OpenCL Optimization

Optimization Strategies

- Maximize parallel execution
  - Exposing data parallelism in algorithms
  - Choosing execution configuration
  - Overlap memory transfer with computation
- Maximize memory bandwidth
  - Keep the hardware busy
- Maximize instruction throughput
  - Get the job done with as few clock cycles as possible

Memory Optimization

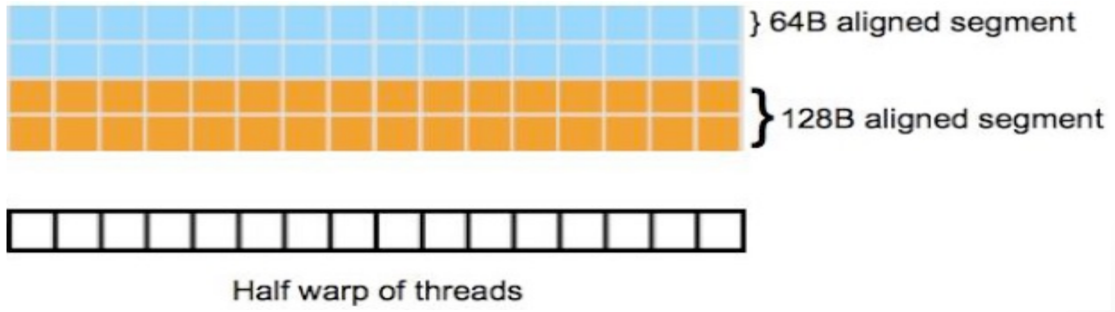
- Minimize host-device data transfer
- Coalesce global memory access
- Use local memory as a cache

Minimizing host-device data transfer

- Host device data transfer has much lower bandwidth than global memory access.
  - 8 GB/s (PCIe,x16 Gen2) vs 141 GB/s (GTX 280)
  - 32 GB/s (PCIe,x16 Gen3) vs 320 GB/s (GTX 1080)
- Minimize transfer
  - Intermediate data can be allocated, operated, de-allocated directly on the GPU
  - Sometimes it’s even better to recompute on GPU, or call kernels that do not have performance gains
- Group transfers
  - One large transfer much better than many small ones

Coalescing

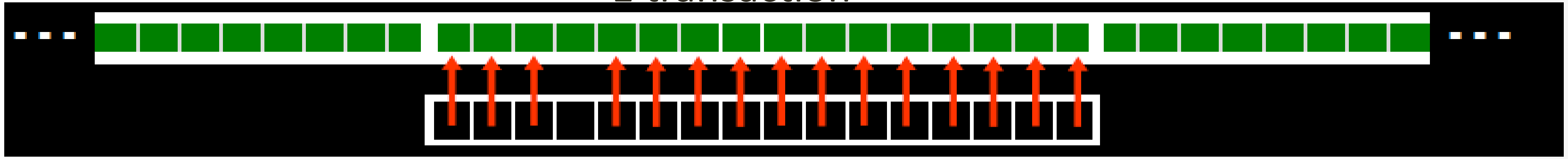
- Global memory latency: 400-600 cycles.
  - The single most important performance consideration!
- Global memory access by threads of a half warp can be coalesced to one transaction for word of size 8-bit, 16-bit, 32-bit, 64-bit or two transactions for 128-bit.
- Global memory can be viewed as composing aligned segments of 16 and 32 words.



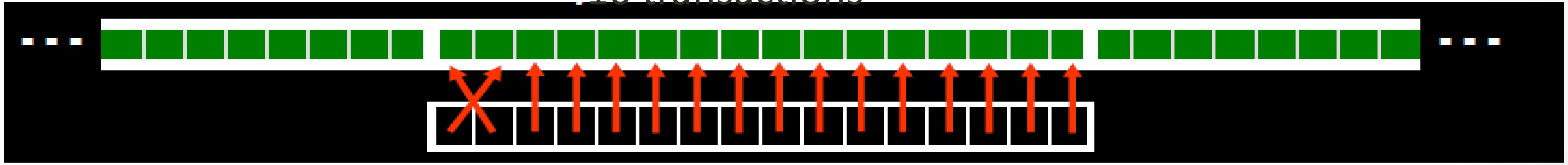
Coalescing Compute Capability 1.0 and 1.1

