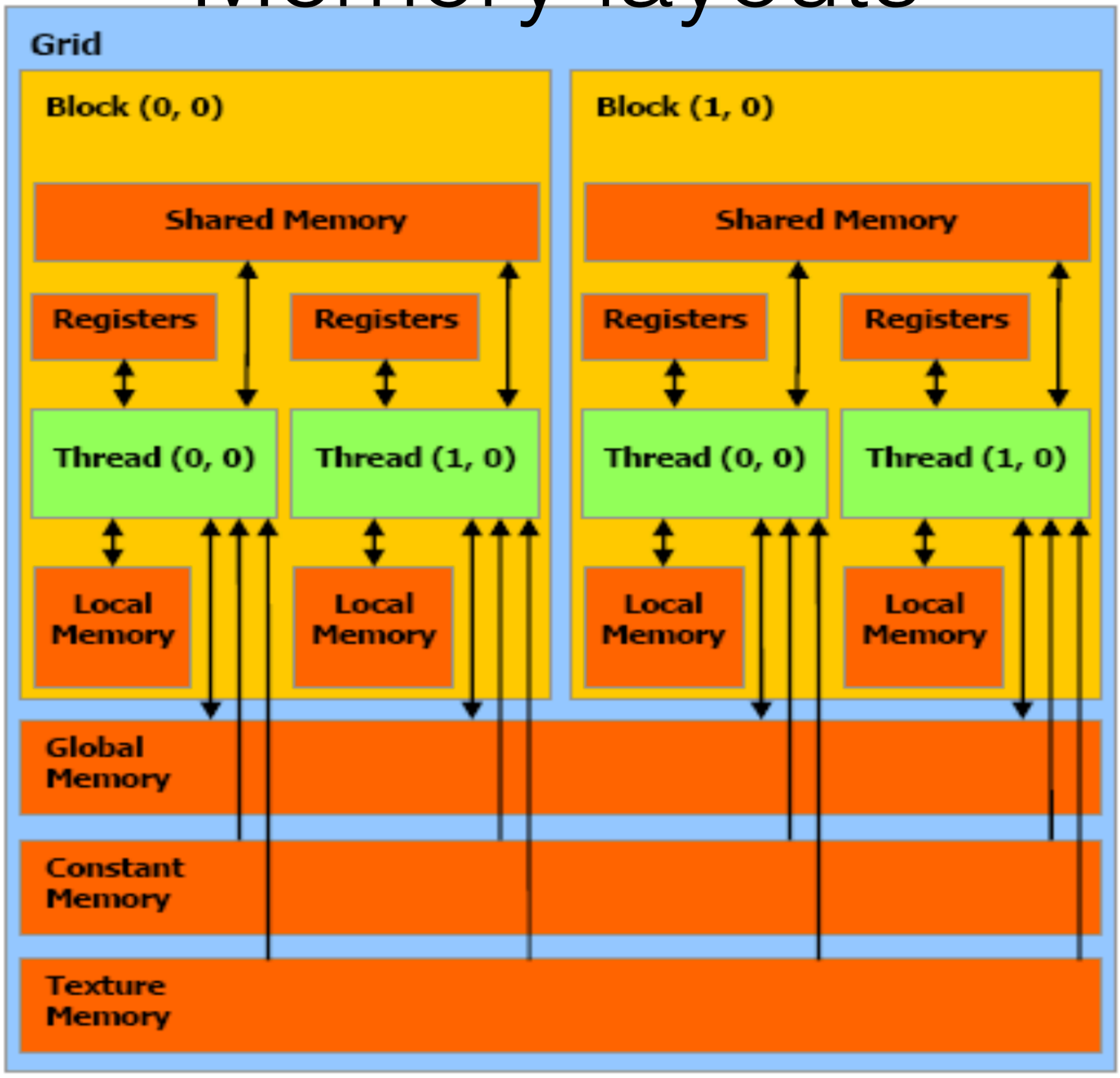
- Ensure it can run even on non CUDA-capable GPU environnement

- Iterate once again

# 3.2 Kernel optimization

- Two further optimisations directions : is you kernel :
  - Memory bound ? (most of the time)
  - Computation bound ?

