GPU Computing: Optimisation Approaches

Dr Paul Richmond
http://paulrichmond.shef.ac.uk

Performance Inhibitors

- Data transfer to/from device memory
- Device underutilisation
- Code Branching
- GPU memory bandwidth

Data Transfer

- CPU (host) and GPU (device) have separate dedicated memory
- All data read/written on the device must be copied via PCIe bus
  - Very expensive operation
- **Optimisation Technique**: Minimise data copies
  - Keep resident data on the device
  - May have to move some computation to the GPU even if is not computationally expensive
  - Might be quicker to re-calculate data on the device than copy it

Data Transfer Example

```
Loop over timesteps
   inexpensive_routine_on_host(data_on_host)
   copy data from host to device
   expensive_routine_on_device(data_on_device)
   copy data from device to host
End loop over timesteps
```

- Port inexpensive routine to the device
- Minimise transfers by moving copy out of the loop

```
copy data from host to device
Loop over timesteps
   inexpensive_routine_on_device(data_on_device)
   expensive_routine_on_device(data_on_device)
End loop over timesteps
copy data from device to host
```

Performance Inhibitors

- Data transfer to/from device memory
- Device underutilisation
- Code Branching
- GPU memory bandwidth

Exposing Parallelism

- GPU performance relies on the use of many threads
  - Degree of parallelism must be much higher than on the CPU
  - Ideally need **many** more threads than cores
- Effort must be made to expose as much parallelism as possible
  - May require re-engineering your problem
- If significant sections of code are serial then GPU acceleration will be limited
  - Amdahl's Law

$$Speedup\ (N) = \frac{1}{B + \frac{1}{N}(1 - B)}$$

Memory Latency

- Access to GPU memory has several hundred cycles of latency
  - When a thread is waiting for data it is stalled
- GPUs have very fast context switching
  - Stalled threads can be switched with active threads
  - Switching hides memory latency if other threads are performing compute
  - Requires many threads ideally performing large amounts of computation
- **Optimisation Technique**: Have lots of threads with high arithmetic intensity
  - Defined as the ratio of arithmetic computation to memory accesses

Exposing parallelism example

```
Loop over i from 1 to 512
   Loop over j from 1 to 512
        independent iteration
```

Original code

1D decomposition

```
Calc i from thread/block ID
   Loop over j from 1 to 512
        independent iteration
```

512 threads

2D decomposition

```
Calc i & j from thread/block ID
        independent iteration
```

262,144 threads

The University Of Sheffield.

NVIDIA GPU RESEARCH CENTER

Performance Inhibitors

- Data transfer to/from device memory
- Device underutilisation
- Code Branching
- GPU memory bandwidth

Code Branching

- On NVIDIA GPUs there are less instructional scheduling units than cores
- Threads are scheduled in groups of 32 (a warp)
- Threads within a warp execute the same instruction in lock-step
  - Single Instruction Multiple Data (SIMD)
- CUDA C Kernels are free to specify branches
  - BUT **all** threads will have to follow **all** code paths within the warp
- **Optimisation Technique**: Avoid inter warp branching wherever possible

Branching Example

- You want to split your threads into two groups:

```
i = blockIdx.x*blockDim.x + threadIdx.x;
if (i%2 == 0)
   …
else
   …
```

```
i = blockIdx.x*blockDim.x + threadIdx.x;
if ((i/32)%2 == 0)
   …
else
   …
```