- **K-th thread in a half warp must access the k-th word in a segment; however, not all threads need to participate**

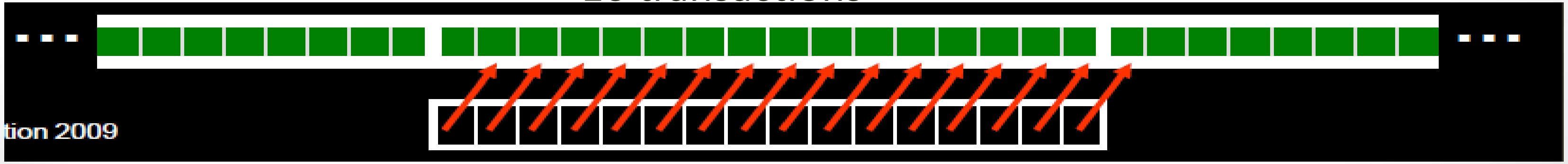
1 transaction



16 transactions

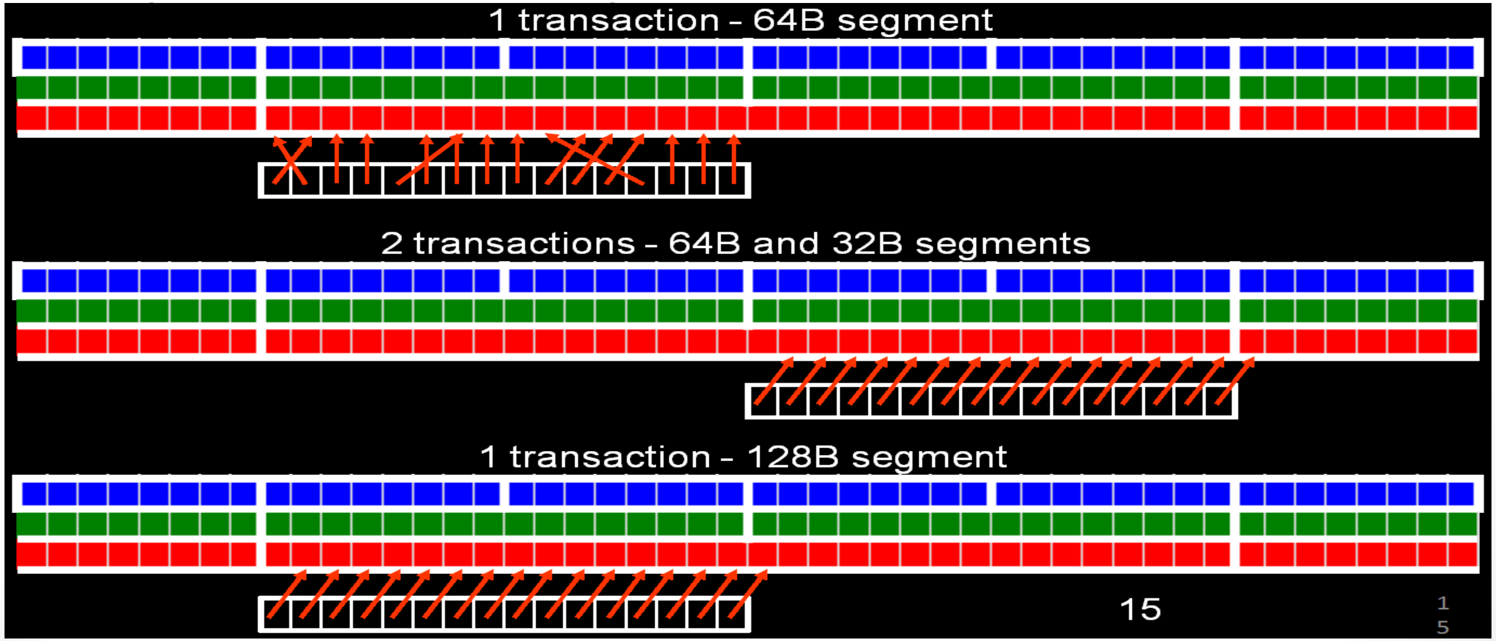


16 transactions



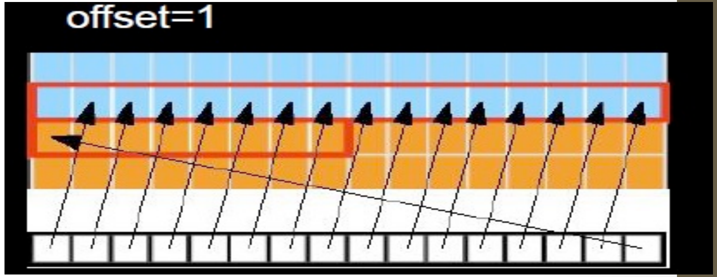
Coalescing in Compute Capability 1.2 and 1.3

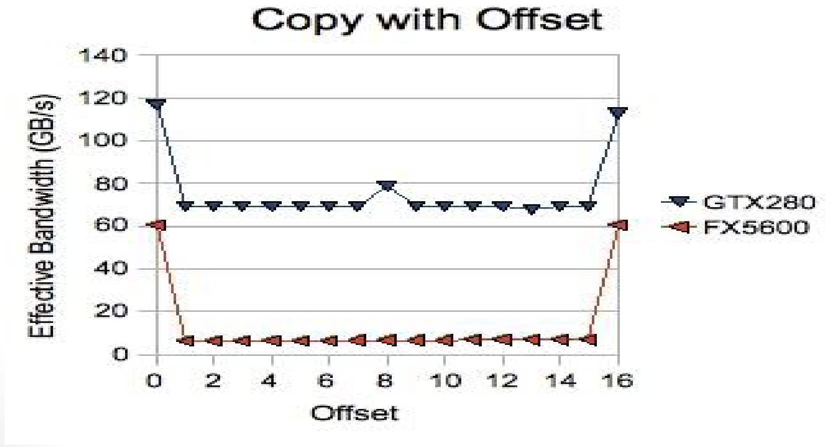
- **Coalescing for any pattern of access that fits into a segment size**