# Memory layouts



- Preliminary notes :
- Test in folder OPTIMIZATIONS
- All tests, profiling… should be done in  calculation platform
- Using nvprof on cards use for desktop is not accurate

# 3.2.1 Memory bound kernels

- Goals
- Improve memory access (bytes / sec on the bus)
- Improve concurrent memory access
- Hide latency
- NOTE : memory requests are issued per wrap !

# Types of readings

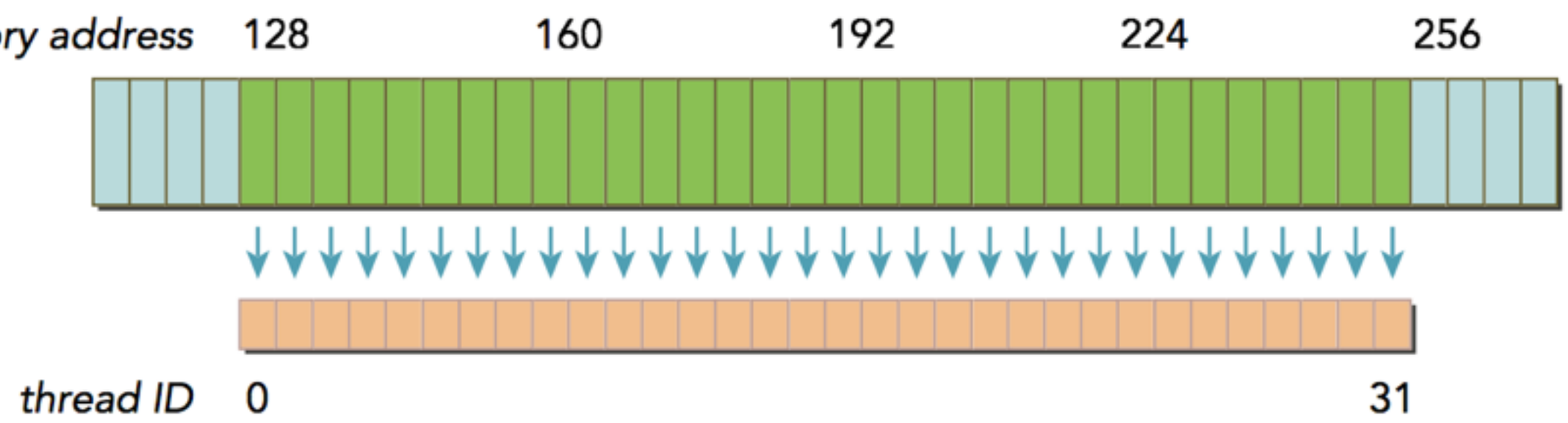
- Cached
- Uncached
- Read-only
- Exemples in OPTIMIZATION folder

# 3.2.1 Memory bound kernels Load granularity

- Cached : 128 bytes cache-line
- Uncached & read-only : 32 bytes
- Can be chosen at compile time :
  - -Xptxas -dlcm=cg : disabled
  - -Xptxas -dlcm=ca : enabled

# Load granularity : L1-cached

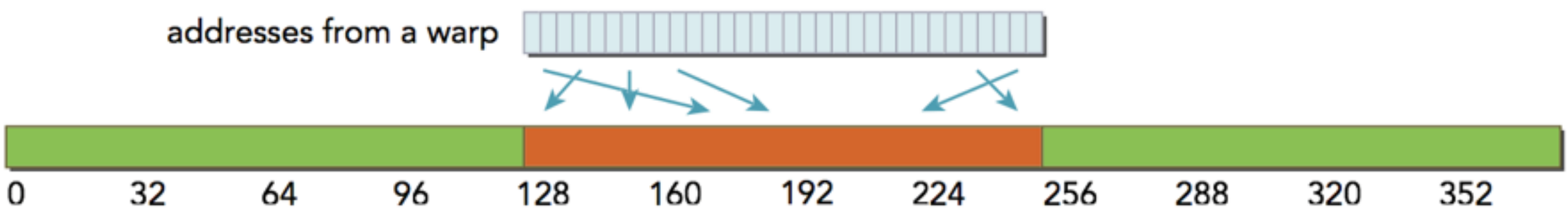
- Best memory transactions : *aligned* & *coalesced.* Better with 32 times (a wrap) 4 bytes ( = one 128bytes cache-line)




Aligned & coalesced reading.
If not, several 128-bytes cache lines must be read : *wasted* bandwidth.

