# Parallel Computing with GPUs: CUDA Performance

Dr Mozhgan Kabiri Chimeh

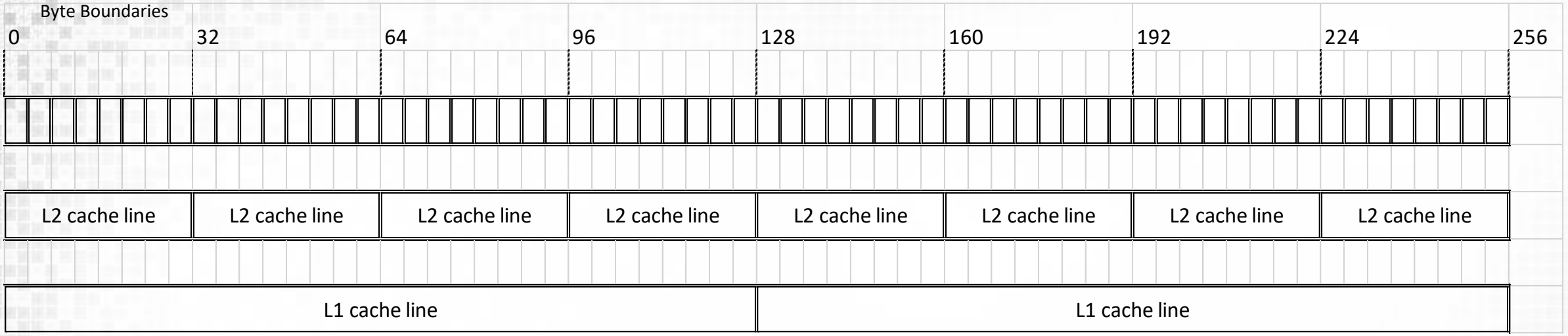
http://mkchimeh.staff.shef.ac.uk/teaching/COM4521


- Global Memory Coalescing
- Global Memory Coalescing with the L1 Cache
- Occupancy and Thread Block Dimensions

# Coalesced Global Memory Access

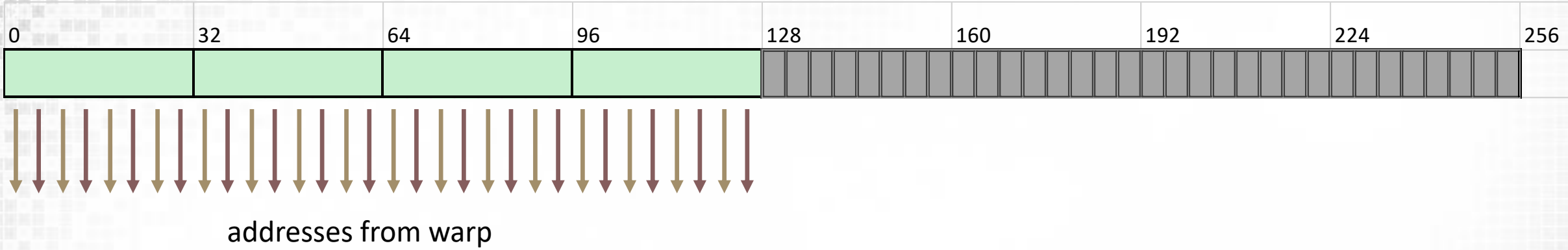
- When memory is loaded/stored from global memory to L2 (and L1) it is moved in cache lines
  - If threads within a warp access global memory in irregular patterns this can cause increased movement (transactions) of data
- Coalesced access is where sequential threads in a warp access sequentially adjacent 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

