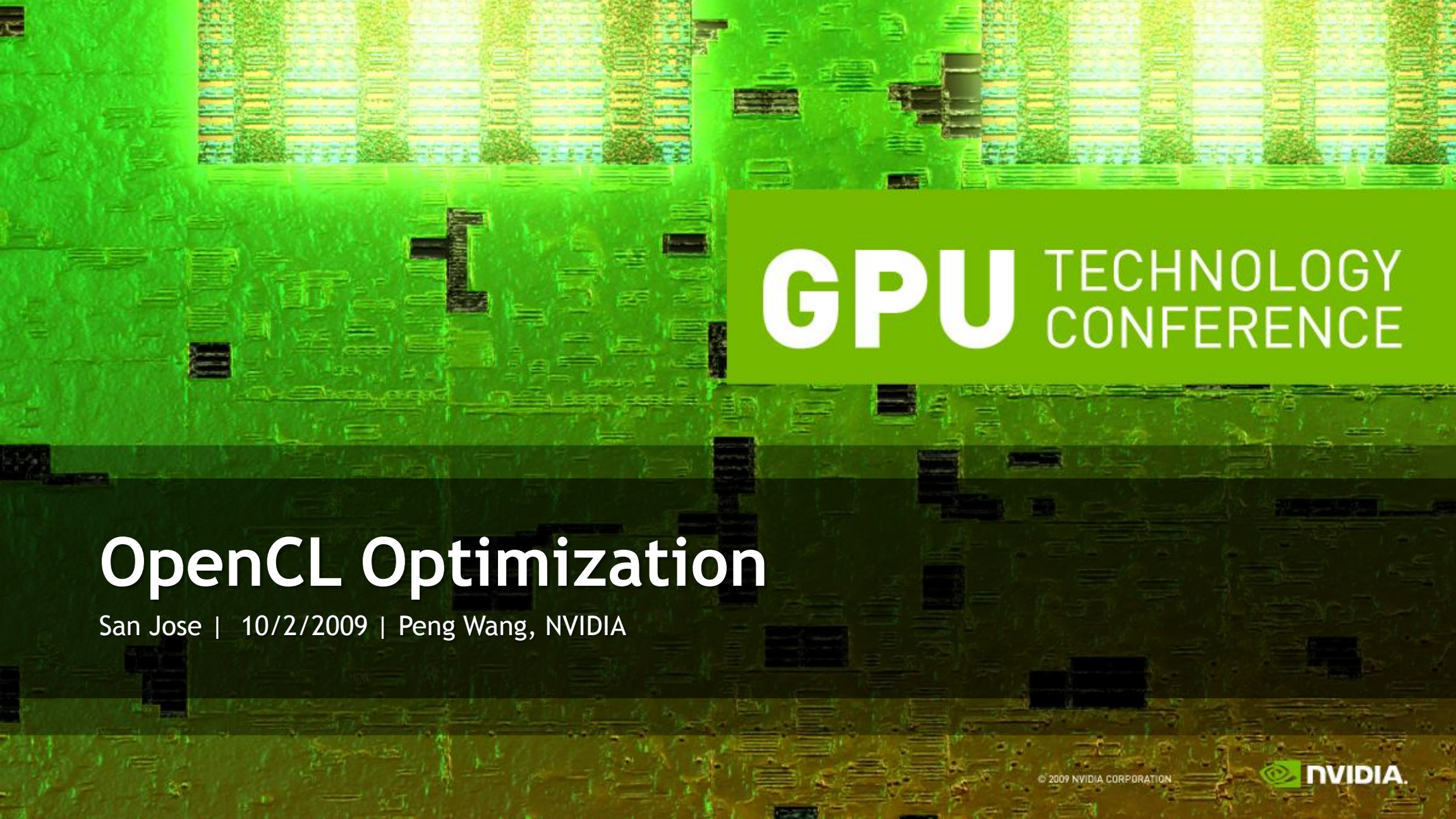


OpenCL Optimization

San Jose | 10/2/2009 | Peng Wang, NVIDIA

NVIDIA.

Outline

- Overview
- The CUDA architecture
- Memory optimization
- Execution configuration optimization
- Instruction optimization
- Summary

Overall Optimization Strategies

- Maximize parallel execution
  - Exposing data parallelism in algorithms
  - Choosing execution configuration
  - Overlap memory transfer with computation
- Maximize memory bandwidth
  - Avoid starving the GPU
- Maximize instruction throughput
  - Get the job done with as few clock cycles as possible
- Profiling your code before doing optimization
  - NVIDIA OpenCL visual profiler

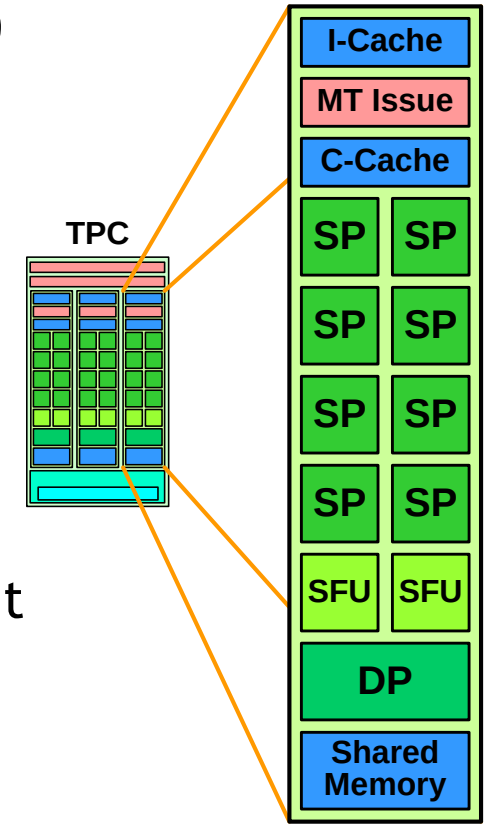
We will talk about how to do those in NVIDIA GPUs.
Very similar to CUDA C.