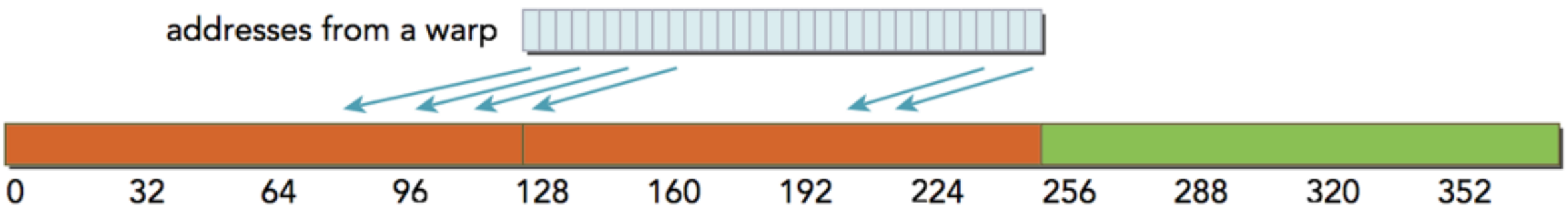
# Load granularity : L1-cached

- Still ok if order is shuffled : line is cached, only one transaction needed, bus use = 100%



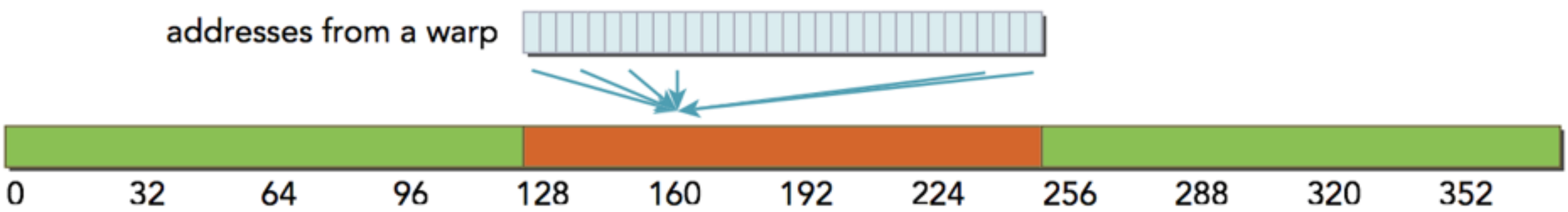
# Load granularity : L1-cached

- Misaligned reading : two lines read. bus use = 50%



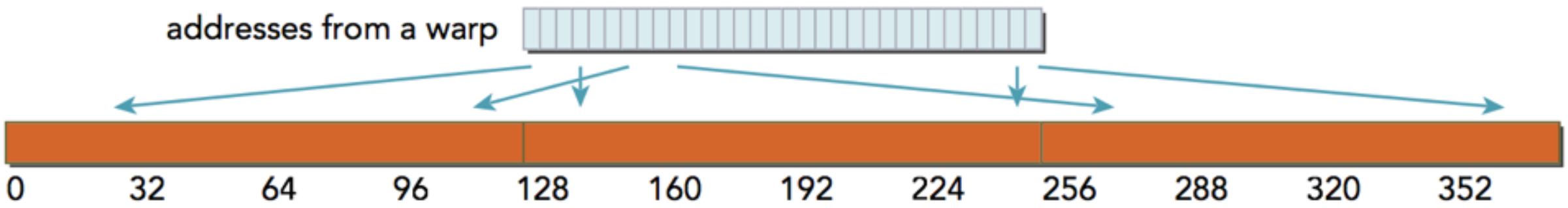
# Load granularity : L1-cached

- Only one element read : two lines read. bus use = 1/32 ~ 3.1 %