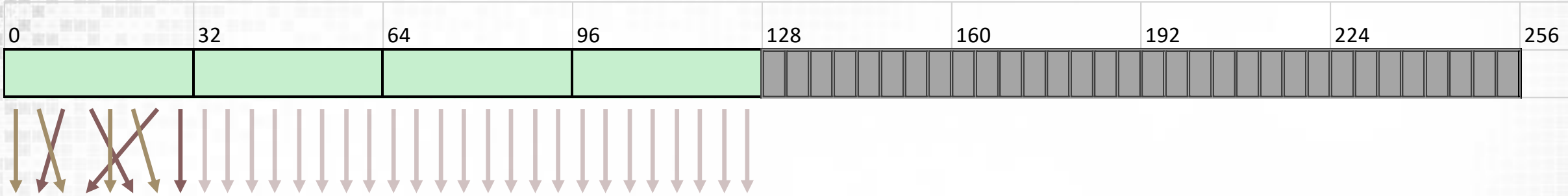
# L2 Coalesced Memory Access




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

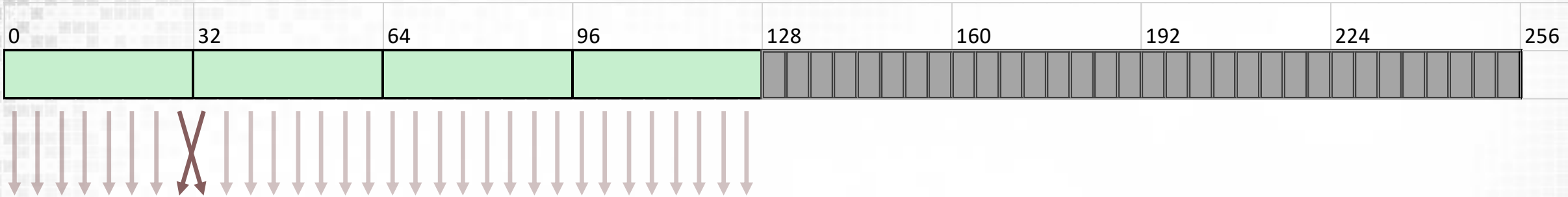
- Global memory always moves through L2
  - But not always through L1 depending on architecture
- In L2 cache line size is 32B
  - For a coalesced read/write within a warp, 4 transactions required
  - 100% memory bus speed

# L2 Permuted Memory Access



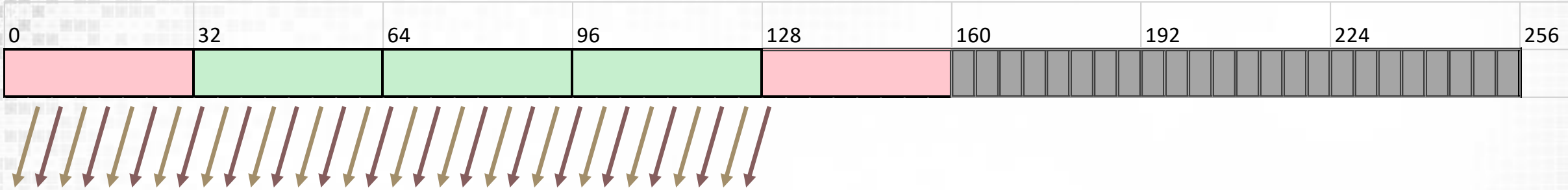

- Permuted Access
  - Within the cache line accesses can be permuted between threads
  - No performance penalty

# 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!

# L2 Strided Memory Access

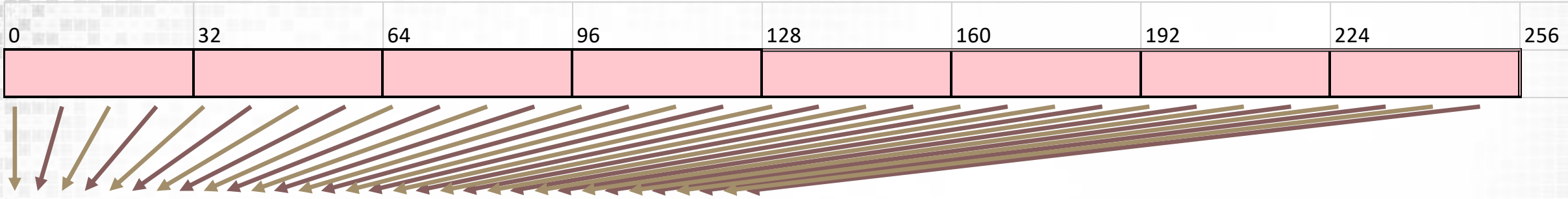
```
__global__ void copy(float *odata, float* idata)
    int xid = (blockIdx.x * blockDim.x + threadIdx.x)* STRIDE;
    odata[xid] = idata[xid];
}
```

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

The University Of Sheffield.

