Parallel Computing with GPUs

Performance
Part 1 – Memory Coalescing

The University Of Sheffield.

Dr Paul Richmond
http://paulrichmond.shef.ac.uk/teaching/COM4521/

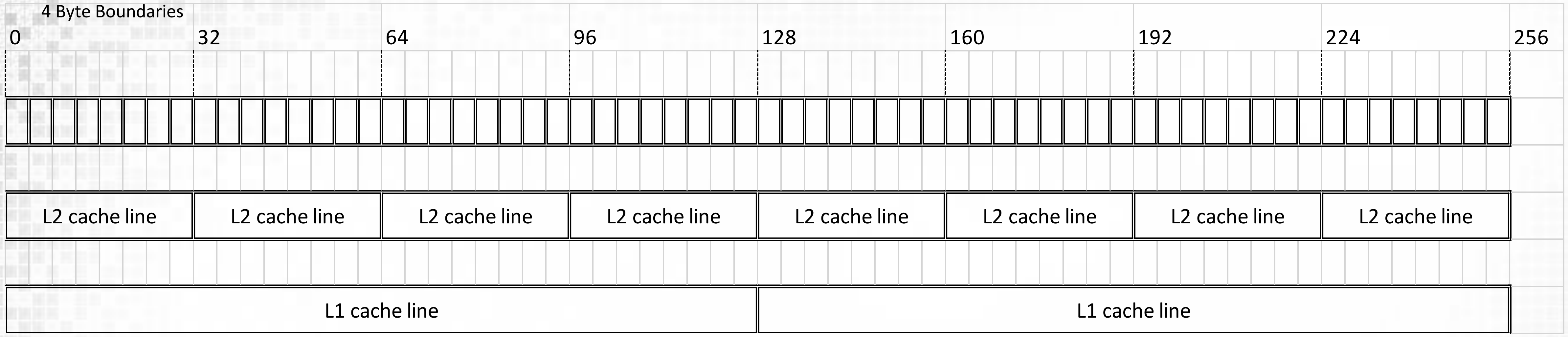
This Lecture (learning objectives)

- ❑Memory Coalescing
  - ❑Explain the process of memory moving via cache lines
  - ❑Analyse the performance implications of strided access patterns
  - ❑Classify use cases of memory access with respect to performance

Coalesced Global Memory Access

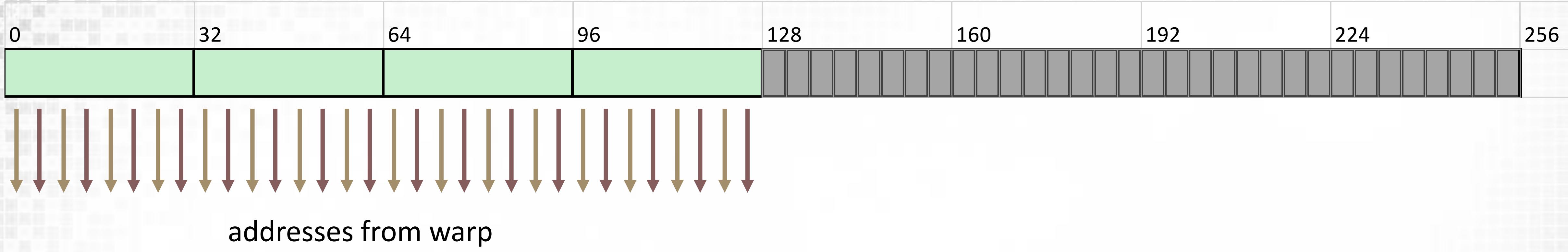
- ❑When memory is loaded/stored from global memory to L2 (and L1) it is moved in cache lines
  - ❑If threads within a warp access global memory in irregular patterns this can cause increased movement (transactions) of data
- ❑Coalesced access is where sequential threads in a warp access sequentially 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 (via read-only cache method from previous lecture)
  - ❑Some hardware has L1 cache for all memory reads
    - ❑ Always via L2 cache first
    - ❑ Compute **3.5**, 3.7 or 5.2 have opt in global 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