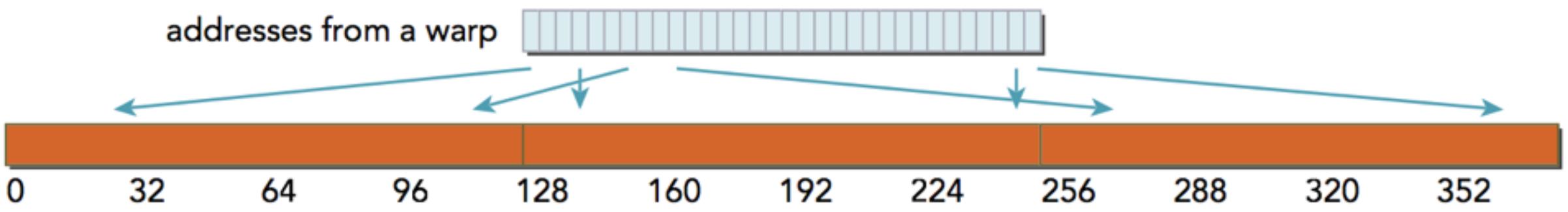
# Load granularity : L1-cached

- Very bad access : up to 32 lines read



# Load granularity : L1-cached

- Very bad access : up to 32 x 128bytes-lines read

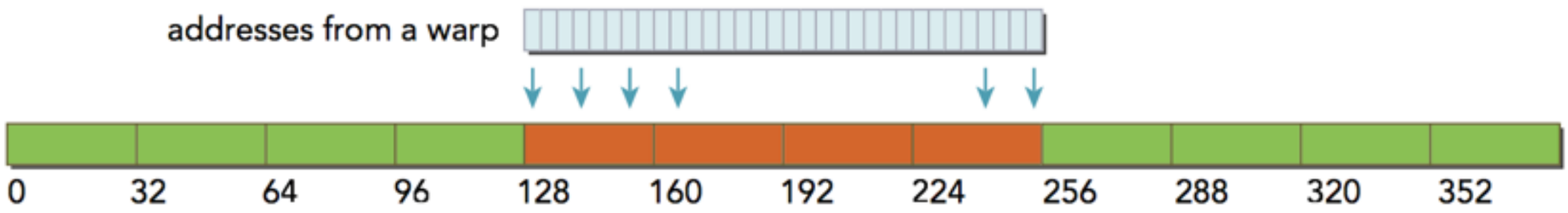


# Load granularity : L1-cached

- Note : difference between CPU and GPU L1 cache : no temporal locality (frequent access does not imply data staying in cache)