NVIDIA GPU RESEARCH CENTER

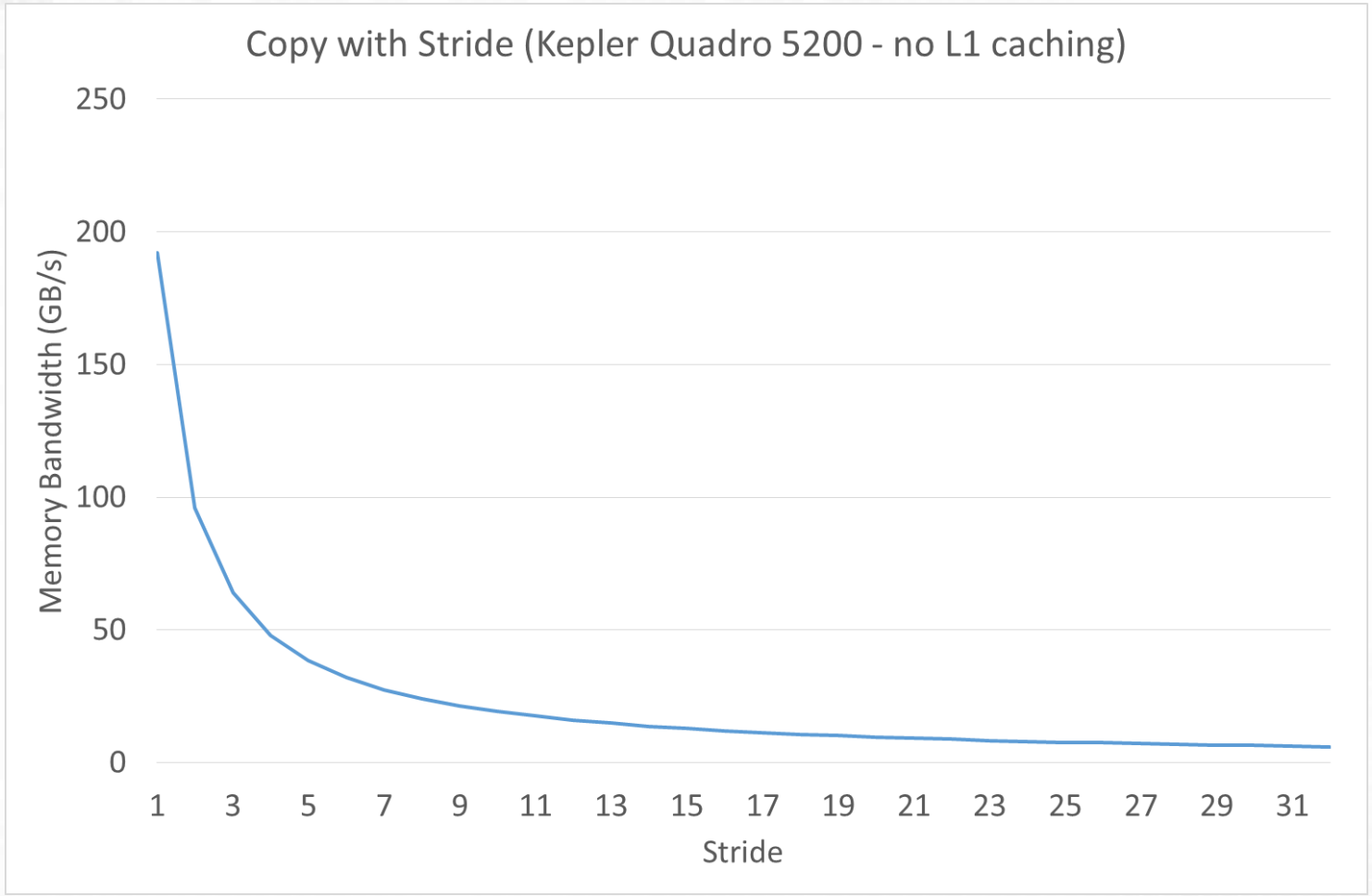
# L2 Strided Memory Access




```
__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

# 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].x;
  float z = d_points[blockIdx.x*blockDim.x + threadIdx.x].x;

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

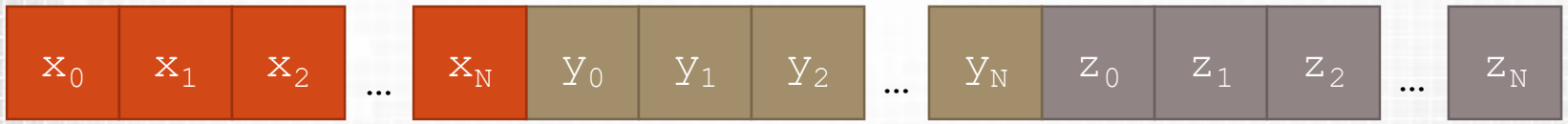

Is this a good kernel? No: Stride of 3 has only 33% memory bandwidth

# 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

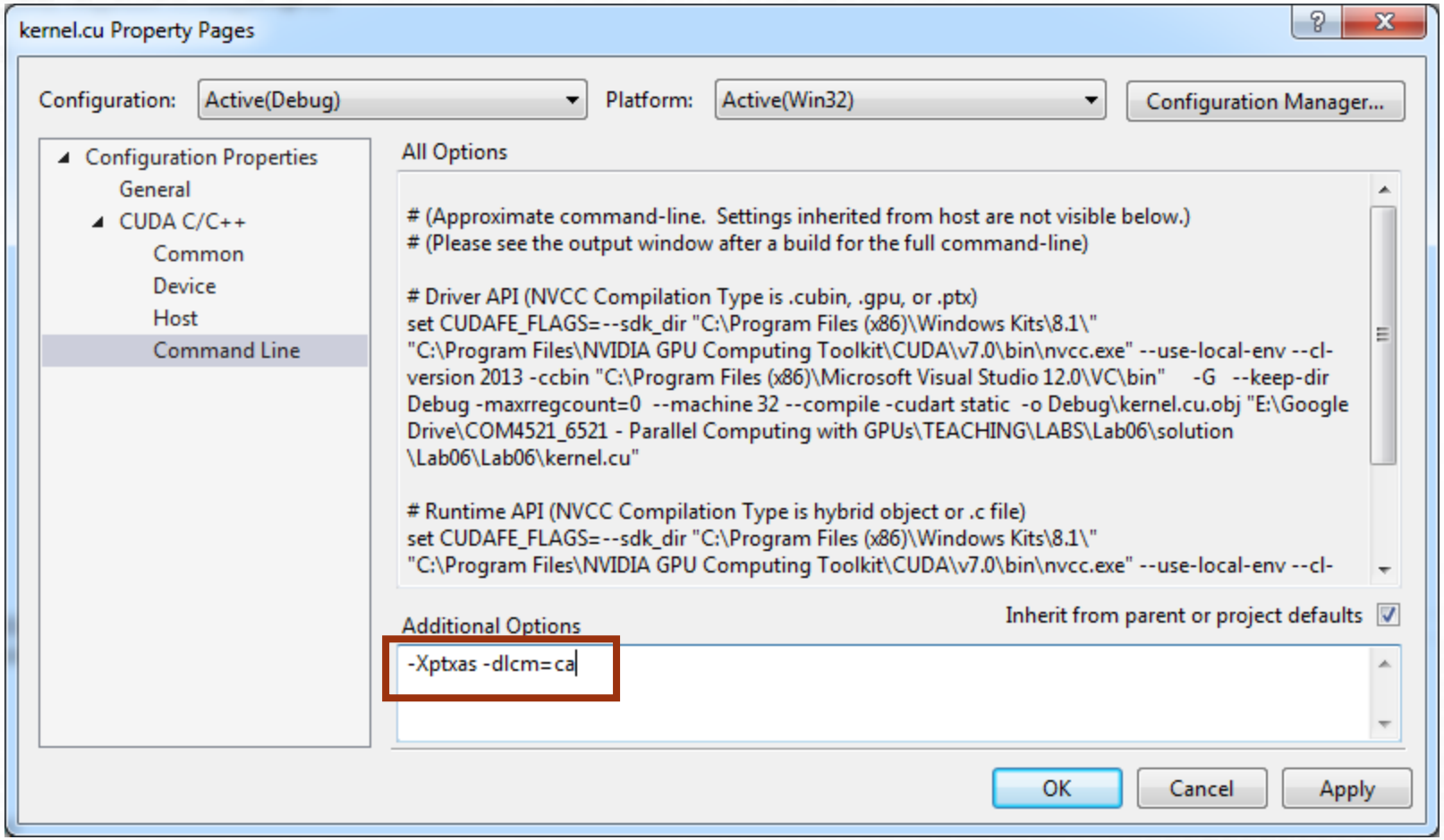


- ❑Global Memory Coalescing
- ❑Global Memory Coalescing with the L1 Cache
- ❑Occupancy and Thread Block Dimensions

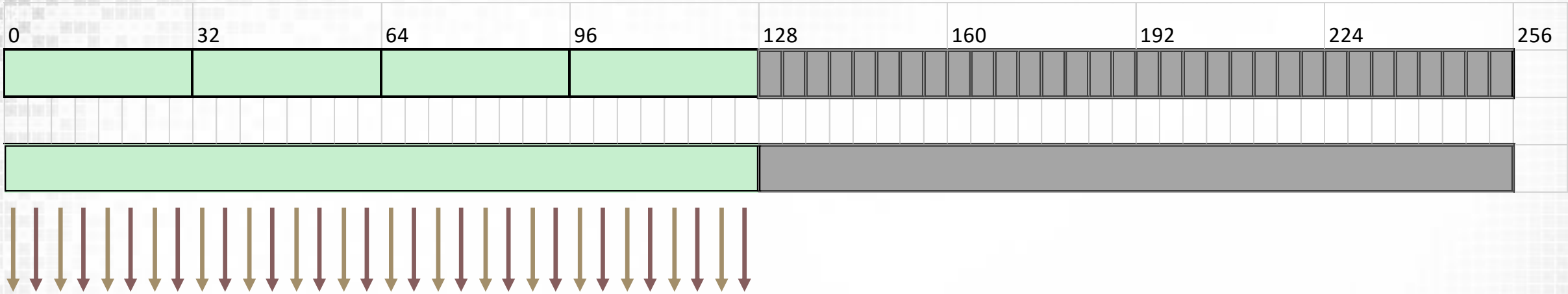
# L1 Cache

- What affect does this have on performance of memory movement?
  - Can be good in certain circumstances
    - Coalesced access with adjacent warps reading same data
  - Can also be bad
    - Un-coalesced access performance is worse
    - Increases over-fetch
- Does my card support global L1 Caching?
  - Check globalL1CacheSupported and localL1CacheSupported CUDA device properties
    - Maxwell 5.2 reports globalL1CacheSupported false when in fact true!
- Enabling L1 caching of global loads
  - Pass the `-Xptxas -dlcm=ca` flag to nvcc at compile time
    - -dlcm=cg can be used to disable L1 on devices which use it by default

# Enabling L1 Cache in Visual Studio



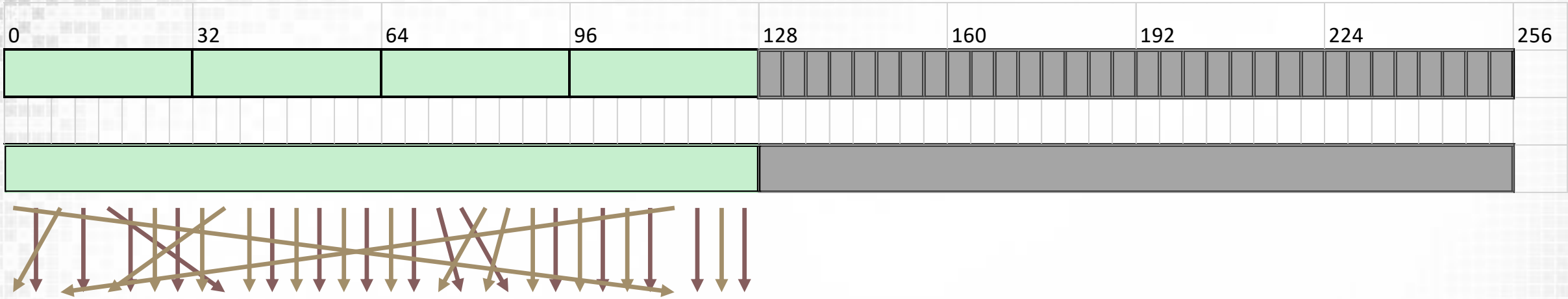
