GPU TECHNOLOGY CONFERENCE

2008: OpenCL Optimizations

San Jose, CA| September 23, 2010
Peng Wang, NVIDIA

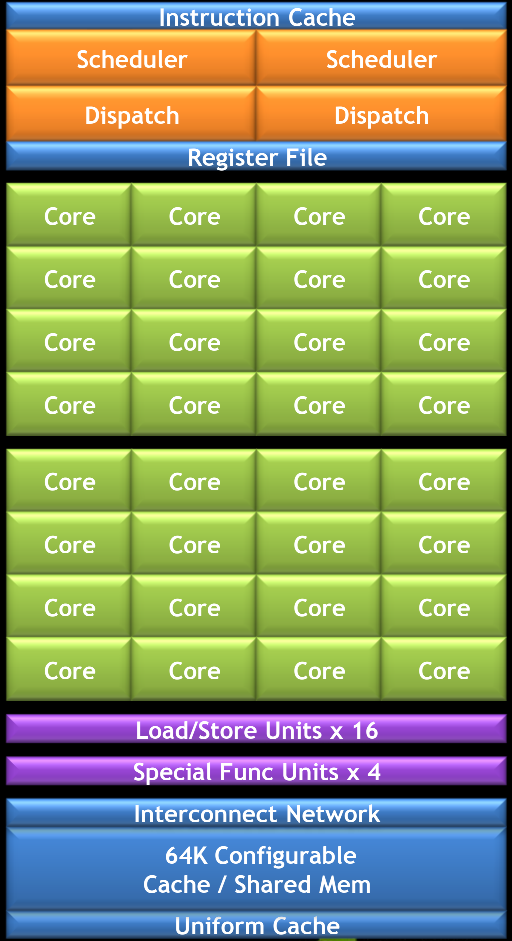
PRESENTED BY NVIDIA.

Optimization Overview

- GPU Architecture
- Memory Optimization
- Execution Configuration
- Instruction Throughput

Fermi Multiprocessor

- 2 Warp Scheduler
  - In-order issue, up to 1536 concurrent threads
- 32 CUDA Cores
  - Full IEEE 754-2008 FP32 and FP64
  - 32 FP32 ops/clock, 16 FP64 ops/clock
- 48 KB shared memory
- 16 KB L1 cache
- 4 FP32 SFUs
- 32K 32-bit registers
  - Up to 63 registers per thread



GPU and Programming Model

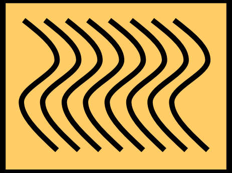
Software

GPU

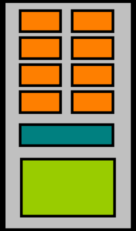
Work-item

CUDA core

Work-items are executed by CUDA cores



Work-group

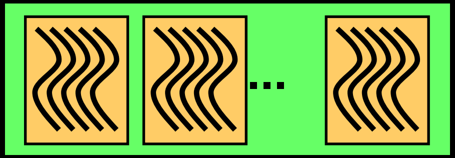


Multiprocessor

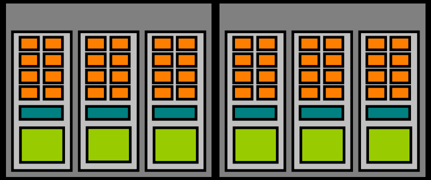
Work-groups are executed on multiprocessors