1 transaction - 64B segment
2 transactions - 64B and 32B segments
1 transaction - 128B segment
15
1
5

Example of Misaligned Accesses

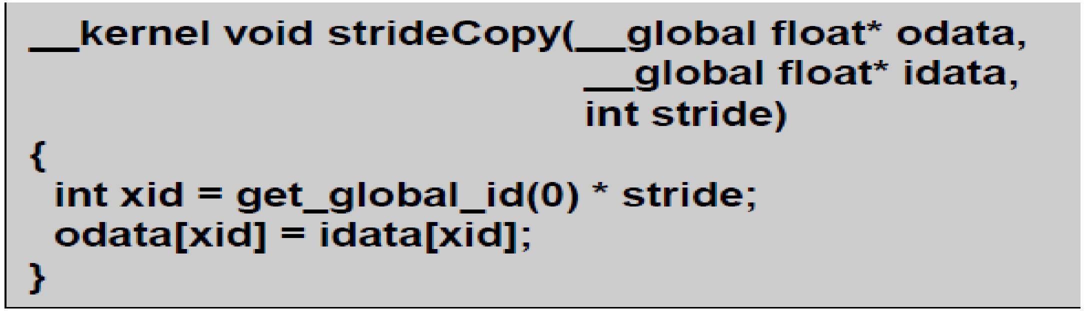
```
__kernel void offsetCopy(__global float *odata,
                         __global float* idata,
                         int offset)
{
  int xid = get_global_id(0) + offset;
  odata[xid] = idata[xid];
}
```