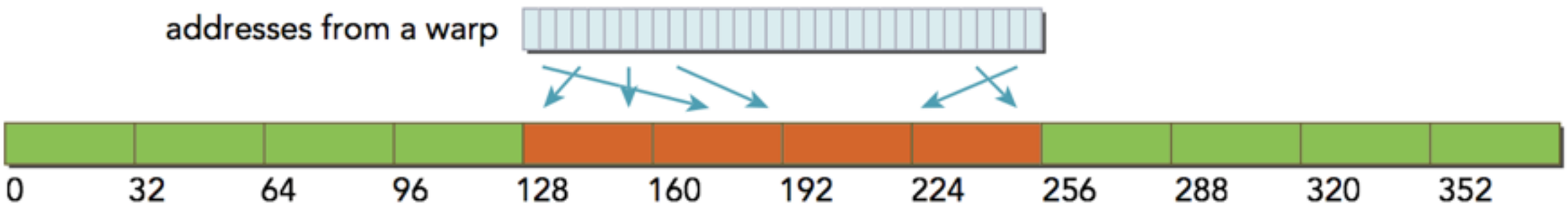
# Load granularity : uncached

- Coalesced & aligned access for 32 bytes lines



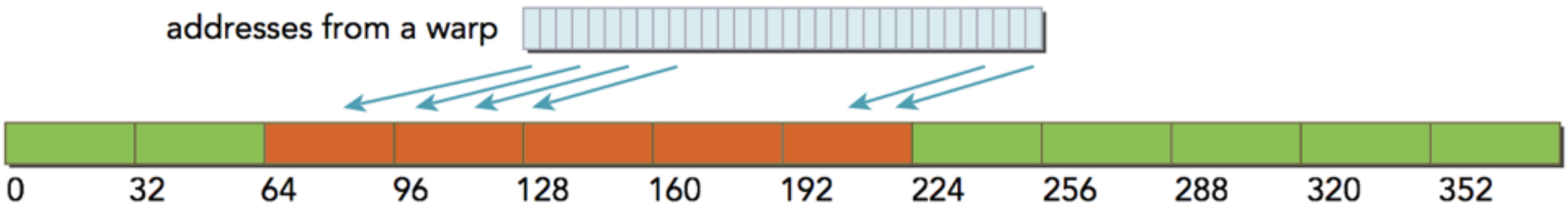
# Load granularity : uncached

- Shuffled acces : ok



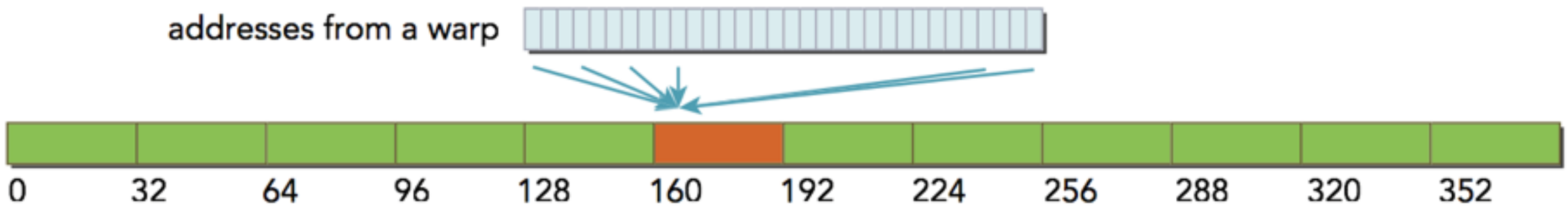
# Load granularity : uncached

- Mis-aligned : bus use at least 80 %. Better than with L1-cache



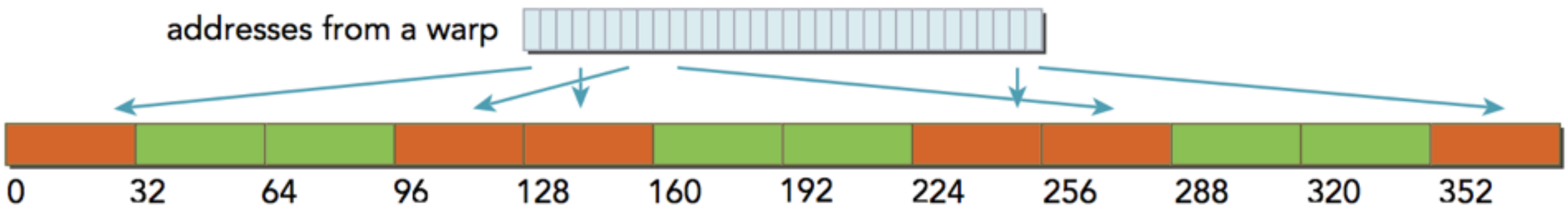
# Load granularity : uncached

- Only one element read : bus use = 12.5 %. 4 x better than with L1 cache



# Load granularity : uncached

- Very bad access : up to 32 x 32 bytes read, still better



# Read-only cache : 32 bytes

- After CC 3.5 : can choose to use read-only with :
  - function _ldg
  - const __restrict__ on pointers

# readSegment.cu

- usage : readSegment <offset>
- Try different offsets. Do you see a difference ?
- Use nvprof —metrics gld_efficiency readSegment <offset>

- Try the same thing by disabling the L1 cache
- Compile with the flag -Xptxas -dlcm=cg