Outline

- Overview
- The CUDA architecture
- Memory optimization
- Execution configuration optimization
- Instruction optimization
- Summary

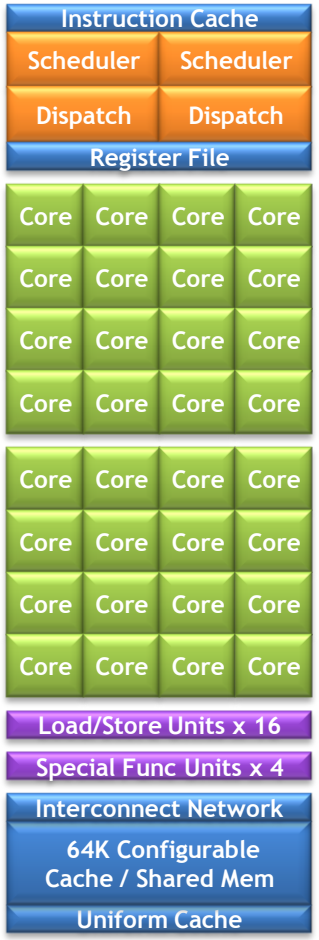
Tesla Compute Architecture (GT200)

- GPU contains 30 Streaming Multiprocessors (SM)
- Each Multiprocessor contains
  - 8 Scalar Processors (SP)
    - IEEE 754 32-bit floating point
    - 32-bit and 64-bit integer
  - 1 Multithreaded Instruction Unit
    - Up to 1024 concurrent threads
  - 1 Double Precision Unit: IEEE 754 64-bit floating point
  - 2 Special Function Units (SFU)
  - 16 KB shared memory
  - 16K 32-bit registers



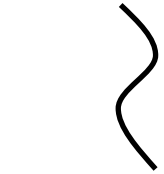
Fermi Compute Architecture

- 16 Multiprocessors
- Each Multiprocessor contains:
  - 32 Cores
    - 32 FP32 ops/clock
    - 16 FP54 ops/clock
  - 2 Warp Scheduler
    - Up to 1536 threads concurrently
  - Up to 48 KB shared memory
  - Up to 48 KB L1 cache
  - 4 SFUs
  - 32K 32-bit registers



Execution Model

OpenCL



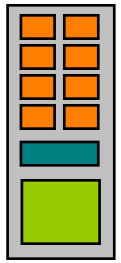
Work-item/thread

GPU

Scalar Processor

Work-item are executed by scalar processors

Work-group

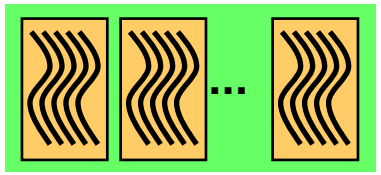


Multiprocessor

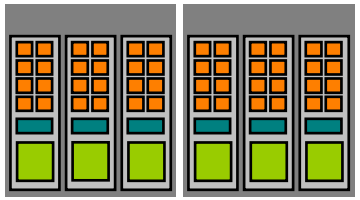
Work-groups are executed on multiprocessors

Work-groups do not migrate

Several concurrent work-groups can reside on one SM- limited by SM resources



Grid

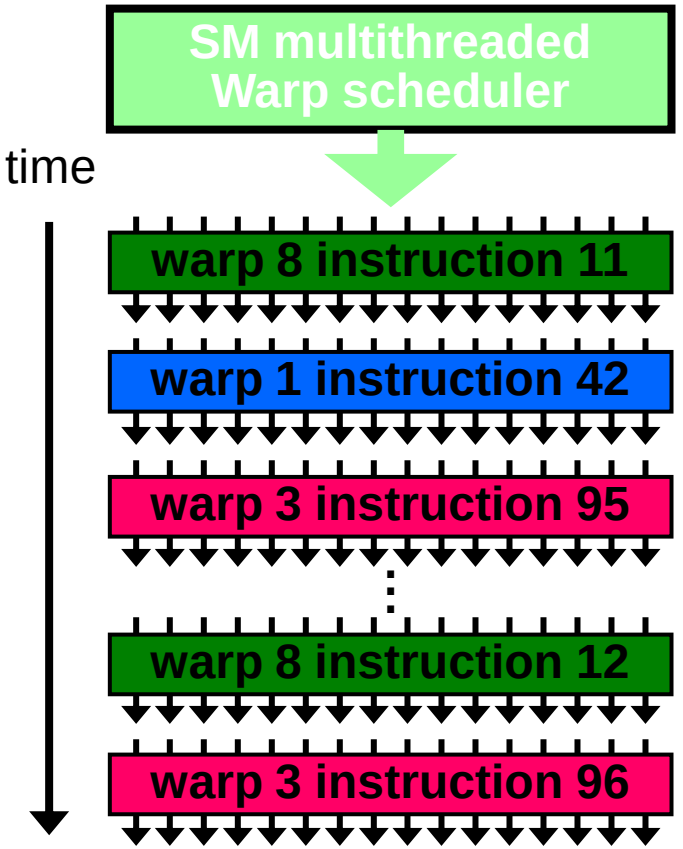


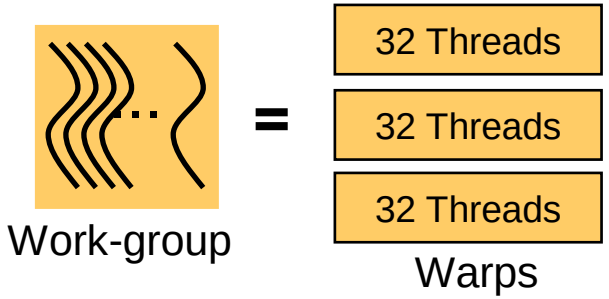
Device

A kernel is launched as a grid of work-groups

In GT200, only one kernel can execute on a device at one time (Up to 16 in Fermi)