Work-groups do not migrate

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



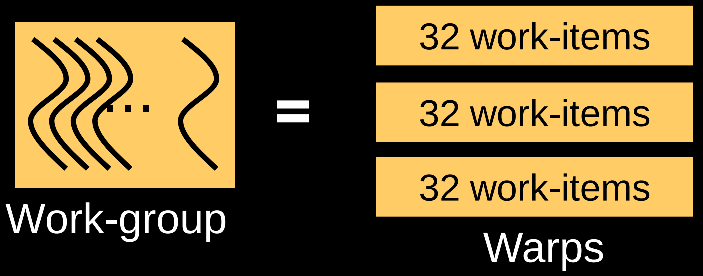
Grid

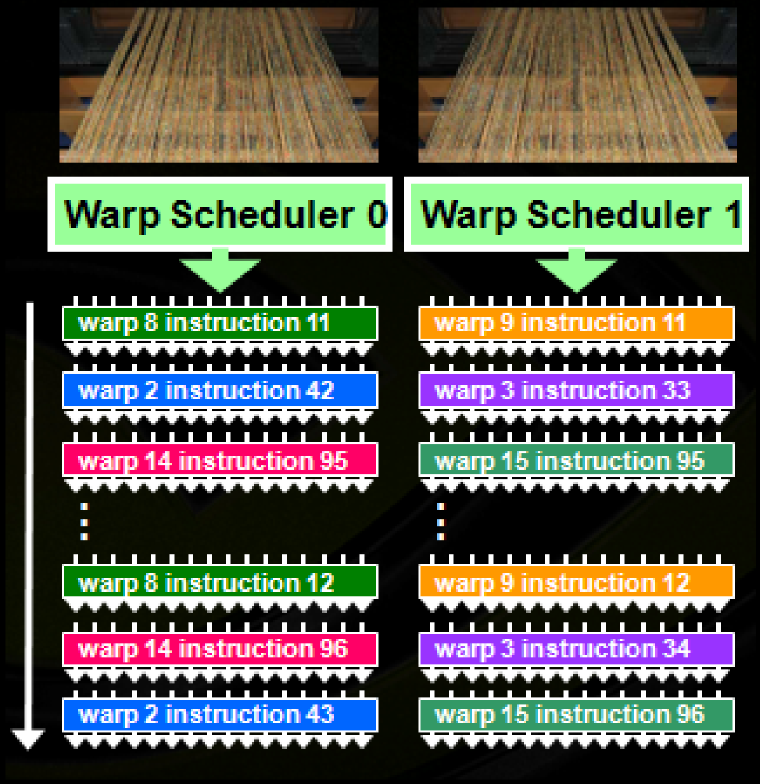


Device

A kernel is launched as a grid of work-groups

Warp and SIMT

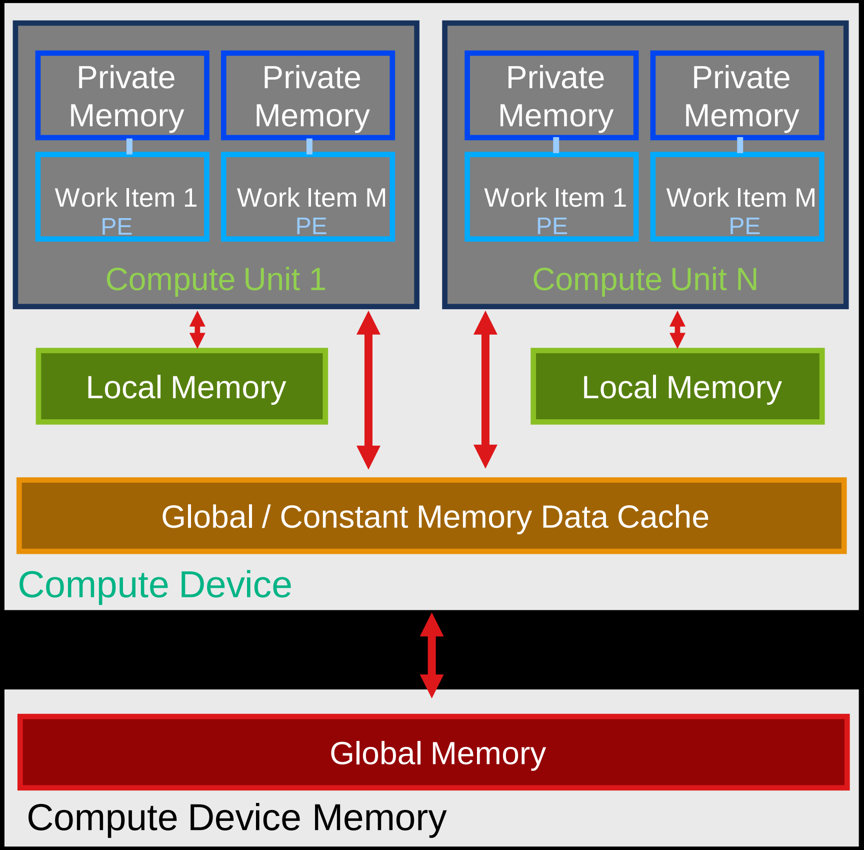




Warp Scheduler 0	Warp Scheduler 1
warp 8 instruction 11	warp 9 instruction 11
warp 2 instruction 42	warp 3 instruction 33
warp 14 instruction 95	warp 15 instruction 95
⋮	⋮
warp 8 instruction 12	warp 9 instruction 12
warp 14 instruction 96	warp 3 instruction 34
warp 2 instruction 43	warp 15 instruction 96

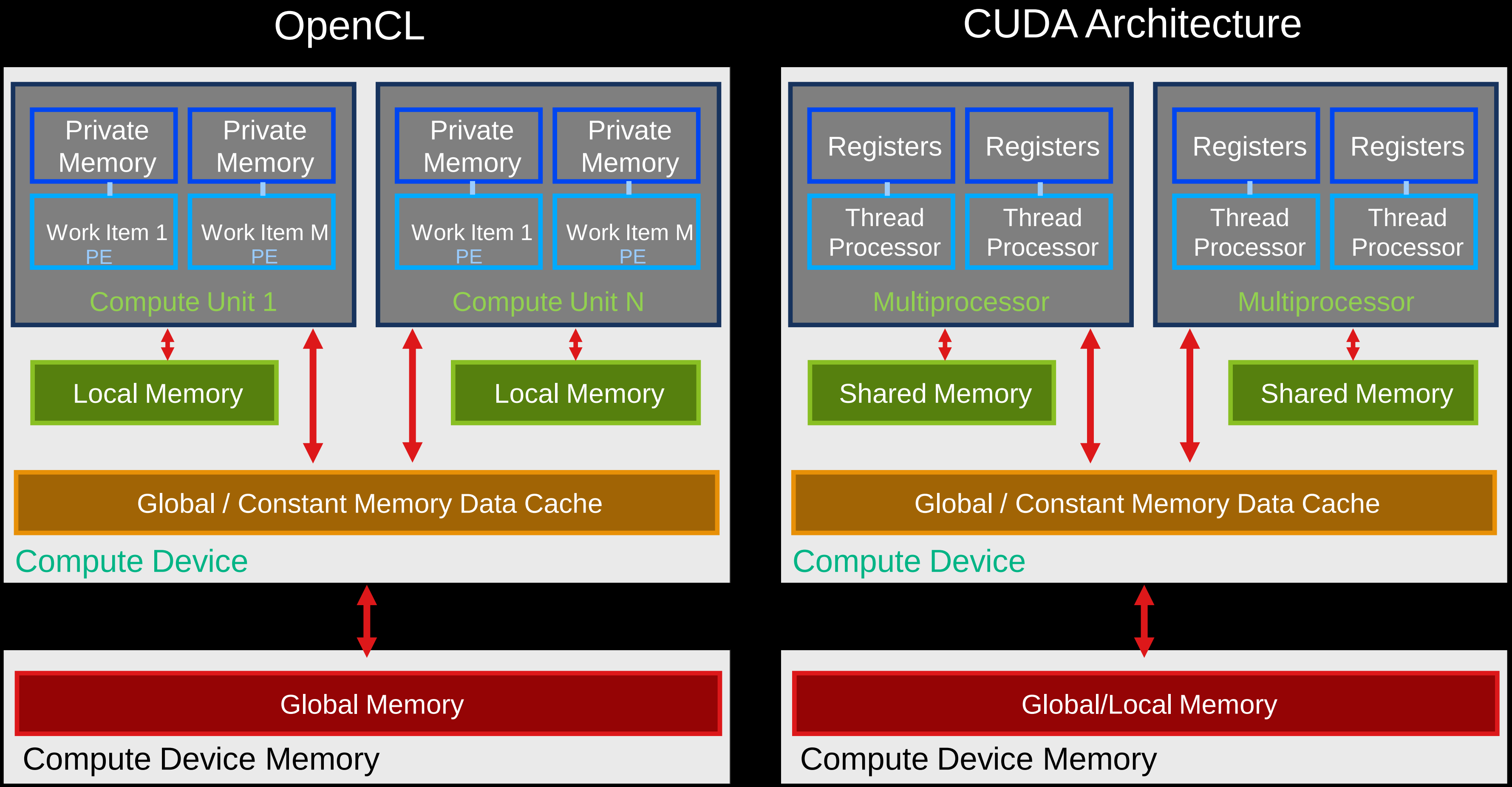
- Work-groups divide into groups of 32 work-items called warps.
- Warps are basic scheduling units
- Warps always perform same instruction (SIMT)
- Each work-item **CAN** execute its own code path
- Fermi SM has 2 warp schedulers (Tesla has 1).
- Context switching is free
- A lot of warps can hide memory latency