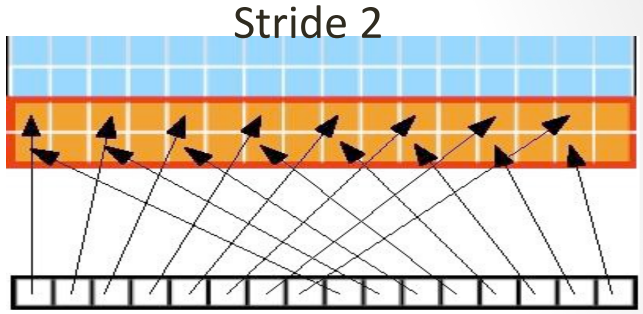


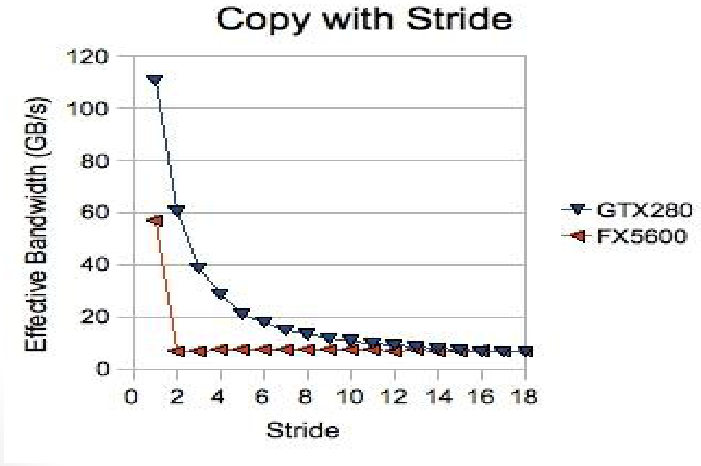
GTX280 (compute capability 1.3) drops by a factor of 1.7 while FX 5600 (compute capability 1.0) drops by a factor of 8.

Example of Strided Accesses



```
__kernel void strideCopy(__global float* odata,
                         __global float* idata,
                         int stride)
{
  int xid = get_global_id(0) * stride;
  odata[xid] = idata[xid];
}
```





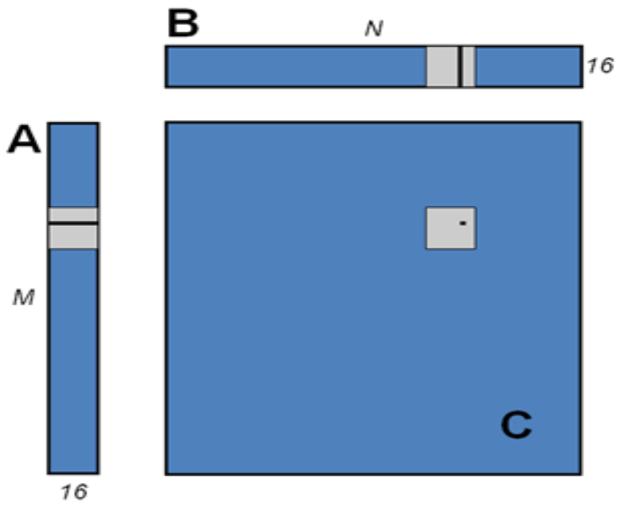
Large strides often arise in applications. However, strides can be avoided using local memory.

Local Memory

- **Latency ~100x smaller than global memory**
- **Cache data to reduce global memory access**
- **Use local memory to avoid non-coalesced global memory access**
- **Threads can cooperate through local memory**