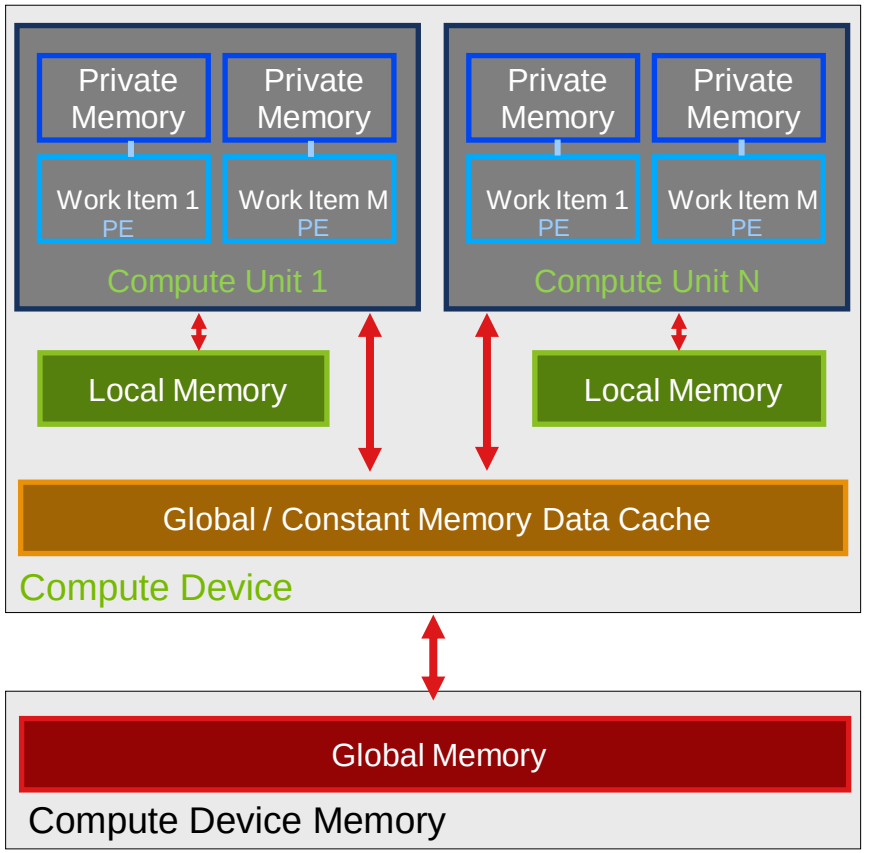
Warp and SIMT





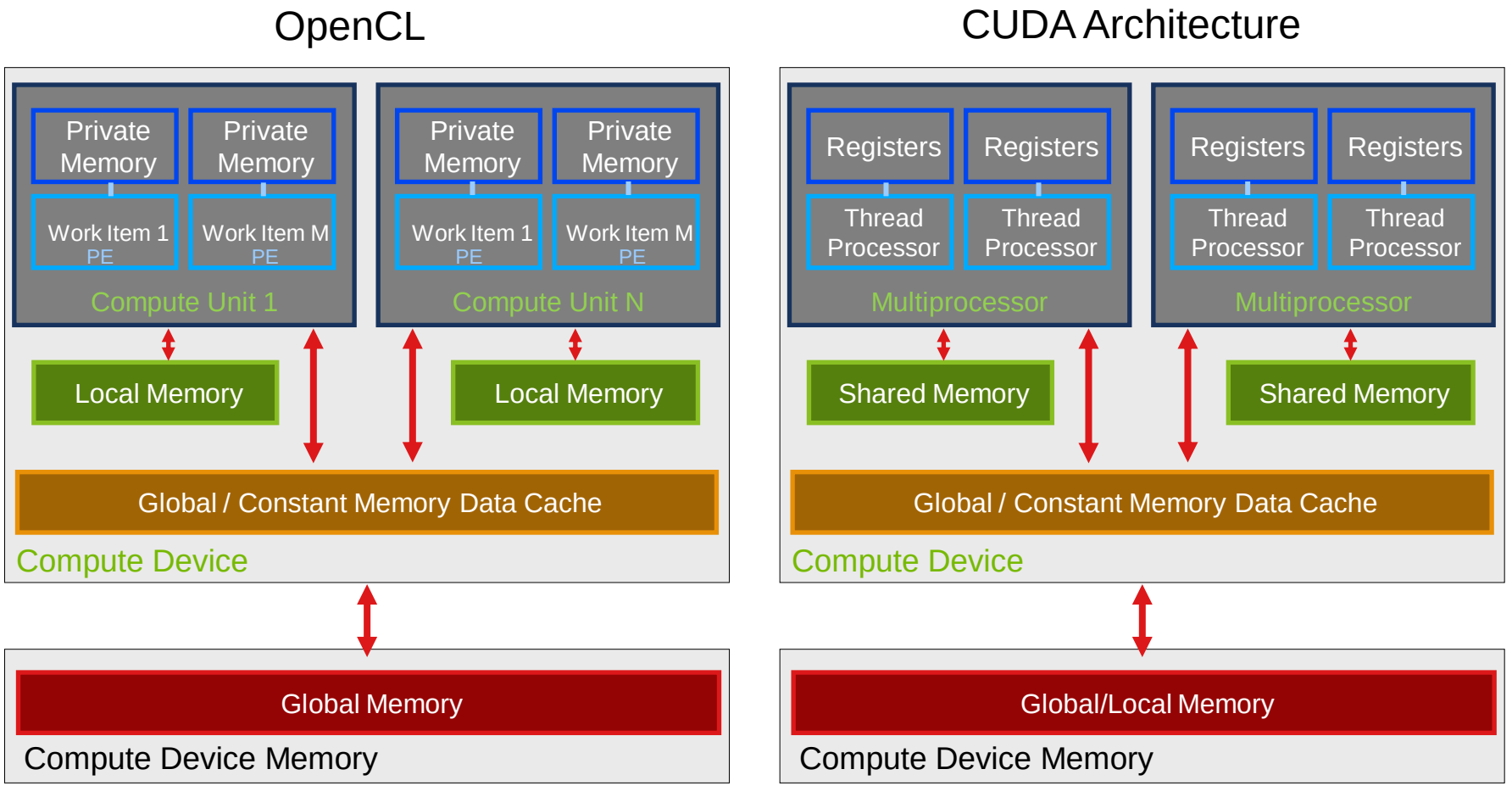
- Work-groups divided into groups of 32 threads called warps.
- Warps always perform same instruction (SIMT)
- Warps are basic scheduling units (Fermi schedules 2 warps at the same time)
- 4 clock cycles to dispatch an instruction to all the threads in a warp (2 in Fermi)
- A lot of warps can hide memory latency

OpenCL Memory Hierarchy



- Global: R/W per-kernel
- Constant : R per-kernel
- Local memory: R/W per-group
- Private: R/W per-thread

Mapping OpenCL to the CUDA Architecture



Outline

- Overview
- The CUDA architecture
- Memory optimization
- Execution configuration optimization
- Instruction optimization
- Summary

Overview of 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 (PCI-e, x16 Gen2) vs 141 GB/s (GTX 280)
- Minimize transfer
  - Intermediate data can be allocated, operated, de-allocated directly on GPU
  - Sometimes it's even better to recompute on GPU
  - Move CPU codes to GPU that do not have performance gains if it can reduce data transfer
- Group transfer
  - One large transfer much better than many small ones: latency ~ 10 microsec, thus for data size < 4KB, transfer time is dominated by latency

Coalescing

- Global memory latency: 400-800 cycles.
  The single most important performance consideration!