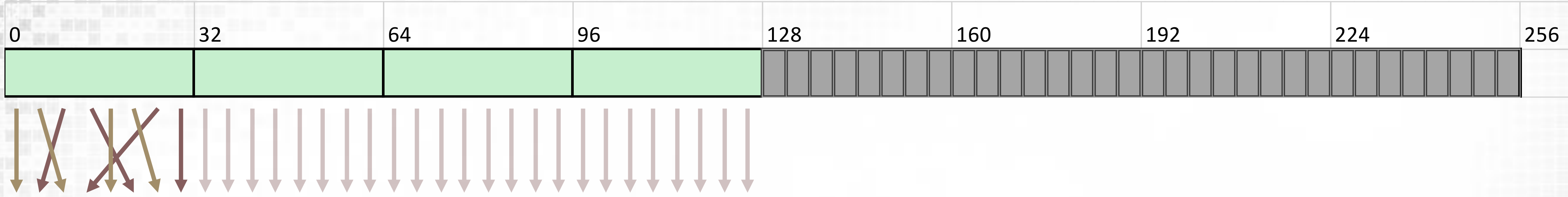
L2 Coalesced Memory Access



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

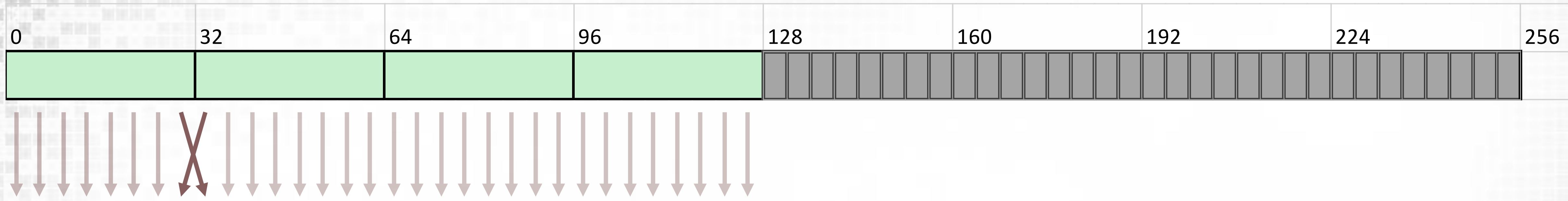
- Global memory always moves through L2
  - But not always through L1 depending on architecture
- In L2 cache line size is 32B
  - For a coalesced read/write within a warp, 4 transactions required
  - 100% memory bus speed

L2 Permuted Memory Access



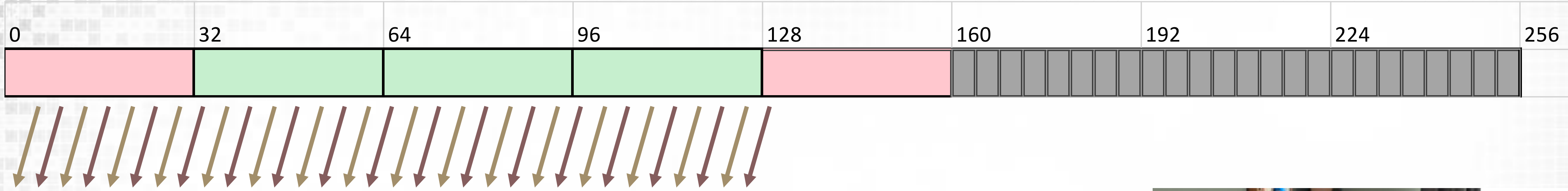
- Permuted Access
  - Within the cache line accesses can be permuted between threads
  - No performance penalty

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!