OpenCL Memory Hierarchy



- Global: R/W per-kernel
  - High latency: 400-800 cycles
  - Throughput: 1.5 GHz * (384 / 8) Bytes * 2 = 144 GB/s
- Constant : R per-kernel
- Local memory: R/W per-group
  - Low latency: a few cycles
  - High throughput: 73.6 GB/s per SM (1.03 TB/s per GPU)
- Private: R/W per-thread

Mapping OpenCL to the CUDA Architecture



General Optimization Strategies

- Find out the limiting factor in kernel performance
  - Memory bandwidth/latency/instruction throughput bound
  - How
    - Rule-of-thumb: compare your code’s instruction-to-byte accessed to Fermi’s peak instruction-issue-rate/bw~3.5.
    - Have good memory access pattern but effective memory throughput is low
    - Manually comment out computation and memory access: watch out for compiler tricks
- Measure effective memory/instruction throughput.
- Optimize for peak memory/instruction throughput
  - Finding out the bottleneck
  - Typically an iterative process

Minimizing CPU-GPU data transfer

- Host<->device data transfer has much lower bandwidth than global memory access.
  - 8 GB/s (PCIe x16 Gen2) vs 156 GB/s & 515 Ginst/s
- 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: 10 microsec latency, 8 GB/s => latency dominated if data size < 80 KB

Coalescing

- Global memory latency: 400-800 cycles.
  The single most important performance consideration!
- Global memory access by a warp (half-warp in pre-Fermi) can be coalesced to one transaction for word of size 8-bit, 16-bit, 32-bit, 64-bit or two transactions for 128-bit.

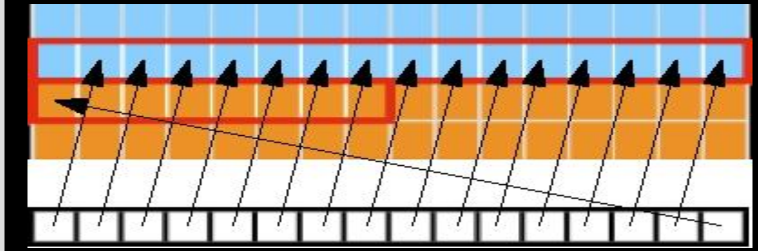
Coalescing criterion on compute capability 2.0

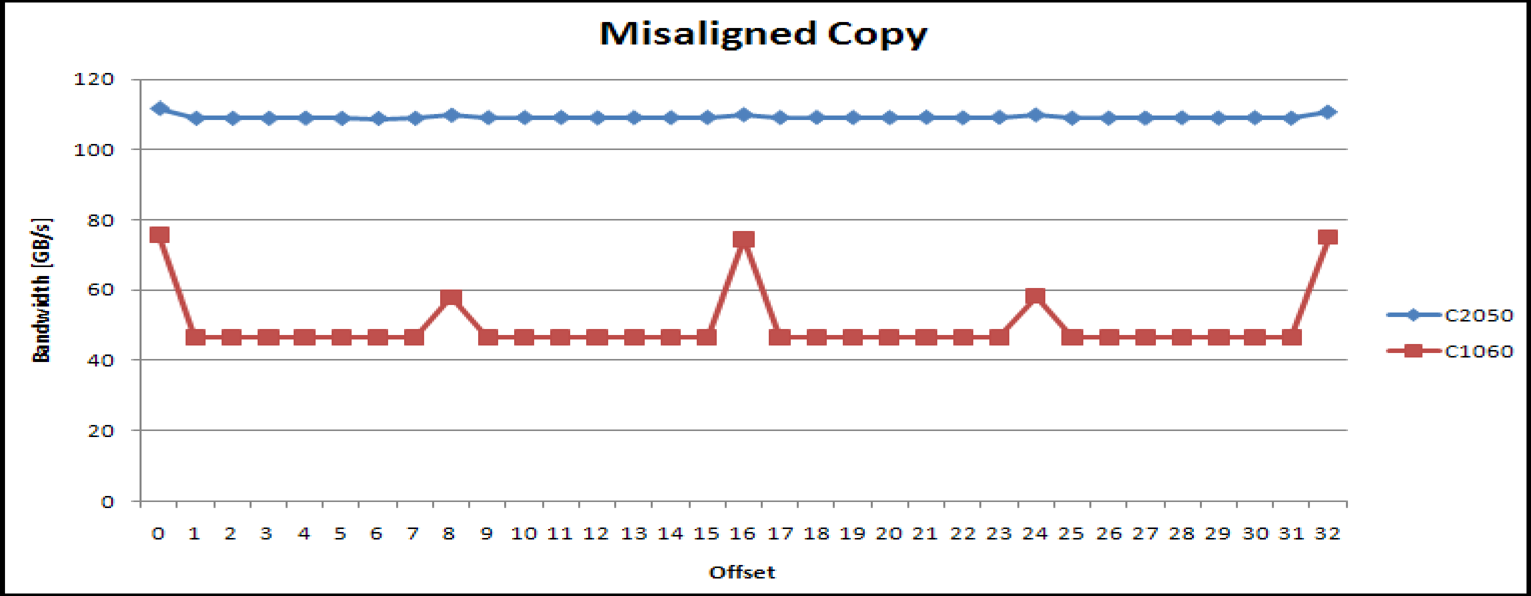
- Coalescing for any pattern of access that fits into a L1 cache line (128B)
- # of transactions = # of accessed L1 cache line