- Global memory access by threads of a half warp (16) can be coalesced to one transaction for word of size 8-bit, 16-bit, 32-bit, 64-bit or two transactions for 128-bit. (On Fermi, coalescing is for warp)

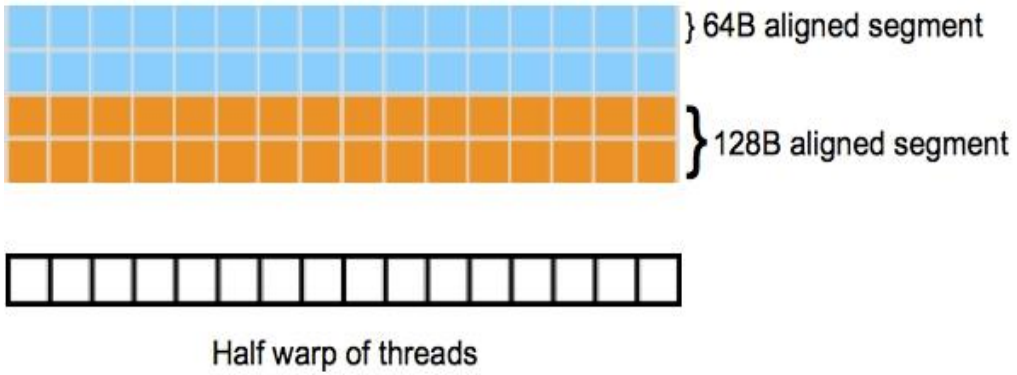
Compute Capability

- First generation CUDA architecture (G80) has compute capability 1.0, 1.1, e.g. GTX 8800, Tesla 870,
- Second generation CUDA architecture (GT200) has compute capability 1.3, e.g. GTX 280, Tesla C1060
- A full list of compute capability of various cards can be found at appendix A.1 of the OpenCL programming guide
- Target compute capability 1.0 with optional optimizations for higher compute capabilities (unless you know exactly the GPU your kernels will run on)

Segments

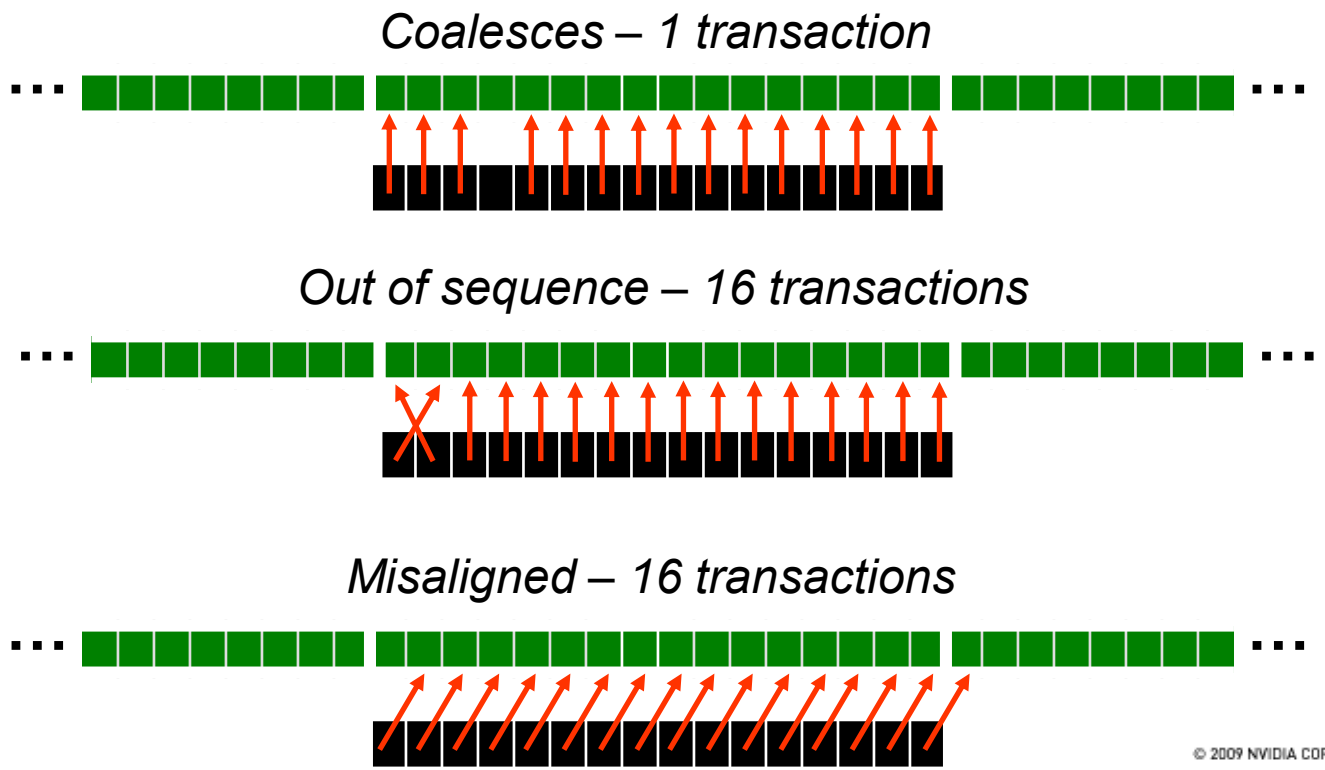
- Global memory can be viewed as composing aligned segments of 16 and 32 words.

E.g. 32-bit word:



Coalescing in 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

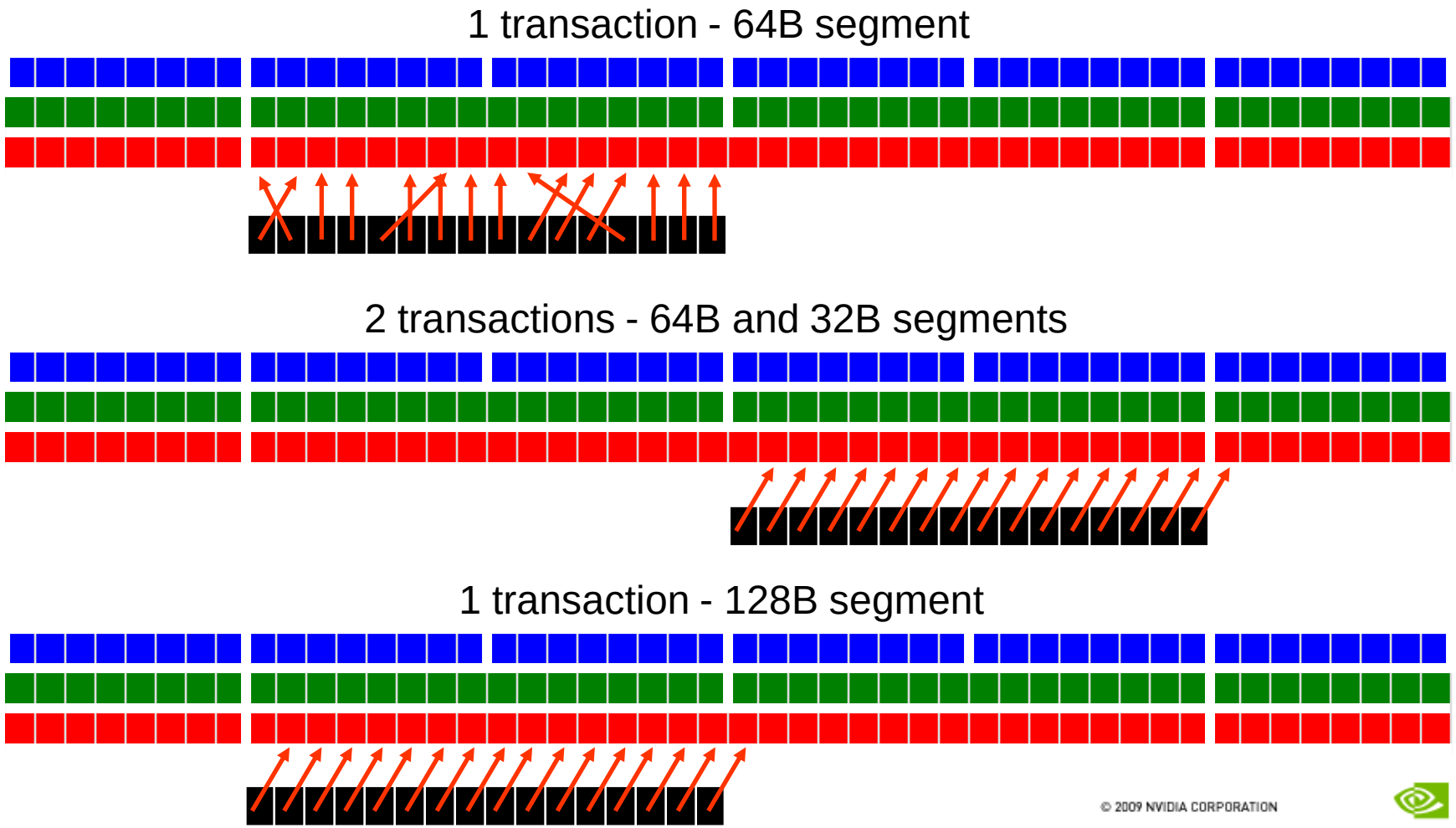


Coalescing in Compute Capability >= 1.2

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

- # of transactions = # of accessed segments

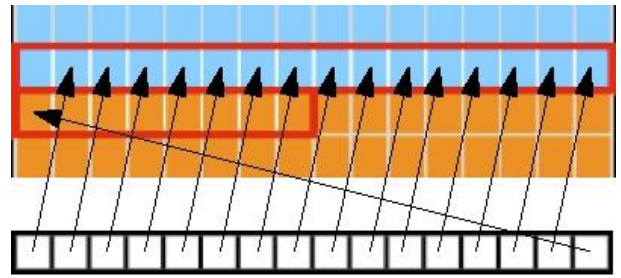
Coalescing in Compute Capability >= 1.2

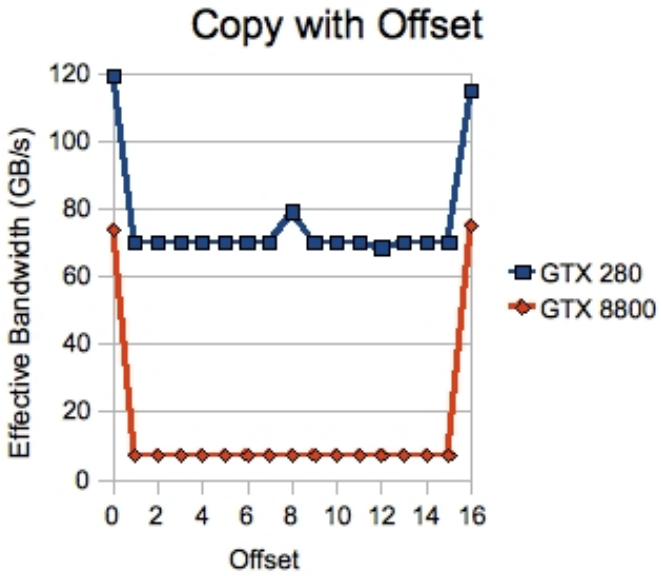


Example of Misaligned Accesses

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

offset=1

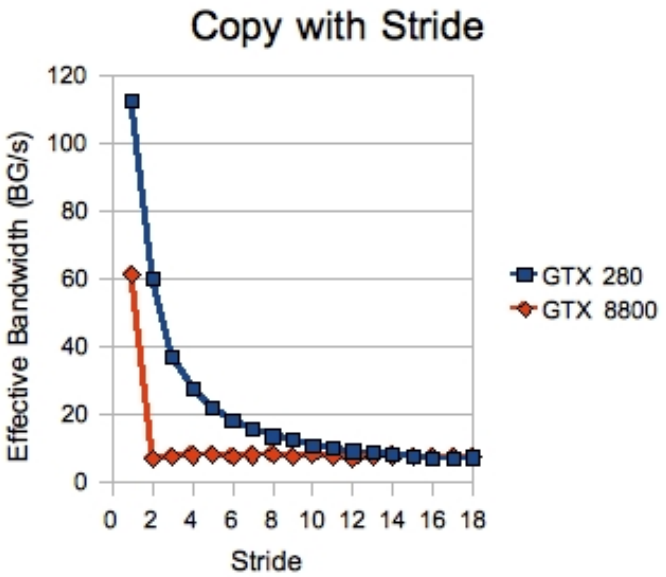




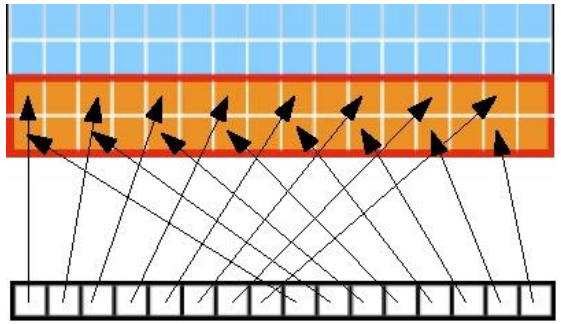
GTX280 (compute capability 1.3) drops by a factor of 1.7 while GTX 8800 (compute capability 1.0) drops by a factor of 8.