# L1 Coalesced Memory Access




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

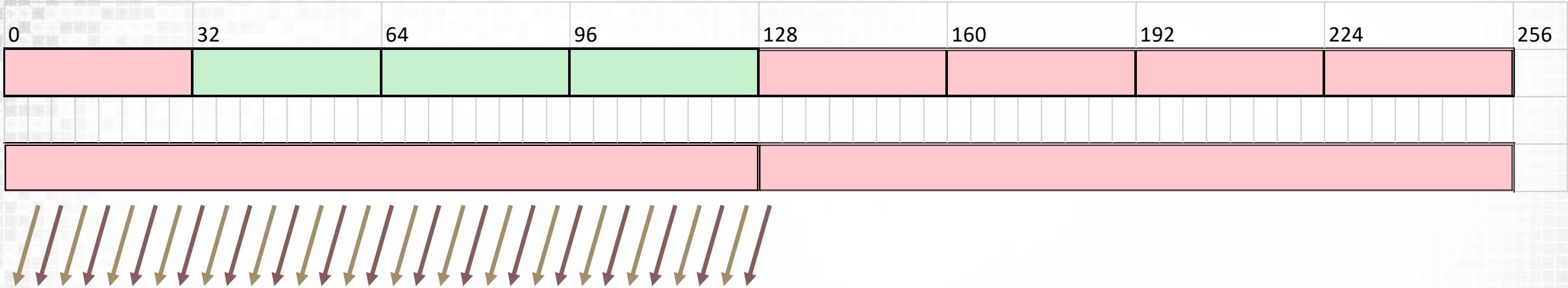
- All addresses fall in one 128B cache line
  - Single transaction
- 100% bus utilisation

# L1 Permuted Memory Access




- Any thread within the warp can permute access
- Same as L2

# L1 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.
  - 2 transactions of 256B of which 128B is required: 50% utilisation

- For strided and random access performance is much worse with L1

- ❑Global Memory Coalescing
- ❑Global Memory Coalescing with the L1 Cache
- ❑Occupancy and Thread Block Dimensions

# Occupancy

- Occupancy is the ratio of active warps on an SMP to the maximum number of active warps supported by the SMP
  - Occupancy = Active Warps / Maximum Active Warps

- Why does it vary?
  - Resources are allocated at thread block level and resources are finite
  - Multiple thread blocks can be assigned to a single Streaming Multi Processor
  - Your occupancy might be limited by either
    1. Number of registers
    2. Shared memory usage
    3. Limitations on physical block size

# Why is occupancy important

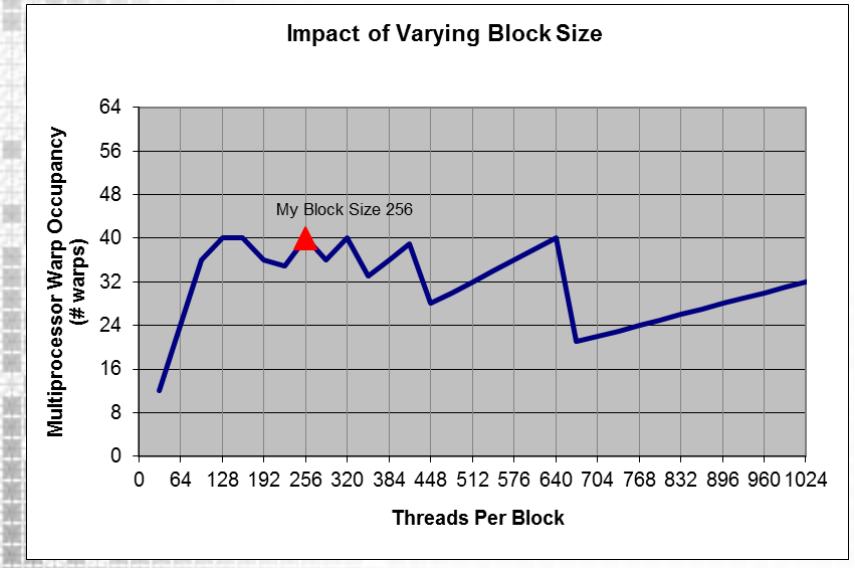
- Implications of Increasing Occupancy
  - Memory bound code
    - Higher occupancy will hide memory latency
    - If bandwidth is less than peak then increasing active warps might improve this
  - Compute bound code
    - Will not improve performance
  - 100% occupancy not required for maximum performance