Example of Misaligned Accesses

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

offset=1



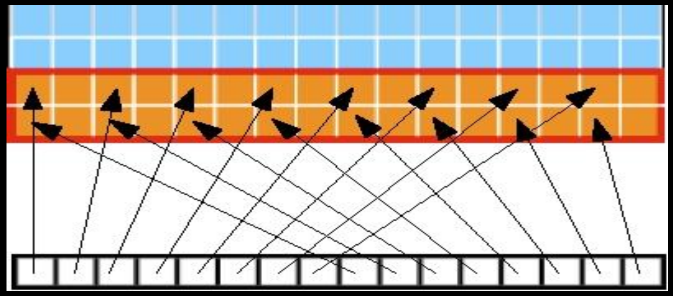


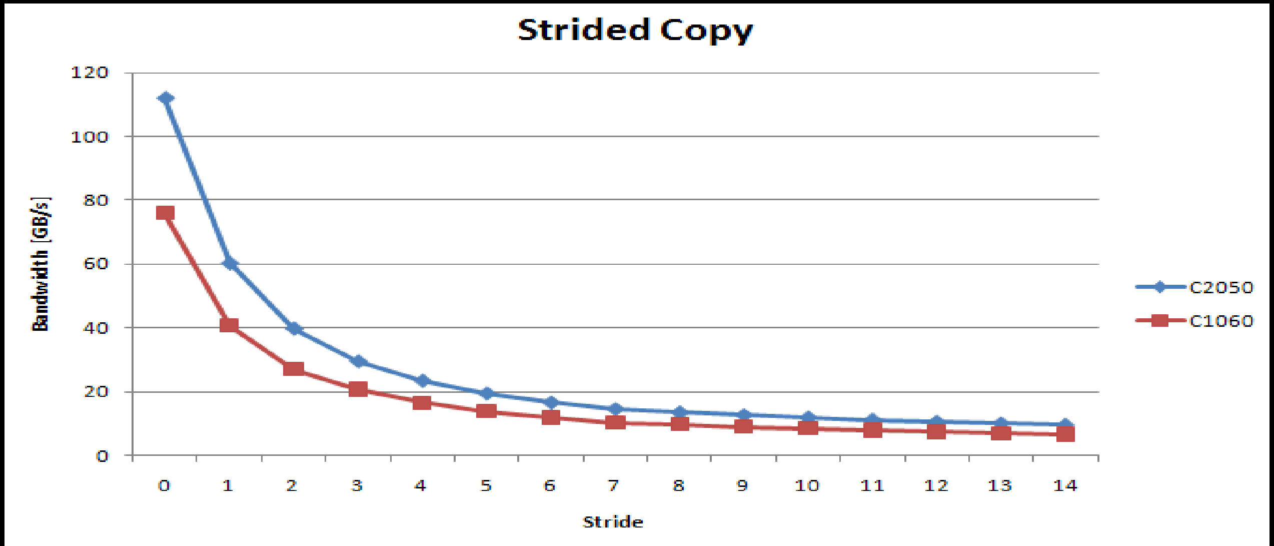
Data reuse among warps: L1 helps on misalgned access.

Example of Strided Accesses

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

stride=2





No reuse among warps

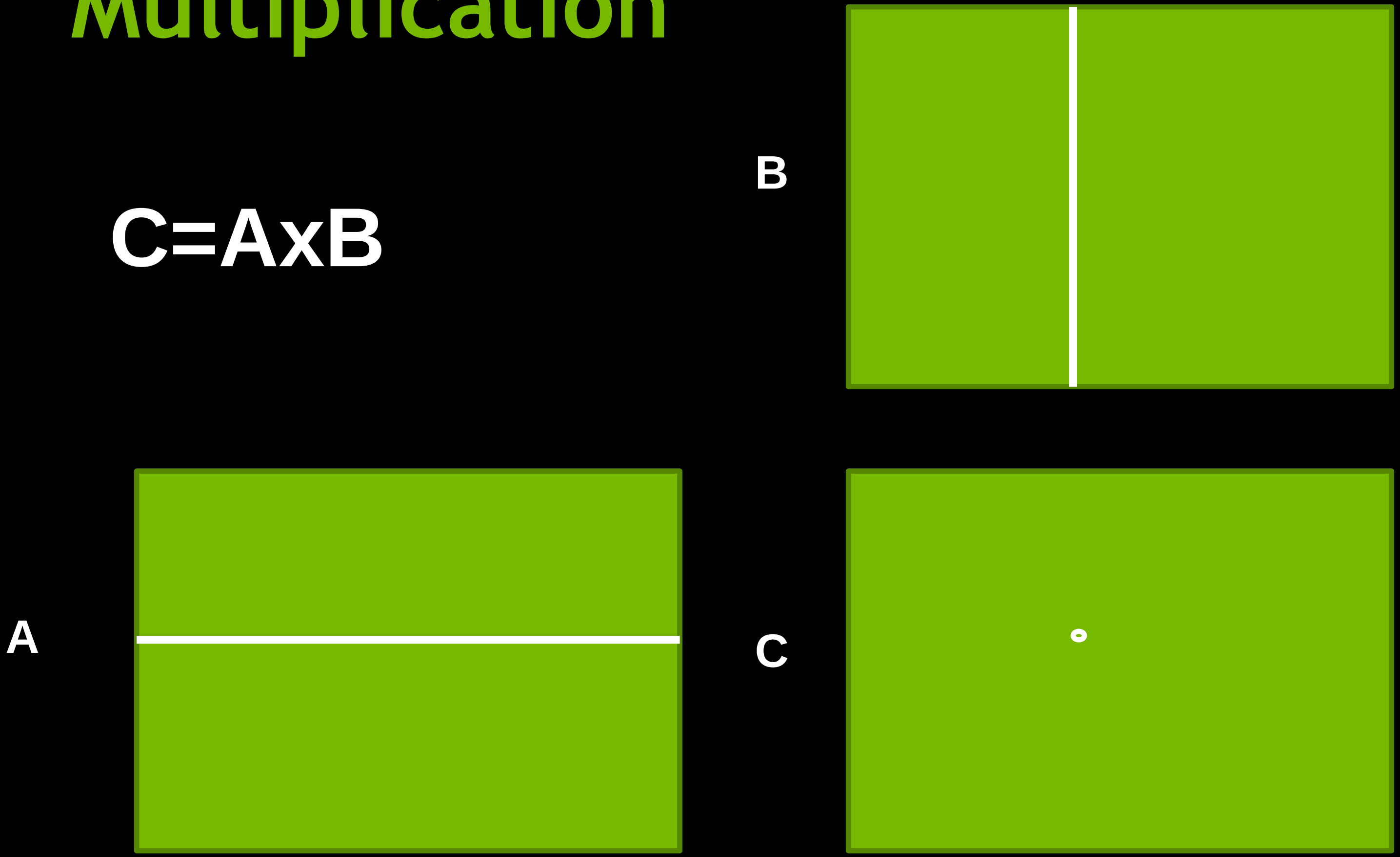
Large strides often arise in applications. However, strides may be avoided using local memory.