# 3.2.1 Memory bound kernels writeSegment.cu

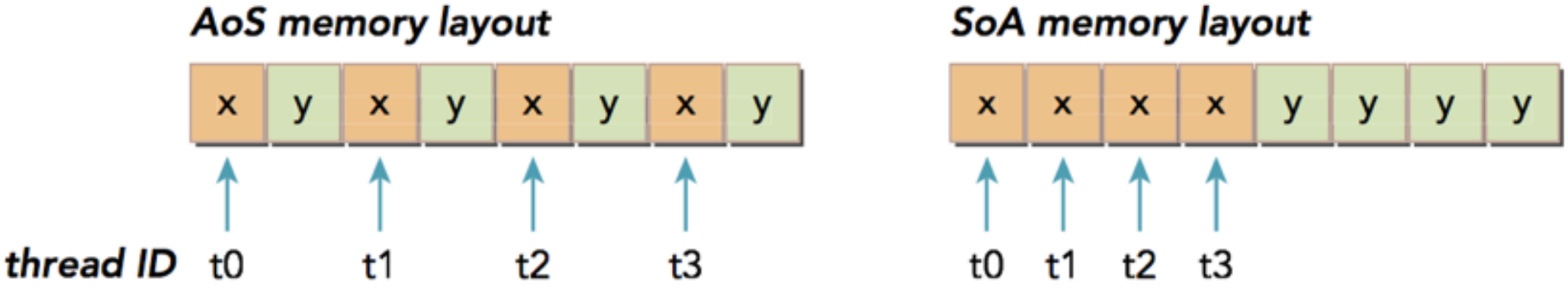
- Stores are performed at a 32-byte segment granularity

- nvprof —metrics gld_efficiency —metrics gst_efficiency ./writeSegment

# 3.2.3 AoS vs SoA

./simpleMathAoS and simpleMathSoA examples

# Array of structures (AoS) vs Structure of arrays (SoA)



# ./simpleMathAoS and simpleMathSoA examples

- Compare the load efficiencies between the two

# 3.2.4 Shared memory

- On-chip, superfast : good for reuse. Shared *per-block.*

- Organised in *banks* of either 4 or 8 bytes (you can choose, will affect performance).

- Access of different words from same bank : *bank conflict*.

- Solution : spread memory, pad arrays.