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];
}
```



stride=2



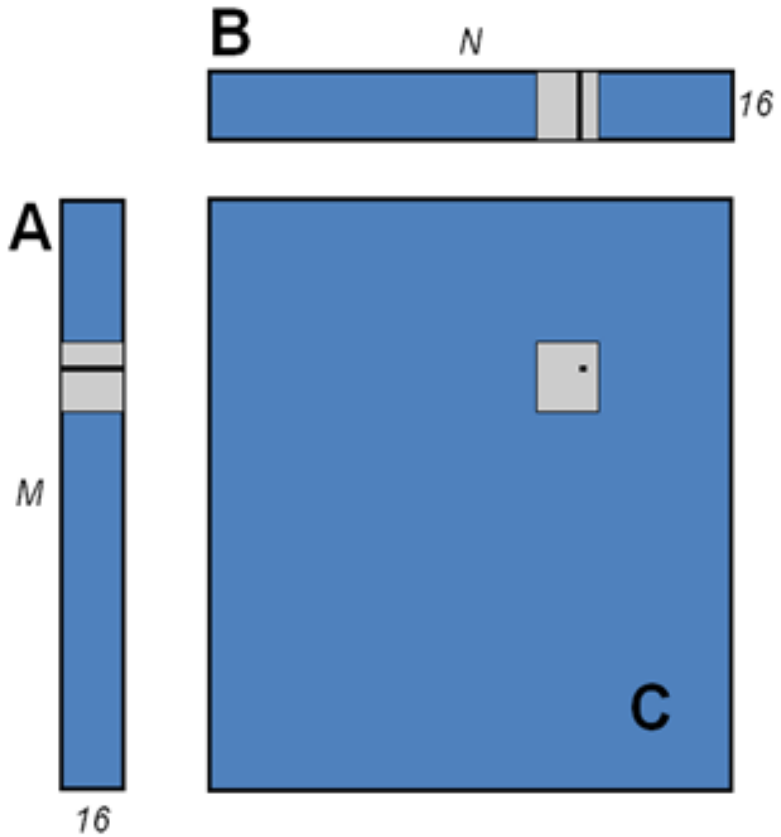
Large strides often arise in applications. However, strides can be avoided using local memory.

Graceful scaling in GTX280

Local Memory

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

Caching Example: Matrix Multiplication

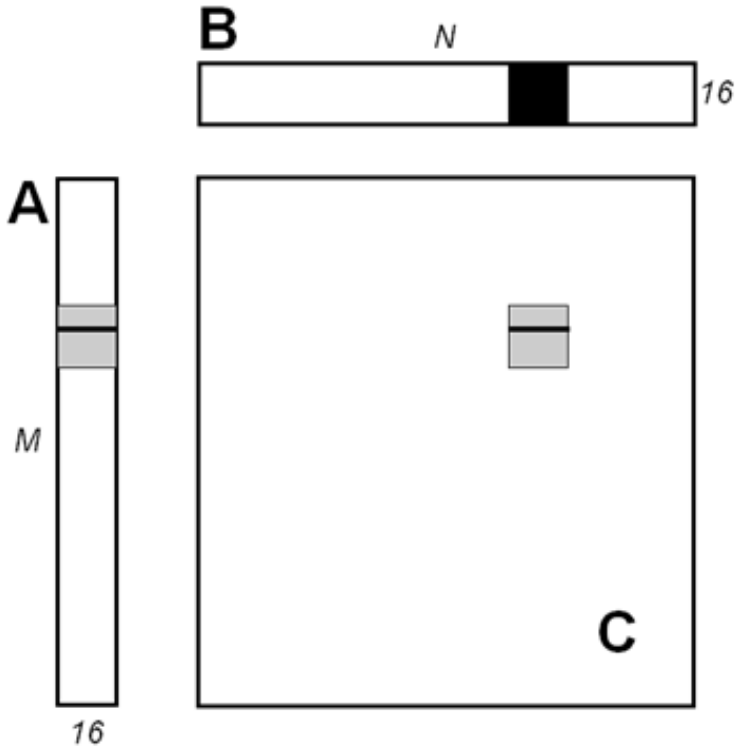


C=AxB

```
__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;
}
```

Every thread corresponds to one entry in C.

Memory Access Pattern of a Half-Warp



Each iteration, threads access the same element in A.
Un-coalesced in CC <= 1.1.

Matrix Multiplication (cont.)

Optimization	NVIDIA GeForce GTX 280	NVIDIA GeForce GTX 8800
No optimization	8.8 GBps	0.7 GBps
Coalesced using local memory to store a tile of A	14.3 GBps	8.2 GBps
Using local memory to eliminate redundant reads of a tile of B	29.7 GBps	15.7 GBps

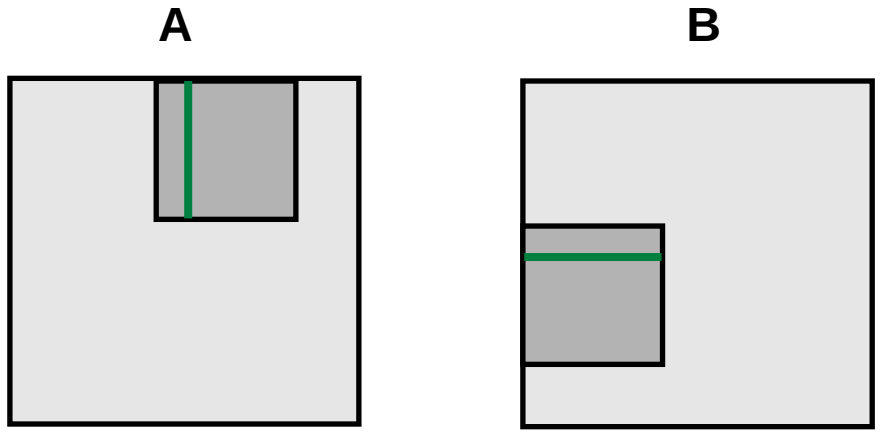
Matrix Multiplication (cont.)