Local Memory

- Low latency: a few cycles
- High throughput: 73.6 GB/s per SM (1.03 TB/s per GPU)
- Main use
  - Inter-work-group communication
  - User-managed cache to reduce redundant global memory accesses
  - Avoid non-coalesced access

Local Memory Example: Matrix Multiplication

C=AxB



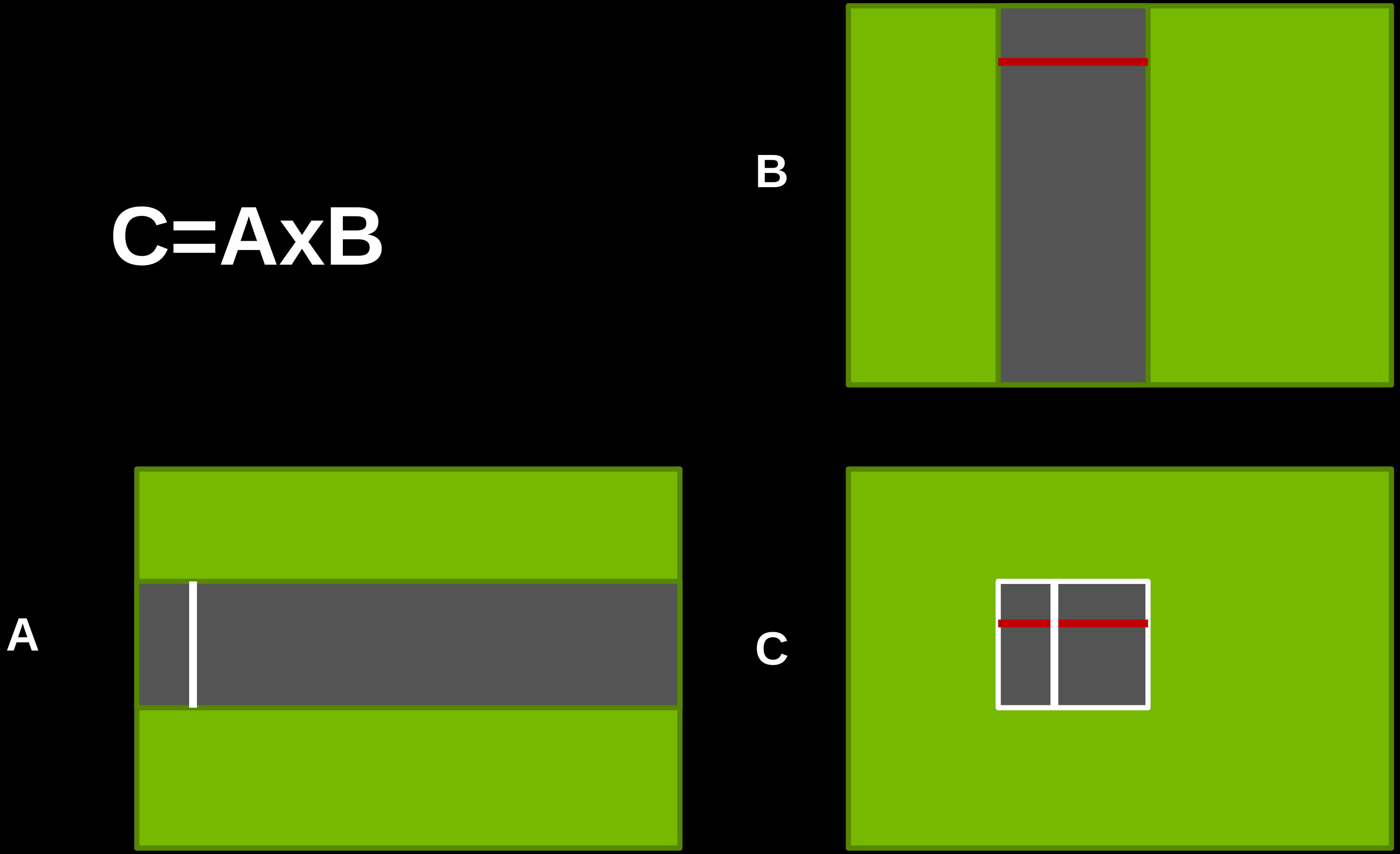
Uncached Kernel

```
__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.

Blocked Matrix Multiplication

C=AxB



Data reuse in the blocked version

Blocked and cached kernel

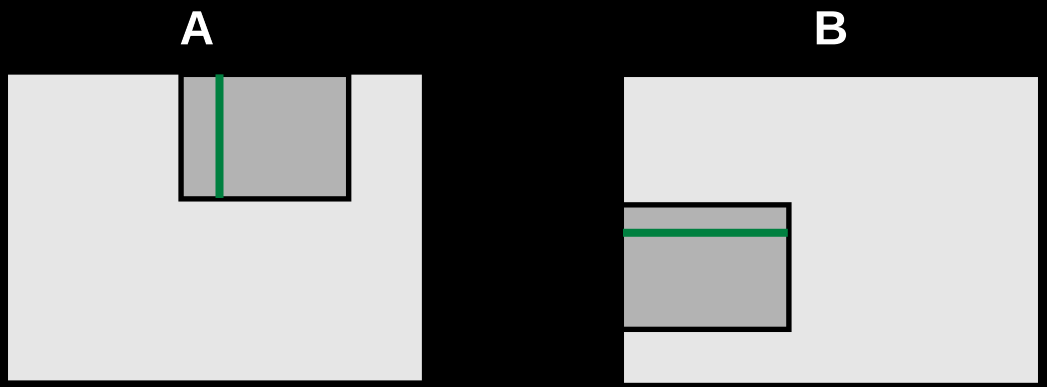
```
__kernel void coalescedMultiply(double*a,
                                double* b,
                                double*c,
                                int N)
{
  __local float aTile[TILE_DIM][TILE_DIM];
  __local double bTile[TILE_DIM][TILE_DIM];