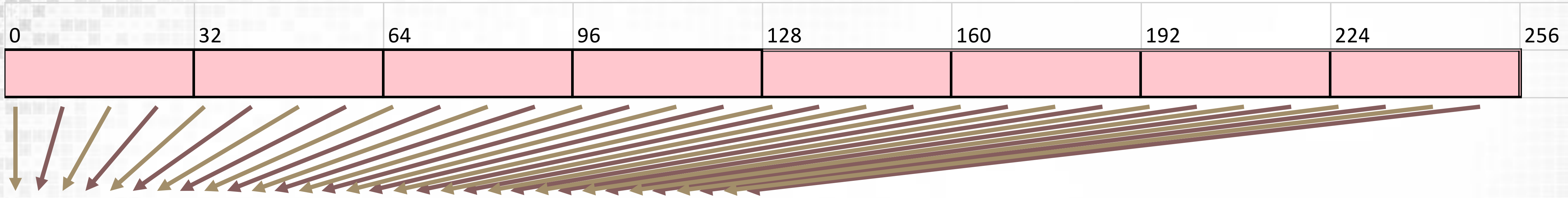
The University Of Sheffield.

L2 Strided Memory Access

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

- ❑How many cache lines transactions for warp if `STRIDE = 2`?

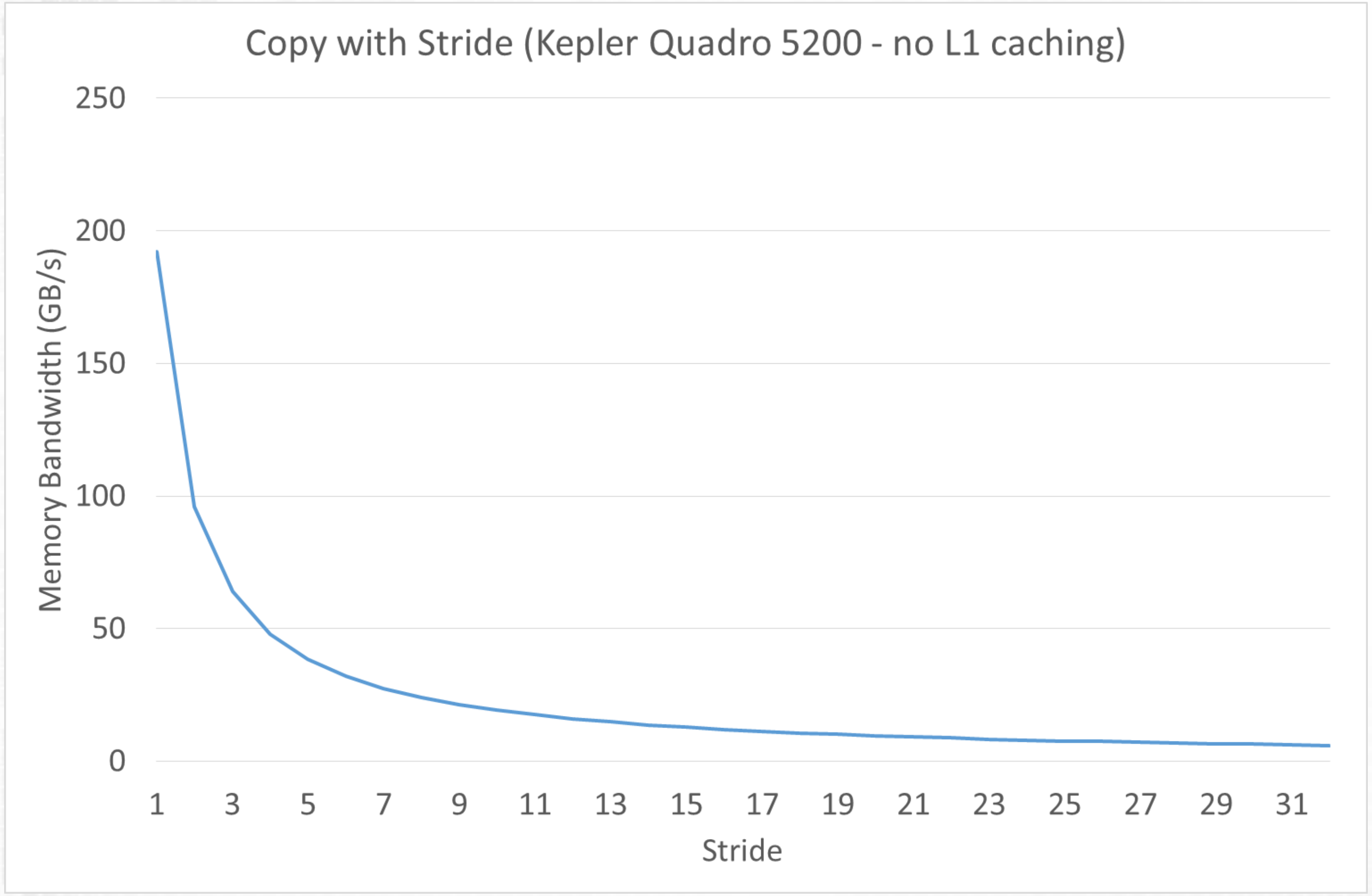
L2 Strided Memory Access



```
__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



Array of Structures vs Structures of Arrays

- ❑Array of Structures (AoS)
  - ❑Common method to store groups of data (e.g. points)