    - Instruction throughput might be optimal
    - Memory bandwidth might be fully saturated

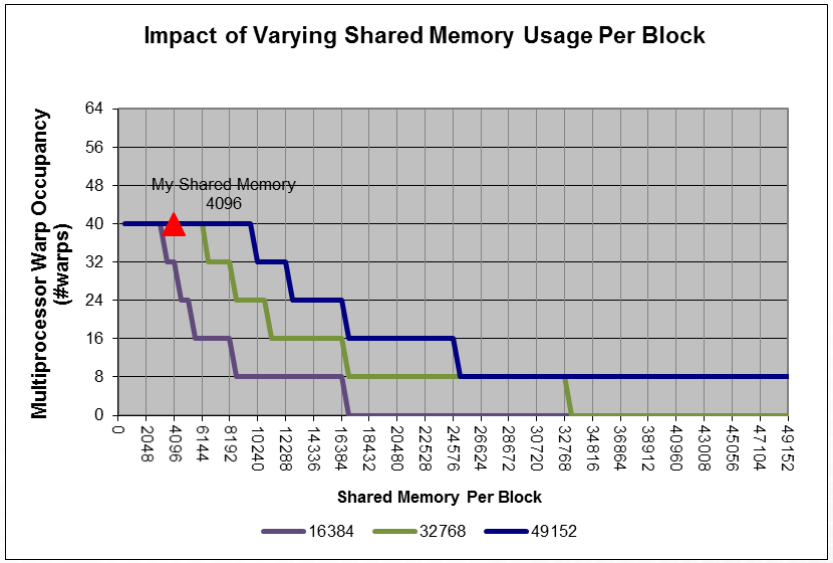
# Occupancy and Thread Block Size

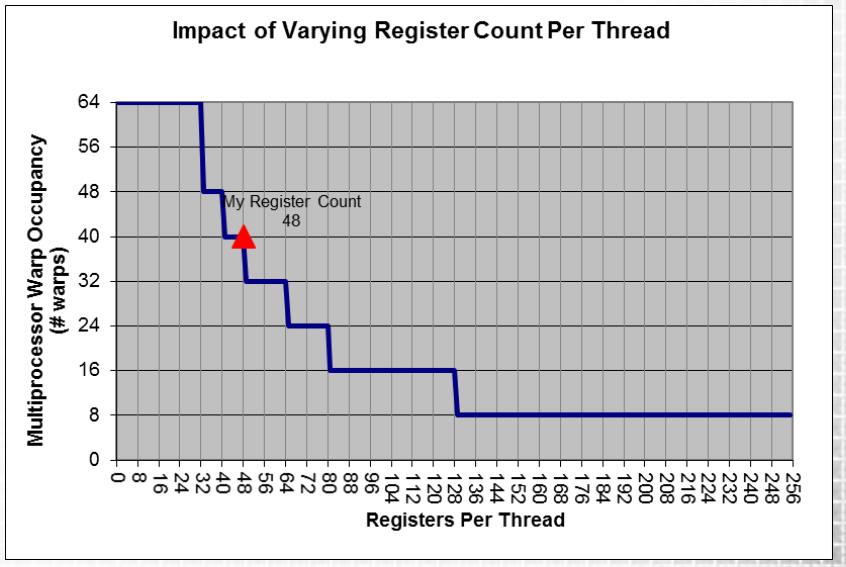
- Thread Block Limitations
  - Always a factor of 32 (warp size)
- Changing the thread block size will change occupancy
  - If thread block size is too small
    - There is a fixed limit on the number of active thread blocks per SM (16 in Kepler, 32 in Maxwell)
  - If thread block is too big
    - Not enough resources for another block
    - Block is stalled until enough resource is available
- The relationship between thread block size and occupancy is non linear
  - Complex interplay between resources

# Occupancy Calculator

- The CUDA Occupancy calculator is available for download
  - http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls
  - By giving a Compute Capability, Threads per block usage registers per thread and shared memory per block occupancy can be predicted.
  - It will also inform you what factor is limiting occupancy (registers, SM, block size)




The University Of Sheffield.

NVIDIA GPU RESEARCH CENTER

# Occupancy Calculator

- How do I know what my SM usage per block is?
  - You either statically declared it or dynamically requested it as a kernel argument
- How do I know what my register usage is?
  - CUDA build rule Device Properties-> Verbose PTX Output = Yes
    - i.e. nvcc –ptxas-options=-v