  int row = get_global_id(1);
  int col = get_global_id(0);
  float sum = 0.0f;
  for (int k = 0; k < N; k += TILE_DIM) {
    aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x];
    bTile[threadIdx.y][threadIdx.x] = b[threadIdx.y*N+col];
    barrier(CLK_LOCAL_MEM_FENCE);
    for (int i = k; i < k+TILE_DIM; i++) {
      sum += aTile[threadIdx.y][i]* bTile[i][threadIdx.x];
  }
  c[row*N+col] = sum;
}
```

Performance Results

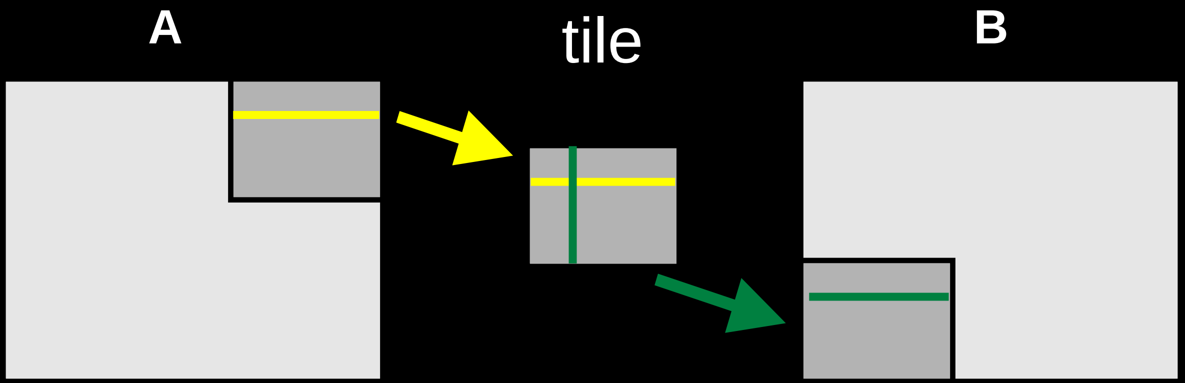
Optimization	C1060	C2050
A, B in global	12 Gflop/s	57 Gflop/s
A, B in local	125 Gflop/s	181 Gflop/s

Coalescing Example: Matrix Transpose

B=A’



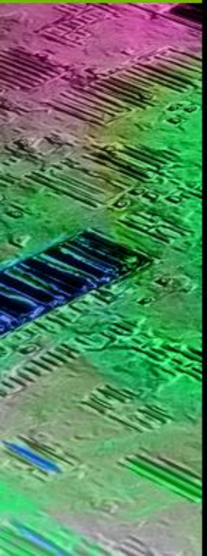
Strided global mem access in naïve implementation, resulting in 32 transactions if stride > 32



Move the strided access into local memory read

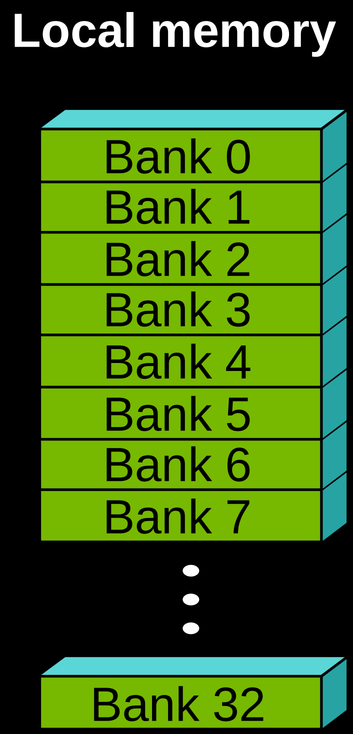
Matrix Transpose Performance

Optimization	C1060	C2050
No optimization	1.6 GB/s	24 GB/s
Using local memory to coalesce global reads	13.4 GB/s	38.8 GB/s



Bank Conflicts

- Local memory is divide into banks.
  - Successive 32-bit words assigned to successive banks
  - Number of banks = 32 (Fermi), 16 (Tesla)
- Bank conflict: two R/W fall in the same bank, the access will be serialized.



Bank Conflicts

- Special cases
  - If all threads in a warp access the same word, one broadcast. Fermi can also do multi-broadcast.
  - If reading continuous byte, no conflict on Fermi
  - If reading double, no conflict on Fermi
- Some tricks
  - Use array[N_BANK][N_BANK+1];
  - Change local memory reads to the same value to see the impact

Memory Optimizations

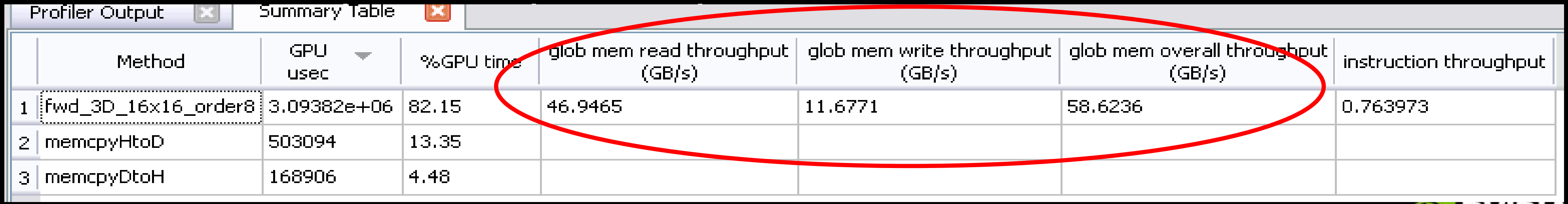
- Strive for perfect coalescing
  - Transpose the data structure, e.g. AOS to SOA; padding
- Launch enough threads per SM to cover latency
- Process several elements per thread
  - Multiple loads get pipelined; indexing calculation may be reused
- Issue global loads as early as possible
  - Group together; prefetch
- Use local memory to reduce global memory access, avoid non-coalesced access.

Global Memory Throughput Metric

- Many codes are memory throughput bound
- Measuring effective memory throughput:
  - From the app point of view ("useful" bytes): number of bytes needed by the algorithm divided by kernel time
  - Compare to the theoretical bandwidth
    - 70-80% is very good
- Finding out bottleneck
  - Start with global memory operations, achieve good throughput
  - Add arithmetic, local memory, etc, measuring perf as you go

Visual Profiler

- Latest Visual Profiler reports memory throughput
  - From HW point of view: count load/store bus transactions of each size (32, 64, 128B) on the TPC
  - Based on counters for one TPC (3 multiprocessors), extrapolate to the whole GPU
  - Need compute capability 1.2 or higher GPU
- The effective and HW memory throughputs are likely to be different



	Method	GPU usec	%GPU time	glob mem read throughput (GB/s)	glob mem write throughput (GB/s)	glob mem overall throughput (GB/s)	instruction throughput
1	fwd_3D_16x16_order8	3.09382e+06	82.15	46.9465	11.6771	58.6236	0.763973
2	memcpyHtoD	503094	13.35				
3	memcpyDtoH	168906	4.48				

Grid Size Heuristics

- # of work-groups / # of SM > 2