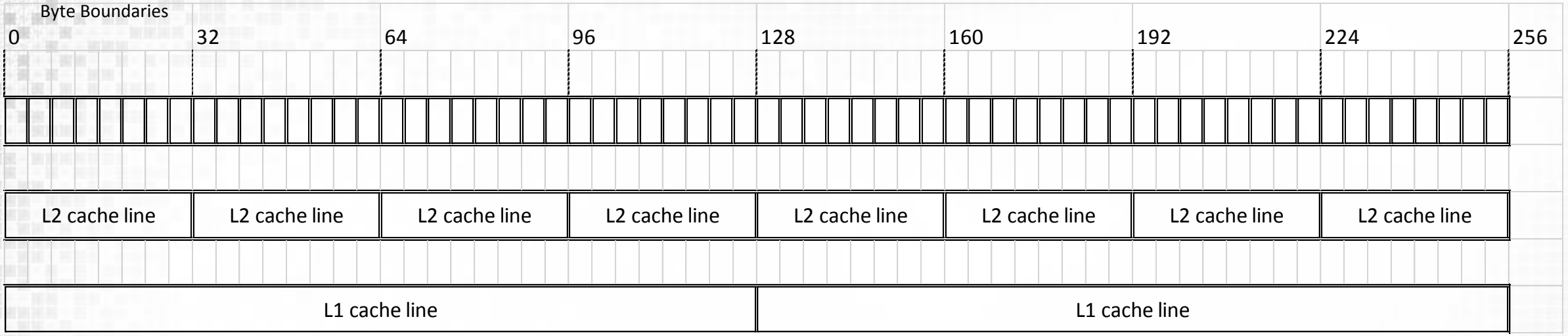
Performance Inhibitors

- Data transfer to/from device memory
- Device underutilisation
- Code Branching
- GPU memory bandwidth

Coalesced Global Memory Access

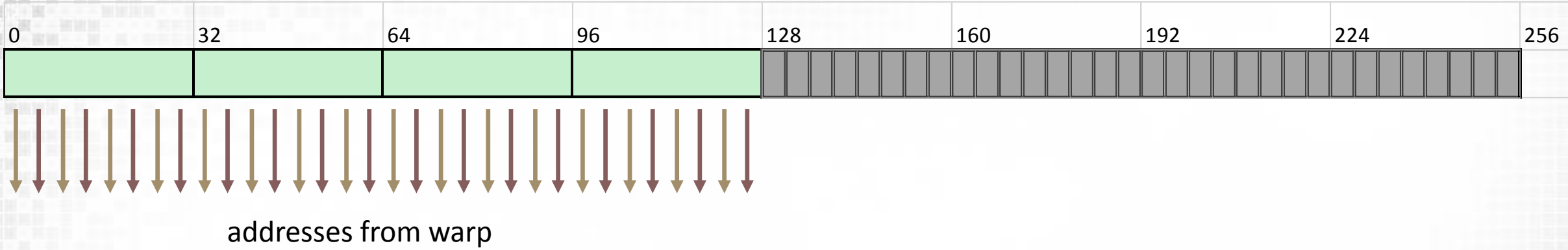
- When memory is loaded/stored from global memory to L2 and L1 it is moved in cache lines
  - If threads within a warp access global memory in irregular patterns this can cause increased movement of data
- Coalesced access is where sequential threads in a warp access sequentially adjacent 4 byte words (e.g. `float` or `int` values).
  - Having coalesced access will reduce the number of cache lines moved and increase memory performance
  - This is one of the most important performance considerations of GPU memory usage!

Use of Memory Cache Levels



- L1 Cache
  - 128B wide cache line transactions
  - Normally used for thread local variables
  - Can also be used for global loads (by default for Compute 2.x)
    - Via L2 cache first
    - Compute 3.5, 3.7 or 5.2 have opt in L1 caching
    - Early Maxwell (Compute 5.0 cant opt in for L1 global loads)
- L2 Cache
  - 32B wide cache line transactions
  - **All** global and local memory pass through