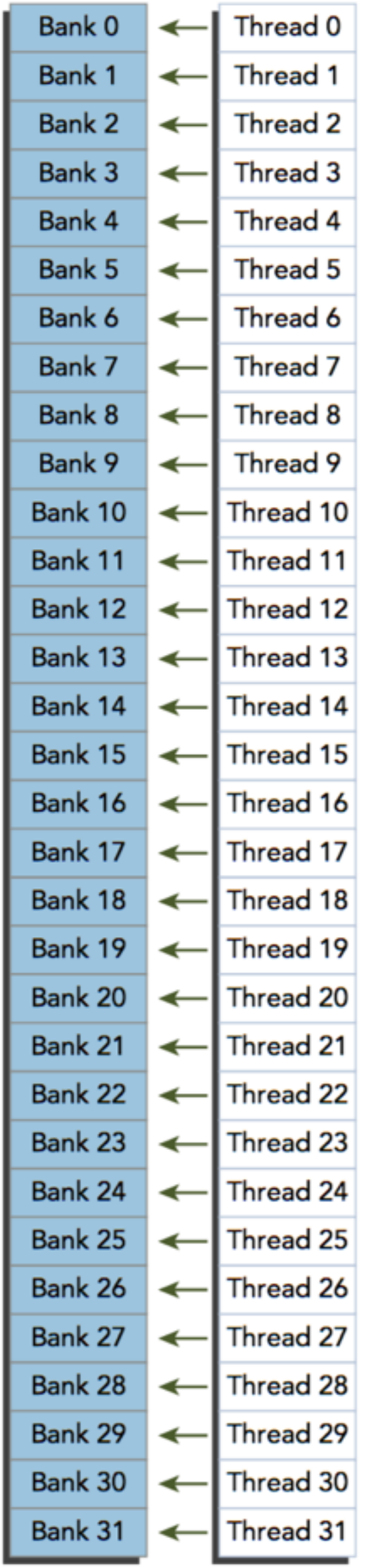
# Types of access

- Parallel access: multiple addresses accessed across multiple banks

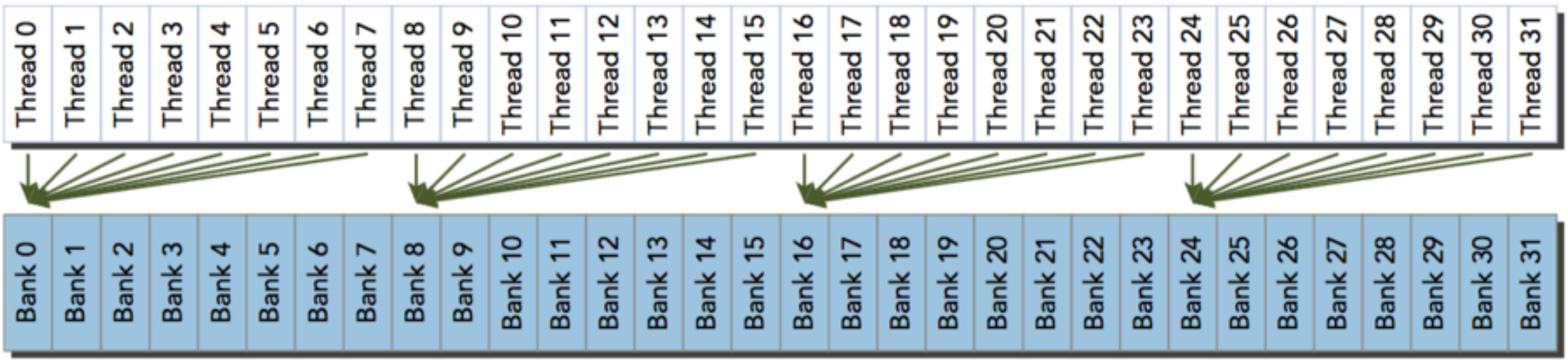
- Serial access: multiple addresses accessed within the same bank

- Broadcast access: a single address read in a single bank

Parallel access



Bank conflict : multiplies number of serial transactions
No conflict for threads in the same warp



# Allocations

- Inside a kernel :
  - __shared__ float array[N];

# Configuration

- Choose for the whole device
- Choose also per kernel
- Depends on your kernel : use more share memory or more registers ?

# Configuration

- For the whole application : cudaError_t cudaDeviceSetCacheConfig(cudaFuncCache cacheConfig);
  - cudaFuncCachePreferNone: no preference(default)
  - cudaFuncCachePreferShared: prefer 48KB shared memory and 16 KB L1 cache
  - cudaFuncCachePreferL1: prefer 48KB L1 cache and 16 KB shared memory
  - cudaFuncCachePreferEqual: prefer 32KB L1 cache and 32 KB shared memory

# Configuration

- For a particular kernel : cudaError_t cudaFuncSetCacheConfig(const void* func, enum cudaFuncCacheca cheConfig);

# 3.2.1 Memory bound kernels Register spilling

- When more register than max allowed per thread : spill to global memory (local memory on Keplers).

# 3.2.2. Instruction bound : toward occupancy

## sumMatrix example

- Goal : saturate instruction bandwidth and memory bandwidth

- Vocabulary : number of active wraps : parallelism exposed to an SM (streaming multiprocessor).

- Attention : high occupancy -> not always higher perf

# sumMatrix.cu

- Tune the blocksize
  - Example in sumMatrix
    - ./sumMatrix <blockXsize> <blockYsize>
- Try following combinations : [16 32], [32 32], [16 16] , [32 16]

# Measuring occupancy

- nvprof —quey-metrics to see a list of supported metrics

- Use nvprof —metrics achieved_occupancy to see the resulting occupancy for each cases

# Load throughput

- nvprof —metrics gld_throughput

- Which combination has the highest throughput ?

# Load efficiency

- nvprof —metrics gld_efficiency

- Which combination has the highest load efficiency ?

## Other good practices

# Exposing more parallelism

- Loop unrolling : gains up to 600%

# « School case » example : reduction

http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

The pdf is in  SLIDES/DOCS folder

Also example at OPTIMIZATIONS/reduceInteger