Caching Example 1: Matrix Multiplication

C=AxB

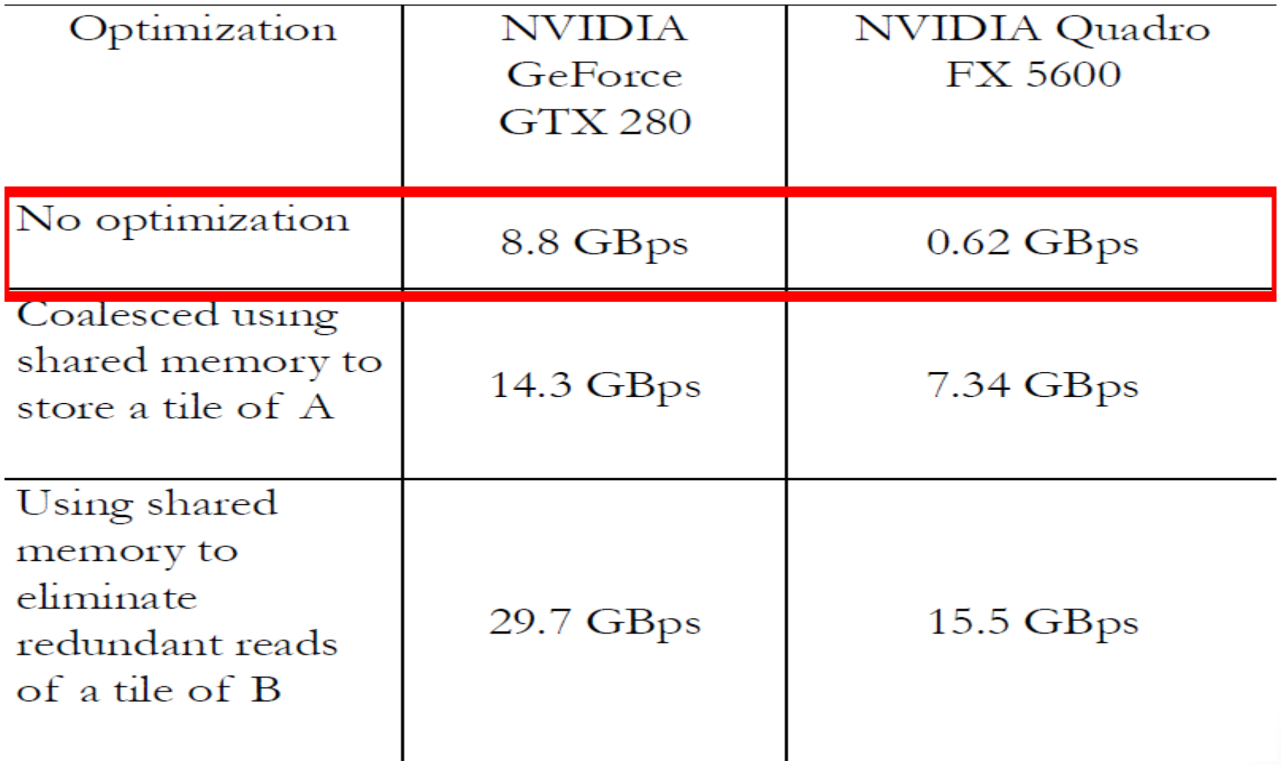


Every thread corresponds to one entry in C.

Uncached version

```
__kernel void simpleMultiply(__global float* a,
                             __global float* b,
                             __global float* c,
                             int N)
{
  int row = get_global_id(1);
  int col  = get_global_id(0);
  float sum = 0.0f;
  for (int i = 0; i < TILE_DIM; i++) {
    sum += a[row*TILE_DIM+i] * b[i*N+col];
  }
  c[row*N+col] = sum;
}
```