```
__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;
}
```

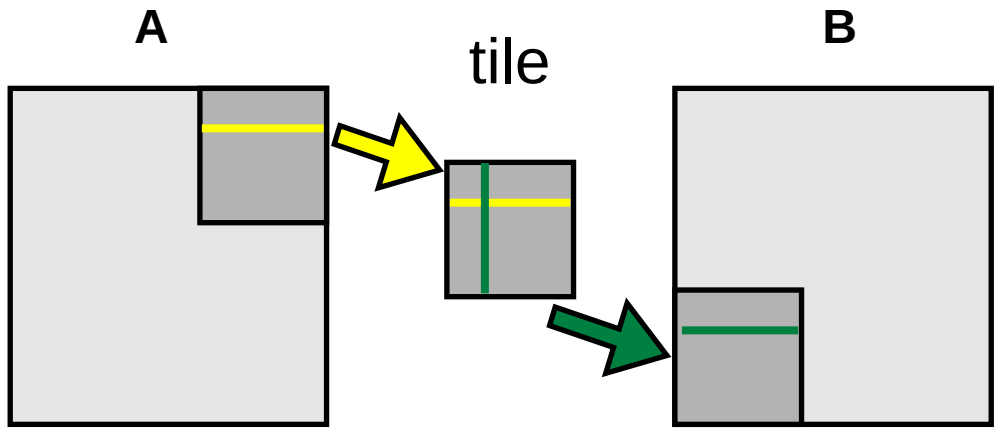
Matrix Multiplication (cont.)

Optimization	NVIDIA GeForce GTX 280	NVIDIA GeForce GTX 8800
No optimization	8.8 GBps	0.7 GBps
Coalesced using local memory to store a tile of A	14.3 GBps	8.2 GBps
Using local memory to eliminate redundant reads of a tile of B	29.7 GBps	15.7 GBps

Coalescing Example: Matrix Transpose



Strided global mem access in naïve implementation, resulting in 16 transactions if stride > 16



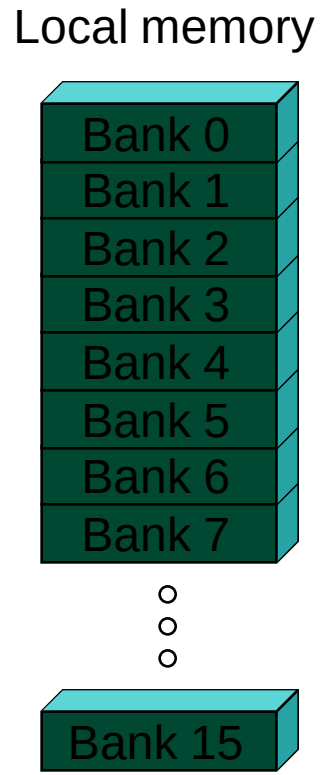
Move the strided access into local memory read

Matrix Transpose Performance

Optimization	NVIDIA GeForce GTX 280	NVIDIA GeForce GTX 8800
No optimization	1.1 GBps	0.5 GBps
Using local memory to coalesce global reads	24.8 GBps	13.2 GBps
Removing bank conflicts	30.3 GBps	15.6 GBps

Bank Conflicts

- A 2nd order effect
- Local memory is divide into banks.
  - Successive 32-bit words assigned to successive banks
  - Number of banks = 16 for CC 1.x (32 in Fermi)
- 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



Outline

- Overview
- The CUDA architecture
- Memory optimization
- Execution configuration optimization
- Instruction optimization
- Summary

Work-group Heuristics for Single Kernel

- # of work-groups > # of SM
  - Each SM has at least one work-group to execute
- # of work-groups / # of SM > 2
  - Multi work-groups can run concurrently on a 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)
- 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 issued 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
  - Maximum allowed warps: 32 in Tesla (48 in Fermi).

Latency Hiding Calculation

- Arithmetic instruction 4 cycles, global memory 400-800 cycles (assume 400 in this slide)
- 400/4 = 100 arithmetic instructions to hide the latency.
- For example, assume the code has 8 arithmetic instructions for every one global memory access. Thus 100/8~13 warps would be enough to hide the latency. This corresponds to 40% occupancy in Tesla.
- Larger than 40%, won't lead to performance gain.

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 19% occupancy in Tesla

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
  - Local memory
  - Scheduling hardware

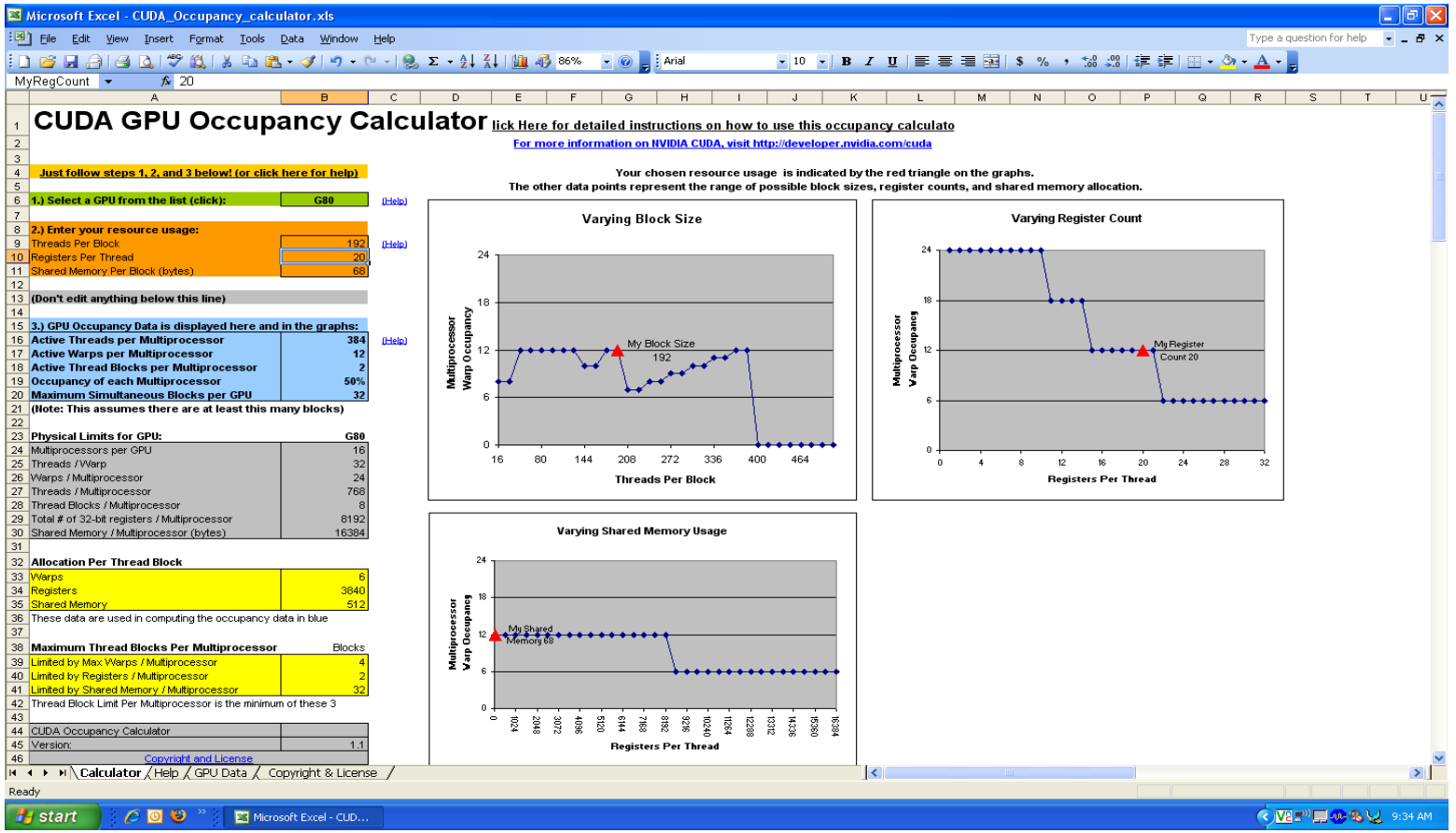
Estimating Occupancy Calculation

- Occupancy depends on both resource usage and execution configuration
- Assume the only resource limitation is register.
- If every thread uses 16 registers and every work-group has 512 work-items, then 2 work-groups use 512*16*2 <= 16384. A 100% occupancy can be achieved.
- However, if every thread uses 17 registers, since 512*17*2 > 16384, only 1 work-group is allowed. So occupancy is reduced to 50%!
- But, if work-group has 256 work-items, since 256*17*3 < 16384, occupancy can be 75%.