```
1>------ Build started: Project: Lab06, Configuration: Debug Win32 ------
1>  Compiling CUDA source file kernel.cu...
1>
1>  E:\Lab06>"nvcc.exe" -gencode=arch=compute_35,code=\"sm_35,compute_35\" --use-local-env --cl-
version 2013 -ccbin "C:\MSVS12.0\VC\bin"  -I"C:\CUDA\v7.0\include" -I"C:\CUDA\v7.0\include"  -G   -
-keep-dir Debug -maxrregcount=0 --ptxas-options=-v --machine 32 --compile -cudart static -Xptxas -
dlcm=ca  -g   -D__CUDACC__ -DWIN32 -D_DEBUG -D_CONSOLE -D_MBCS -Xcompiler "/EHsc /W3 /nologo /Od
/Zi /RTC1 /MDd  " -o Debug\kernel.cu.obj "E:\Lab06\kernel.cu"
1>  ptxas info    : 0 bytes gmem
1>  ptxas info    : Function properties for cudaGetDevice
1>      8 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
1>  ptxas info    : Function properties for cudaFuncGetAttributes
1>      8 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
1>  ptxas info    : Function properties for cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags
1>      24 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
1>  ptxas info    : Function properties for cudaDeviceGetAttribute
1>      16 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
1>  ptxas info    : Compiling entry function '_Z9addKernelPiS_S_' for 'sm_35'
1>  ptxas info    : Function properties for _Z9addKernelPiS_S_
1>      0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
1>  ptxas info    : Used 6 registers, 332 bytes cmem[0]
1>  ptxas info    : Function properties for cudaMalloc
1>      8 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
1>  ptxas info    : Function properties for cudaOccupancyMaxActiveBlocksPerMultiprocessor
1>      16 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
1>  kernel.cu
1>  Lab06.vcxproj -> E:\Lab06.exe
1>  copy "C:\CUDA\v7.0\bin\cudart*.dll" "E:\Lab06\Debug\"
1>  C:\CUDA\v7.0\bin\cudart32_70.dll
1>  C:\CUDA\v7.0\bin\cudart64_70.dll
1>           2 file(s) copied.
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