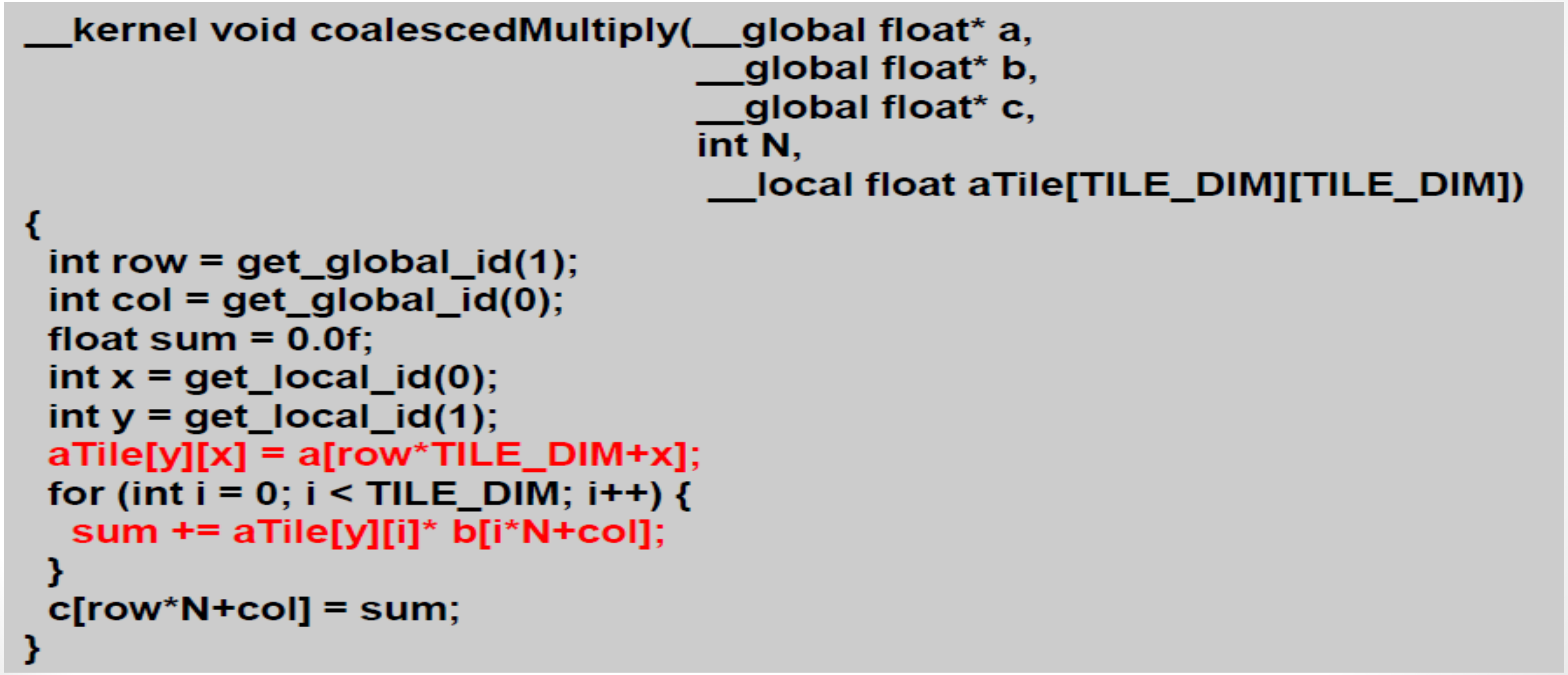
Matrix Multiplication …



Optimization	NVIDIA GeForce GTX 280	NVIDIA Quadro FX 5600
No optimization	8.8 GBps	0.62 GBps
Coalesced using shared memory to store a tile of A	14.3 GBps	7.34 GBps
Using shared memory to eliminate redundant reads of a tile of B	29.7 GBps	15.5 GBps