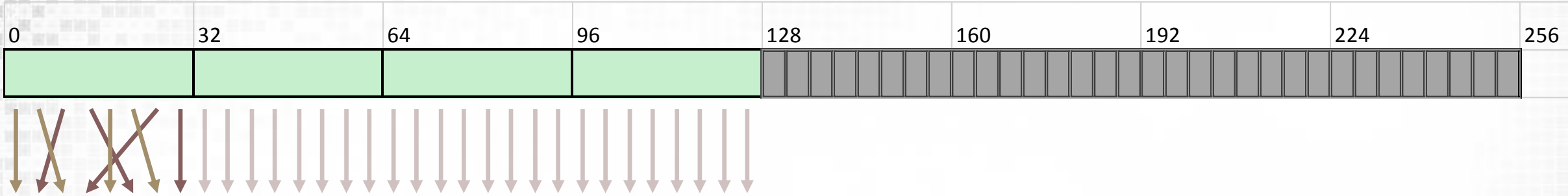
L2 Coalesced Memory Access




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

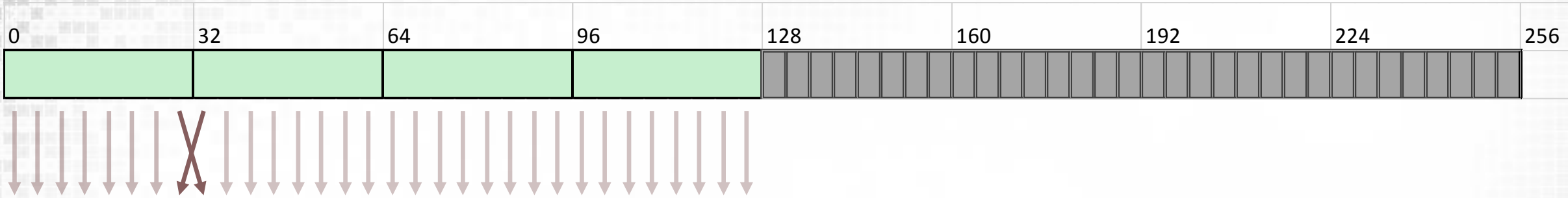
- Global memory always moves through L2
  - But not always through L1 depending on architecture
- In L2 cache line size is 32B
  - For a coalesced read/write within a warp, 4 transactions required
  - 100% memory bus speed

L2 Permuted Memory Access



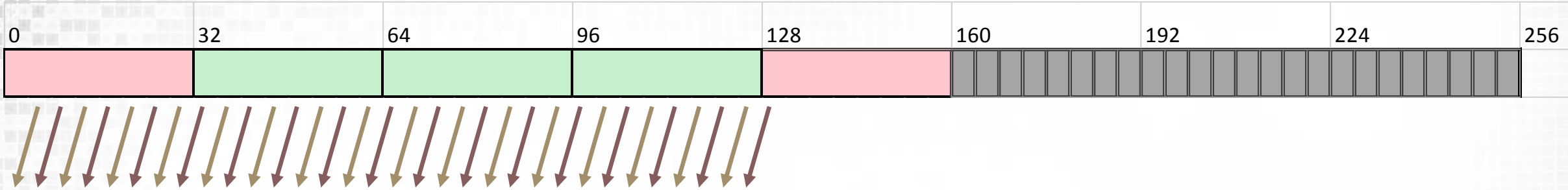
- Permuted Access
  - Within the cache line accesses can be permuted between threads
  - No performance penalty

L2 Permuted Memory Access



- Permuted Access
  - Permuted access within 128 byte segments is permitted
    - Will NOT cause multiple loads
    - Must not be permuted over the 128 byte boundary

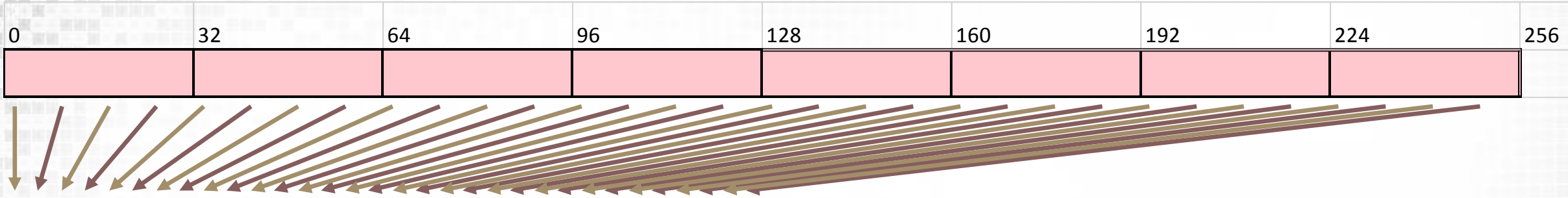
L2 Offset Memory Access




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

- If memory accesses are offset then parts of the cache line will be unused (shown in red) e.g.
  - 5 transactions of 160B of which 128B is required: 80% utilisation
- Use thread blocks sizes of multiples of 32!