# Intelligent Launching

- Since CUDA 6.5 It is possible to launch block sizes to maximise occupancy (using the Occupancy API)
  - This does not guarantee good performance!
  - However: Usually a good compromise
- `cudaOccupancyMaxPotentialBlockSize:` will find best block size and minimum grid size
  - Actual grid size must be calculated

```
int blockSize;
int minGridSize;
int gridSize;

                                                    Static SM size
cudaOccupancyMaxPotentialBlockSize(&minGridSize, &blockSize, MyKernel, 0, 0);
gridSize = (arrayCount + blockSize - 1) / blockSize; //round up
MyKernel <<< gridSize, blockSize >>>(d_data, arrayCount);
```

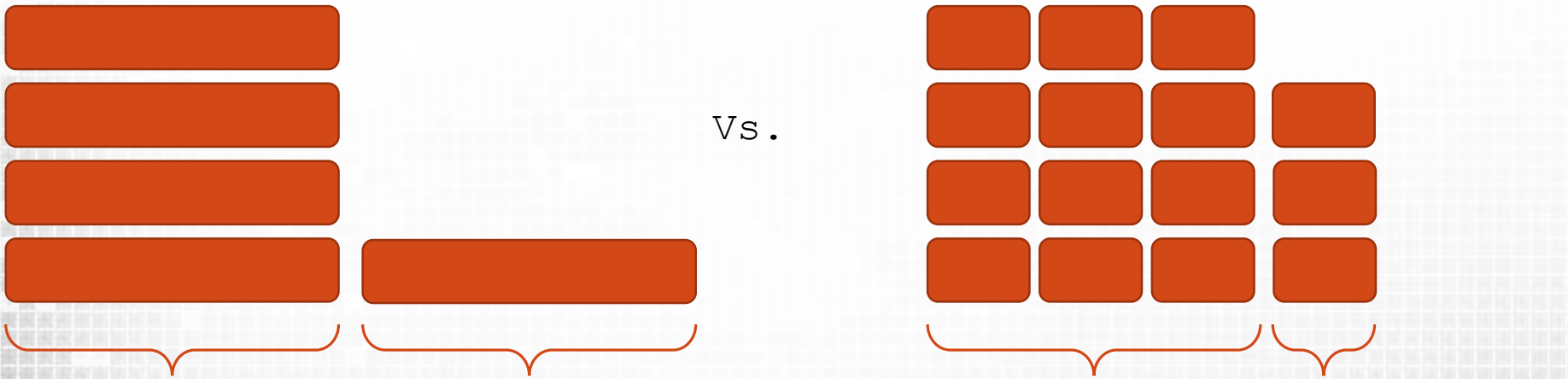
# Occupancy SDK for Shared Memory

- What if SM use varies depending on block size?