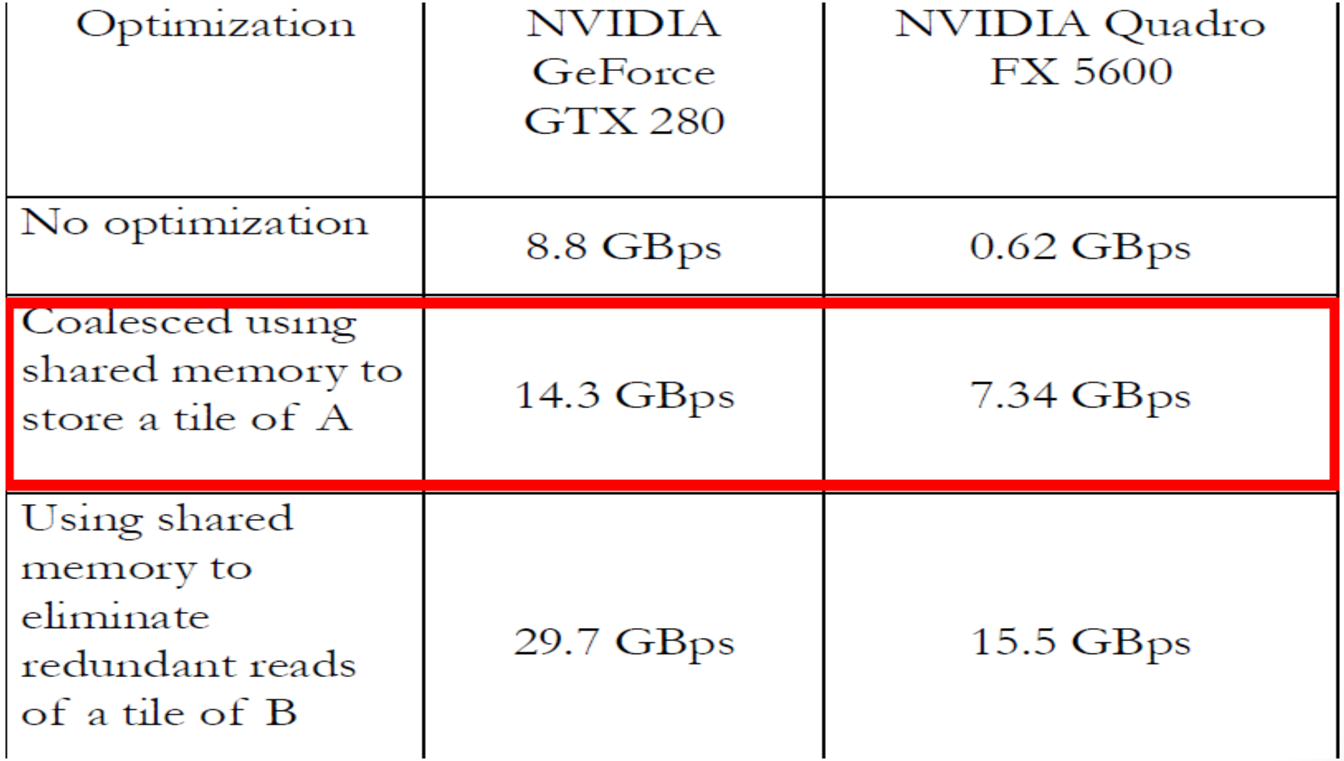
Matrix Multiplication

- Cached and coalesced



```
__kernel void coalescedMultiply(__global float* a,
                                __global float* b,
                                __global float* c,
                                int N,
                                 __local float aTile[TILE_DIM][TILE_DIM])
{
  int row = get_global_id(1);
  int col = get_global_id(0);
  float sum = 0.0f;
  int x = get_local_id(0);
  int y = get_local_id(1);
  aTile[y][x] = a[row*TILE_DIM+x];
  for (int i = 0; i < TILE_DIM; i++) {
   sum += aTile[y][i]* b[i*N+col];
  }
  c[row*N+col] = sum;
}
```

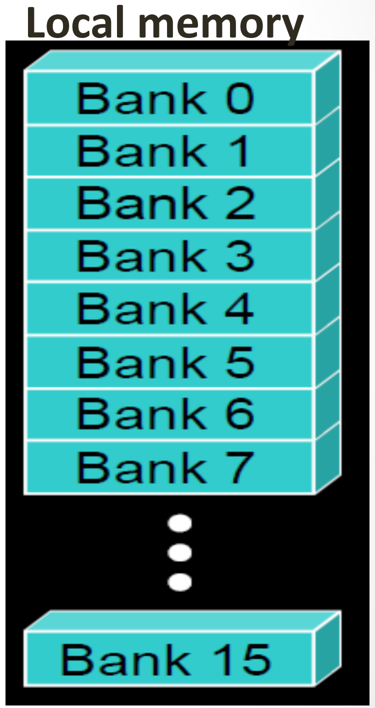
Matrix Multiplication



Optimization	NVIDIA GeForce GTX 280	NVIDIA Quadro FX 5600
No optimization	8.8 GBps	0.62 GBps
Coalesced using shared memory to store a tile of A	14.3 GBps	7.34 GBps
Using shared memory to eliminate redundant reads of a tile of B	29.7 GBps	15.5 GBps