  - Multi blocks 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-group Size Heuristics

- Work-group size should be a multiple of 32 (warp size)
- Want as many warps running as possible to hide latencies
- Minimum: 64. I generally use 128 or 256. But use whatever is best for your app.
- Depends on the problem, do experiments!

Latency Hiding

- Key to understanding:
  - Instructions are issued in order
  - A work-item blocks when one of the operands isn’t ready:
  - Latency is hidden by switching warps
- Conclusion:
  - Need enough warps to hide latency

Occupancy

- Occupancy: ratio of active warps per SM to the maximum number of allowed warps
- Maximum number: 32 in pre-Fermi, 48 in Fermi

Dynamical Partitioning of SM Resources

- Local memory is partitioned among blocks
- Registers are partitioned among threads: <= 63
- Work-group slots: <= 8
- Work-item slots: <= 1536
- Any of those can be the limiting factor on how many work-items can be launched at the same time on a SM

Latency Hiding Occupancy Calculation

- Assume global memory takes 400 cycles, we need 400 arithmetic instructions to hide the latency.
- For example, assume the code has 16 independent arithmetic instructions for every one global memory access. Thus 400/16~26 warps would be enough (54% occupancy).
- Note beyond 54%, in this example higher occupancy won’t lead to performance increase.

Register Dependency Latency Hiding

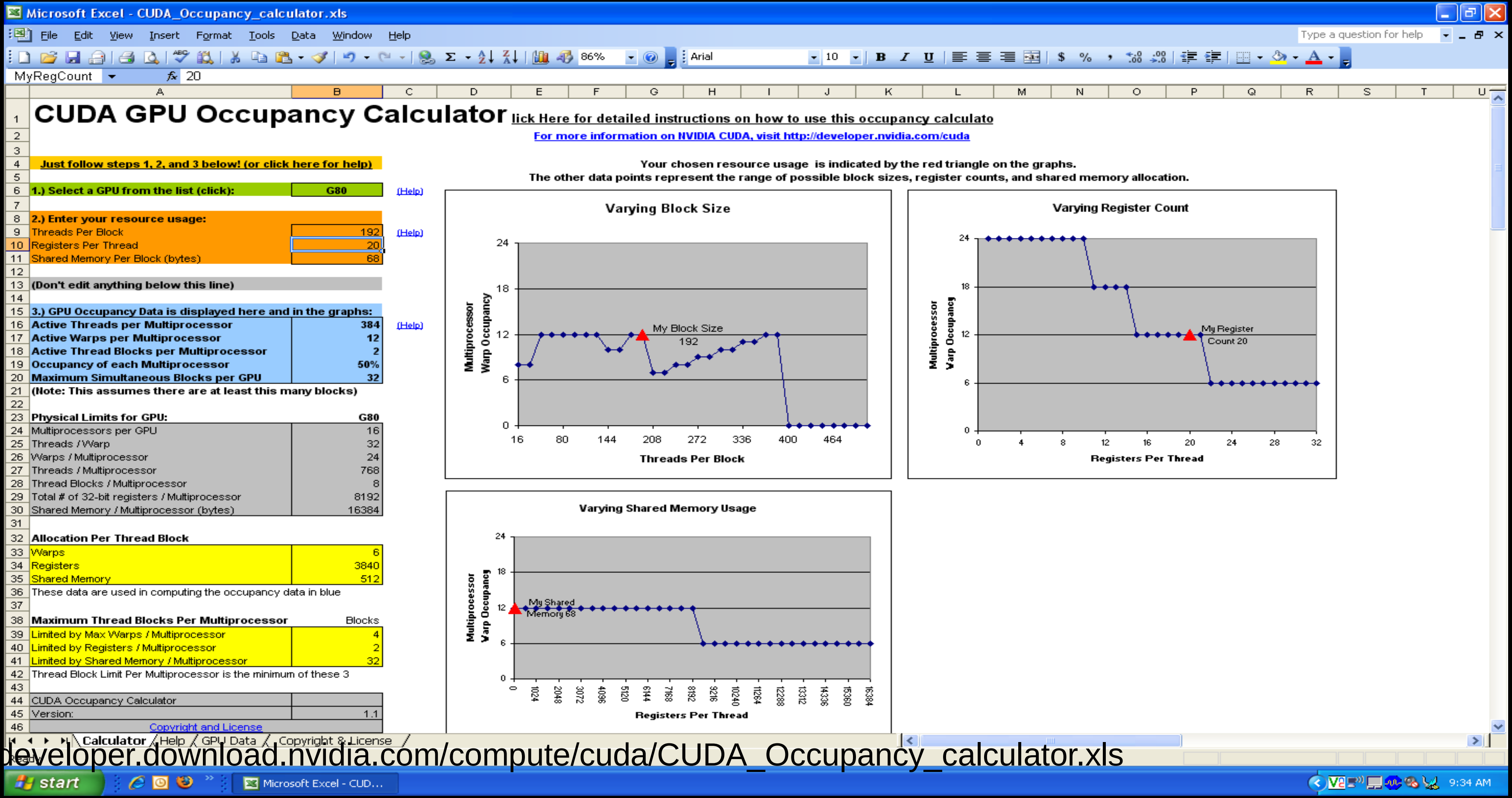
- If an instruction uses a result stored in a register written by an instruction before it, this is ~ 24 cycles latency

- So in the worst case, we need 24 warps to hide register dependency latency. This corresponds to 50% occupancy

Occupancy Optimizations

- **Increase occupancy to achieve latency hiding**
- If adding a single instruction leads to significant perf drop, occupancy is the primary suspect
- Output resource usage info
  - Dump ptx, then pass to ptxas with option -v
- Compiler option –nv-cl-maxrregcount=n
- Dynamical allocating local memory
- After some point (generally 50%), further increase in occupancy won’t lead to performance increase

Occupancy Calculator



1	**CUDA GPU Occupancy Calculator**	lick Here for detailed instructions on how to use this occupancy calculato	
2		For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda	
4	Just follow steps 1, 2, and 3 below! (or click here for help)		
6	1.) Select a GPU from the list (click):	G80	(Help)
8	2.) Enter your resource usage:		
9	Threads Per Block	192	(Help)
10	Registers Per Thread	20	
11	Shared Memory Per Block (bytes)	68	
13	(Don't edit anything below this line)		
15	3.) GPU Occupancy Data is displayed here and in the graphs:		
16	Active Threads per Multiprocessor	384	(Help)
17	Active Warps per Multiprocessor	12	
18	Active Thread Blocks per Multiprocessor	2	
19	Occupancy of each Multiprocessor	50%	
20	Maximum Simultaneous Blocks per GPU	32	
21	(Note: This assumes there are at least this many blocks)		
23	Physical Limits for GPU:	G80	
24	Multiprocessors per GPU	16	
25	Threads / Warp	32	
26	Warps / Multiprocessor	24	
27	Threads / Multiprocessor	768	
28	Thread Blocks / Multiprocessor	8	
29	Total # of 32-bit registers / Multiprocessor	8192	
30	Shared Memory / Multiprocessor (bytes)	16384	
32	Allocation Per Thread Block		
33	Warps	6	