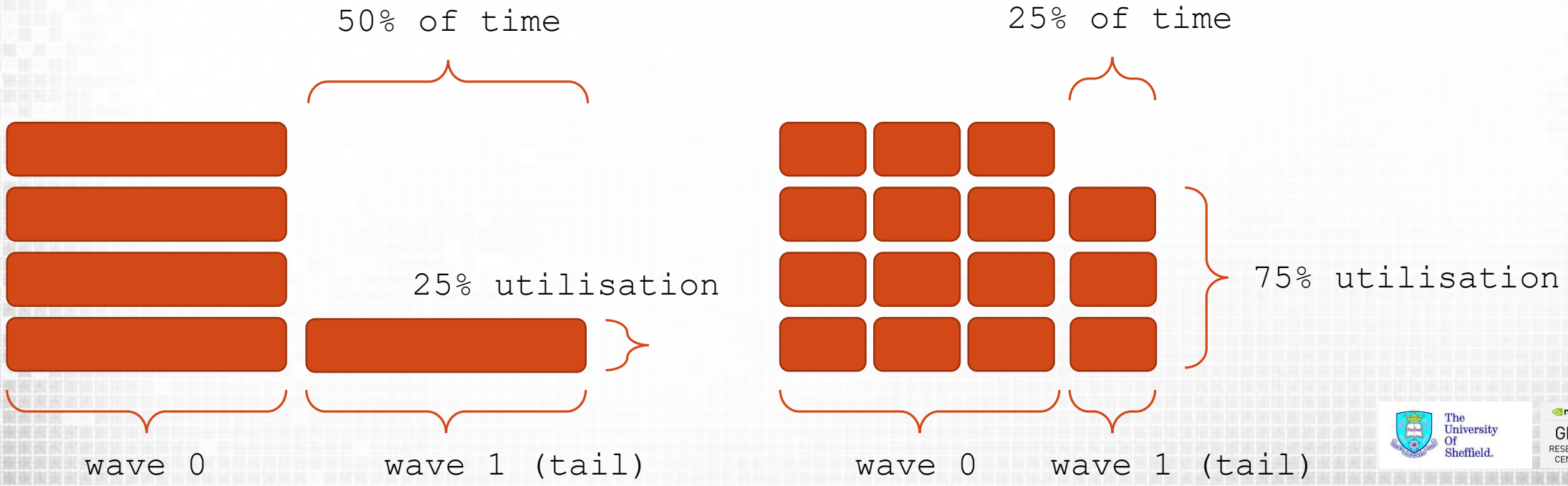
```
int SMFunc(int blockSize){
  return blockSize*sizeof(int);
}



void launchMyKernel(int *d_data, int arrayCount)
{
  int blockSize;
  int minGridSize;
  int gridSize;

  cudaOccupancyMaxPotentialBlockSizeVariableSMem(&minGridSize, &blockSize, MyKernel, SMFunc, 0);
  gridSize = (N + blockSize - 1) / blockSize;
  MyKernel <<< gridSize, blockSize, SMFunc(gridSize) >>>(d_data, arrayCount);
}
```

# Other considerations for block sizes

- Waves and Tails
  - A Wave is a set of thread blocks that run concurrently on the device
    - Grid launch may have multiple waves
  - A Tail is the partial thread block left as a result of dividing problem size by thread block dimensions
- Performance Implications
  - Larger thread blocks sizes may result in inefficient execution



# Other considerations for block sizes



# Summary

- For best memory bandwidth coalesced access is very important
- Care should be taken to avoid unnecessary offsets or strides which degrade memory performance
- Structures of Arrays should be used rather than Arrays of Structures
- L1 cache can be good for improving performance but will reduce performance when strided or random access patterns are used
- Occupancy is a measure which can be used for improving the performance of memory bound code
- Large thread blocks might be good for occupancy but introduce large tails (benchmarking to balance tradeoff is therefore crucial!)

# Acknowledgements and Further Reading

- Cache line sizes
- GPU Performance Analysis
  - http://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC2012-GPU-Performance-Analysis.pdf
- Waves and Tails (http://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC2012-GPU-Performance-Analysis.pdf)
- How to enable use of L1 cache
  - http://acceleware.com/blog/opt-in-L1-caching-global-loads
- Better Performance at Lower Occupancy
  - http://nvidia.fullviewmedia.com/gtc2010/0922-a5-2238.html