Other Resource Limitations on Occupancy

- Maximum number of warps (32 in Tesla and 48 in Fermi)
- Maximum number of work-groups per SM (8 in Tesla and Fermi)
- So occupancy calculation in realistic case is complicated, thus…

Occupancy Calculator



http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

NVIDIA.

Outline

- Overview
- The CUDA architecture
- Memory optimization
- Execution configuration optimization
- Instruction optimization
- Summary

Instruction Throughput

- Throughput: # of instructions per cycle, or 1/# of cycles per instruction

- In SIMT architecture,

  SM Throughtput = Operations per cycle/WarpSize

  Instruction Cycles = WarpSize/Operations per cycle

- Maximizing throughput: using smaller number of cycles to get the job done

Arithmetic Instruction

- Int add, shift, min, max: 4 cycles (Tesla), 2 cycles (Fermi)
- Int mul: 16 cycles (Tesla), 2 cycles (Fermi)
- FP32 add, mul, mad, min, max: 4 cycles (Tesla), 2 cycles (Fermi)
- FP64 add, mul, mad: 32 cycles (Tesla), 2 cycles (Fermi)
- Int divide and modulo are expensive
  - Divide by 2^n, use “>> n”
  - Modulo 2^n, use “& (2^n – 1)”
- Avoid automatic conversion of double to float
  - Adding “f” to floating literals (e.g. 1.0f) because the default is double

Math Functions

- There are two types of runtime math libraries
- Native_function() map directly to the hardware level: faster but lower accuracy
  - See appendix B of the programming guide for a list of maximum ulp for math functions
- Function(): slower but higher accuracy
  - A few to 10x more cycles, depending on whether the argument needs to be reduced
- Use native math library whenever speed is more important than precision

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.

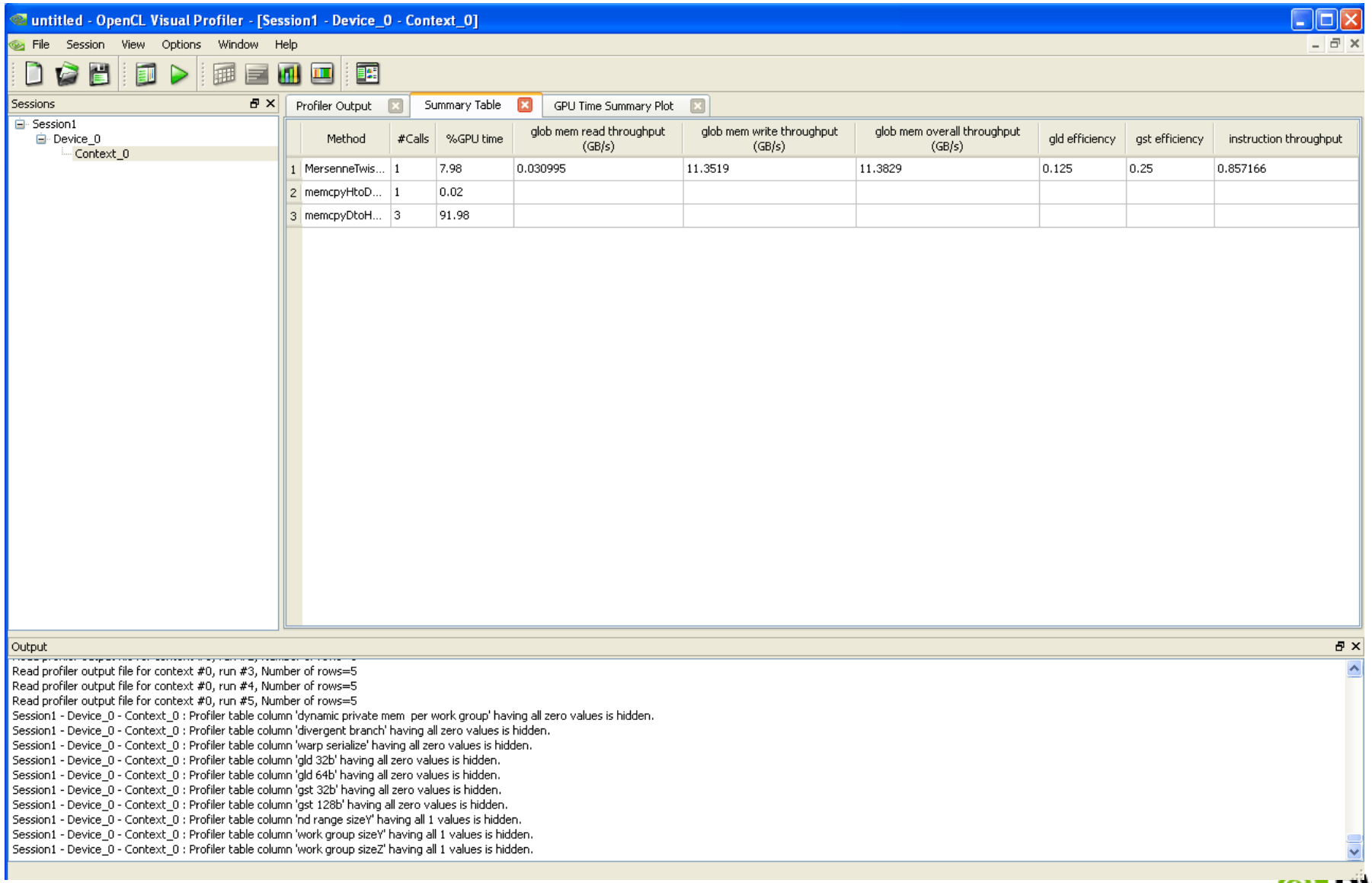
- No penalty if different warps diverge
  - E.g. no divergence in the case of
    if (local_id/warp_size > 2)
      …
    else
      …

Scalar Architecture and Compiler

- NVIDIA GPUs have a scalar architecture
  - Use vector types in OpenCL for convenience, not performance
  - Generally want more work-items rather than large vectors per work-item
- Use the -cl-mad-enable compiler option
  - Permits use of FMADs, which can lead to large performance gains
- Investigate using the -cl-fast-relaxed-math compiler option
  - enables many aggressive compiler optimizations

NVIDIA OpenCL Visual Profiler

- Profiling of kernel execution time, device<->host data transfer , calling numbers.
- Global memory bandwidth and instruction issue rate
- Number of coalesced ld/st
- Graph tools for visualizing the profiling data
- Occupancy analysis
- Export to CSV



Summary

- OpenCL programs run on GPU can achieve great performance if one can
  - Maximize parallel execution
  - Maximize memory bandwidth
  - Maximize instruction throughput

Thank you and enjoy OpenCL!