Bank Conflicts

- A 2nd order effect compared to global memory coalescing
- Local memory is divide into banks.
- Successive 32-bit words assigned to successive banks
- Number of banks = 16 for CC 1.x
- R/W different banks can be performed simultaneously.
- Bank conflict: two R/W fall in the same bank, the access will be serialized.
- Thus, accessing should be designed to avoid bank conflict



Work-group Heuristics

- **# of work-groups > # of SM (Compute Units)**
- **Each SM has at least one work-group to execute**
- **# of work-groups / # of SM > 2**
- **Multiple work-groups can run concurrently on an SM**
- **Work on another work-group if one work-group is waiting on barrier**
- **# of work-groups / # of SM > 100 to scale well to future device**

Work-item Heuristics

- The number of work-items per work-group should be a multiple of 32 (warp size for nvidia gpus or preffered workgroup multiple)
- Want as many warps running as possible to hide latencies
- Minimum: 64
- Larger, e.g. 256 may be better
- Depends on the problem, do experiments!

Occupancy

- Hide latency: thread instructions are executed sequentially. So executing other warps when one warp is paused is the only way to hide latencies and keep the hardware busy

- Occupancy: ratio of active warps per SM to the maximum number of allowed warps
  - 32 in GT 200, 24 in GeForce 8 and 9-series.

Global Memory Latency Hiding

- **Enough warps can hide the latency of global memory access**
- **We need 400/4 = 100 arithmetic instructions to hide the latency. For example, assume the code has 8 arithmetic instructions (4 cycle) for every one global memory access (~400 cycles). Thus 100/8~13 warps would be enough. This corresponds to 54% occupancy.**

Register Dependency Latency Hiding

- **If an instruction uses a result stored in a register written by an instruction before it, this is ~ 24 cycles latency**
- **So, we need 24/4=6 warps to hide register dependency latency. This corresponds to 25% occupancy**

Occupancy Considerations

- Increase occupancy to achieve latency hiding
- After some point (e.g. 50%), further increase in occupancy won’t lead to performance increase
- Occupancy is limited by resource usage:
  - Registers – eg. 8K
  - Local memory –eg. 32K
  - Scheduling hardware

Resource Limitation on Occupancy

- Work-groups on a SM partition registers and local memory
- If every thread uses 10 registers and every work-group has 256 work-items, then 3 work-groups use 256*10*3 < 8192. A 100% occupancy can be achieved.
- However, if every thread uses 11 registers, since 256*11*3 > 8192, only 2 work-groups are allowed. So occupancy is reduced to 66%!
- But, if work-group has 128 work-items, since 128*11*5 < 8192, occupancy can be 83%.

Other Resource Limitations on Occupancy

- **Maximum number of warps.**
- **Maximum number of work-groups per SM: 8**
- **So occupancy calculation in realistic case is complicated**

Instruction Throughput

- Throughput: # of instructions per cycle
- In SIMT architecture, if T is the number of operations per clock cycle

  SM Throughtput = T/WarpSize
- Maximizing throughput: using smaller number of cycles to get the job done

Arithmetic Instruction Throughput

- Int, and float add, shift, min, max, and float mul, mad: T = 8
- Int divide and modulo are expensive
- Avoid automatic conversion of double to float
  - Adding “f” to floating literals (e.g. 1.0f) because the default is double

Memory Instructions

- **Use local memory to reduce global memory access**
- **Increase algorithm's arithmetic intensity (the ratio of arithmetic to global memory access instructions). The higher of this ratio, the fewer of warps are required to hide global memory latency.**

Control Flow

- **If branching happens within a warp, different execution paths must be serialized, increasing the total number of instructions.**
- **No penalty if different warps diverge**
- **No divergence if controlling condition depends only on local_id/warp_size**

QUESTIONS??