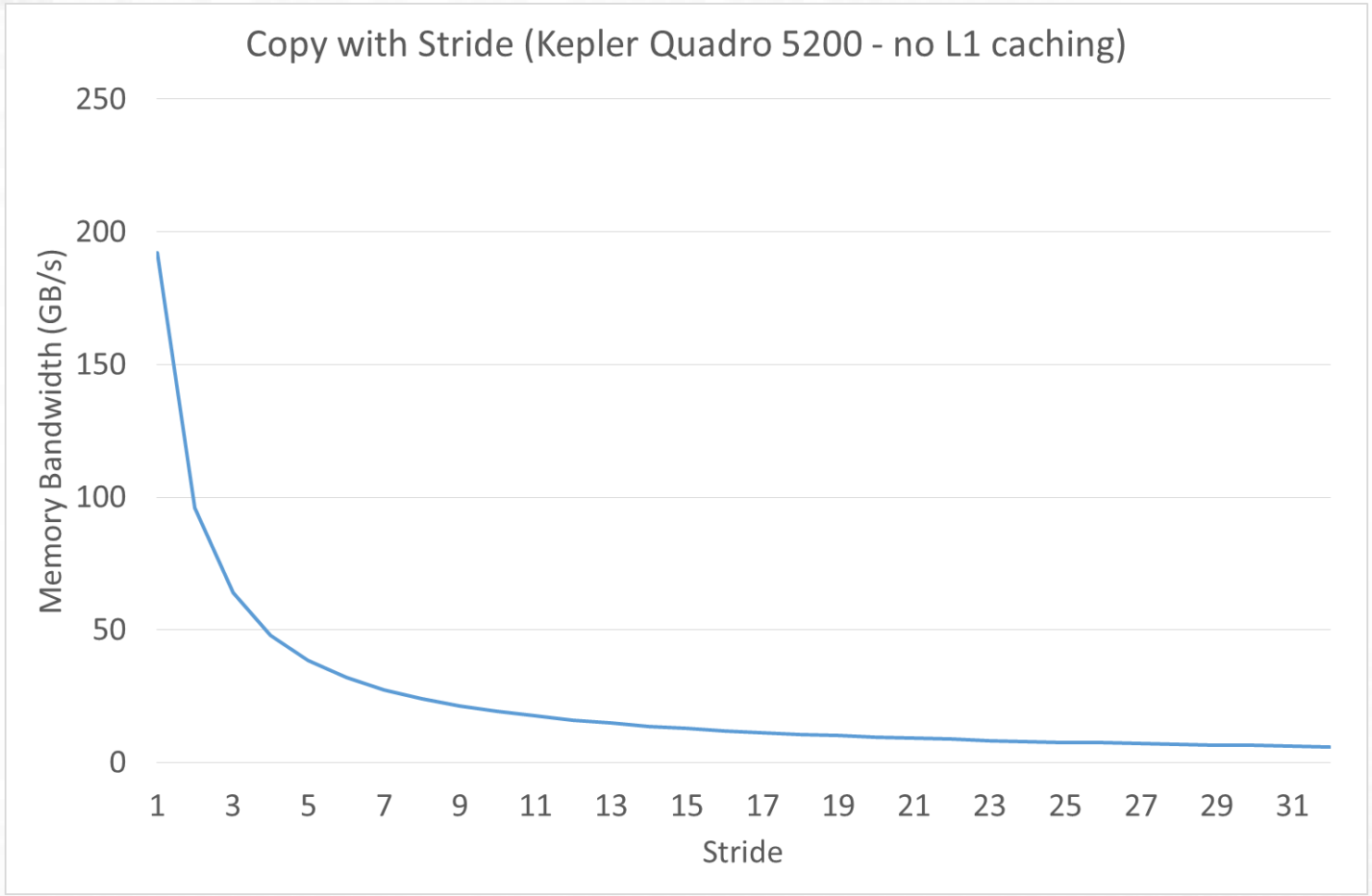
L2 Strided Memory Access




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

- Strided memory access can result in bad performance e.g.
  - A stride of 2 causes 8 transactions: 50% useful memory bandwidth
  - As stride of >32 causes 32 transactions: ONLY 3.125% bus utilisation!
    - This is as bad as random access
    - Transpose data if it is stride-N

Degradation in Strided Access Performance



- Note: Performance worsens beyond a stride of just 8 as adjacent or concurrent warps (on same SM) can't re-use cache lines from L2

Summary

- For good performance the following guidelines should be considered
  - Minimise data transfer to and from the device
  - Ensure the device is kept busy by providing lots of parallelism (> threads)
  - Avoid conditional statements and branches in kernels
  - Align memory accesses to 32B boundaries