```
struct point {
  float x, y, z;
};
__device__ struct point d_points[N];

__global__ void manipulate_points()
{
  float x = d_points[blockIdx.x*blockDim.x + threadIdx.x].x;
  float y = d_points[blockIdx.x*blockDim.x + threadIdx.x].y;
  float z = d_points[blockIdx.x*blockDim.x + threadIdx.x].z;

  func(x, y, z);
}
```

Is this a good kernel?

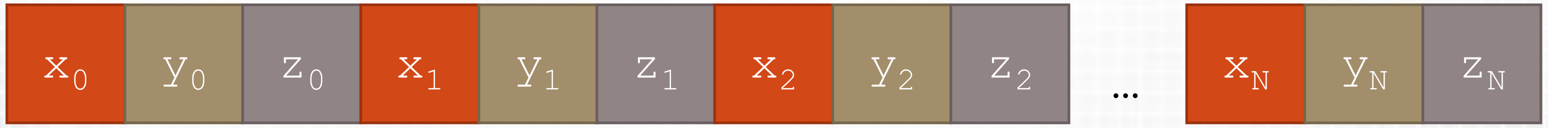
Array of Structures vs Structures of Arrays

- Array of Structures (AoS)
  - Common method to store groups of data (e.g. points)

```
struct point {
  float x, y, z;
};
__device__ struct point d_points[N];

__global__ void manipulate_points()
{
  float x = d_points[blockIdx.x*blockDim.x + threadIdx.x].x;
  float y = d_points[blockIdx.x*blockDim.x + threadIdx.x].y;
  float z = d_points[blockIdx.x*blockDim.x + threadIdx.x].z;

  func(x, y, z);
}
```


x_0 y_0 z_0 x_1 y_1 z_1 x_2 y_2 z_2 … x_N y_N z_N

Is this a good kernel? No: Stride of 3 has only 33% memory bandwidth

Array of Structures vs Structures of Arrays

- An Alternative: Structure of Arrays (SoA)

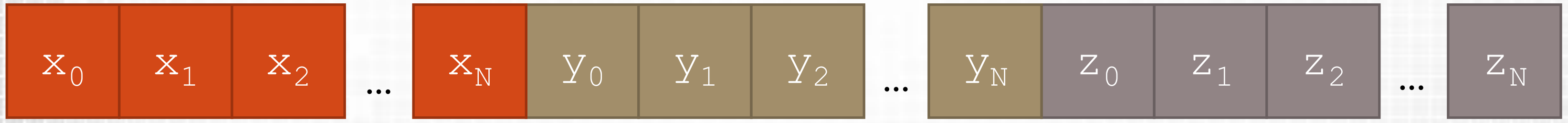
```
struct points {
  float x[N], y[N], z[N];
};

__device__ struct points d_points;

__global__ void manipulate_points()
{
  float x = d_points.x[blockIdx.x*blockDim.x + threadIdx.x];
  float y = d_points.y[blockIdx.x*blockDim.x + threadIdx.x];
  float z = d_points.z[blockIdx.x*blockDim.x + threadIdx.x];

  func(x, y, z);
}
```

100% effective memory bandwidth



Summary

- ❑Memory Coalescing
  - ❑Explain the process of memory moving via cache lines
  - ❑Analyse the performance implications of strided access patterns
  - ❑Classify use cases of memory access with respect to performance

- ❑Next Lecture: The L1 Cache (and more on Coalescing)