34	Registers	3840	
35	Shared Memory	512	
36	These data are used in computing the occupancy data in blue		
38	Maximum Thread Blocks Per Multiprocessor	Blocks	
39	Limited by Max Warps / Multiprocessor	4	
40	Limited by Registers / Multiprocessor	2	
41	Limited by Shared Memory / Multiprocessor	32	
42	Thread Block Limit Per Multiprocessor is the minimum of these 3		
44	CUDA Occupancy Calculator		
45	Version:	1.1	
46	Copyright and License		

Your chosen resource usage is indicated by the red triangle on the graphs.
The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

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

Instruction Optimization

- If you find out the code is instruction bound
  - Compute-intensive algorithm can easily become memory-bound if not careful enough
  - Typically, worry about instruction optimization after memory and execution configuration optimizations

Fermi Arithmetic Instruction Throughputs

- Int & fp32: 2 cycles
- fp64: 2 cycles
- Fp32 transendental: 8 cycles
- 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)

Runtime Math Library and Intrinsics

- Two types of runtime math library functions
  - func():
    - Slower but higher accuracy (5 ulp or less)
    - Examples: sin(x), exp(x), pow(x, y)
  - native__func():
    - Fast but lower accuracy (see prog. guide for full details)
    - Examples: __sin(x), __exp(x), __pow(x, y)
- A number of additional intrinsics:
  - native__sincos(), native__rcp(), ...
- Use –cl-fast-relaxed-math

Control Flow

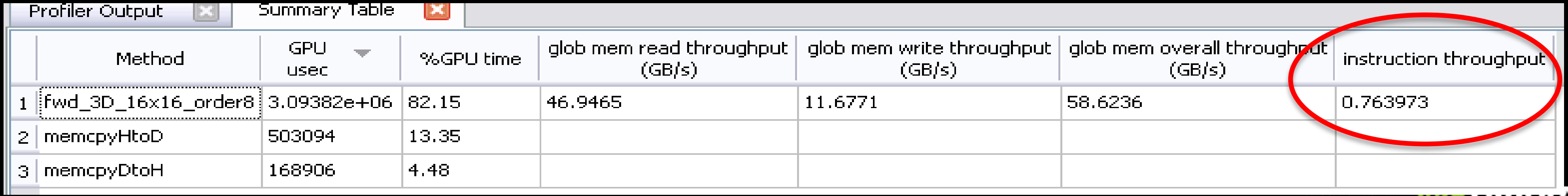
- Instructions are issued per 32 threads (warp)
- Divergent branches:
  - Threads within a single warp take different paths
    - if-else, ...
  - Different execution paths within a warp are serialized
- Different warps can execute different code with no impact on performance

Control Flow

- Avoid diverging within a warp
  - Example with divergence:
    - `if (get_local_id(0) > 2) {...} else {...}`
    - Branch granularity < warp size
  - Example without divergence:
    - `if (get_local_id(0) / WARP_SIZE > 2) {...} else {...}`
    - Branch granularity is a whole multiple of warp size

Profiler and Instruction Throughput

- **Visual Profiler derives:**
  - – Instruction throughput
    - Fraction of SP arithmetic instructions that could have been issued in the same amount of time
      - – So, not a good metric for code with DP arithmetic or transcendentals
  - – Extrapolated from one multiprocessor to GPU
- Change the conditional statement and see how that affect the instruction throughput



	Method	GPU usec	%GPU time	glob mem read throughput (GB/s)	glob mem write throughput (GB/s)	glob mem overall throughput (GB/s)	instruction throughput
1	fwd_3D_16x16_order8	3.09382e+06	82.15	46.9465	11.6771	58.6236	0.763973
2	memcpyHtoD	503094	13.35				
3	memcpyDtoH	168906	4.48				

PRESENTED BY

Summary

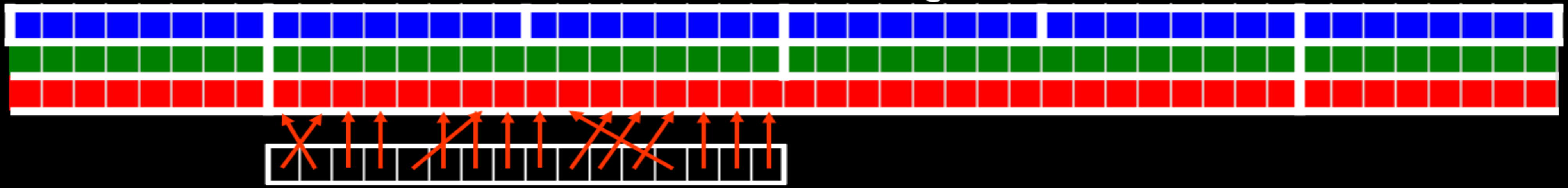
- Optimization needs an understanding of GPU architecture
- Memory optimization: coalescing, local memory
- Execution configuration: latency hiding
- Instruction throughput: use high throughput inst
- Do measurements!
  - Use the Profiler, simple code modifications
  - Compare to theoretical peaks

Coalescing on compute capability 1.2, 1.3

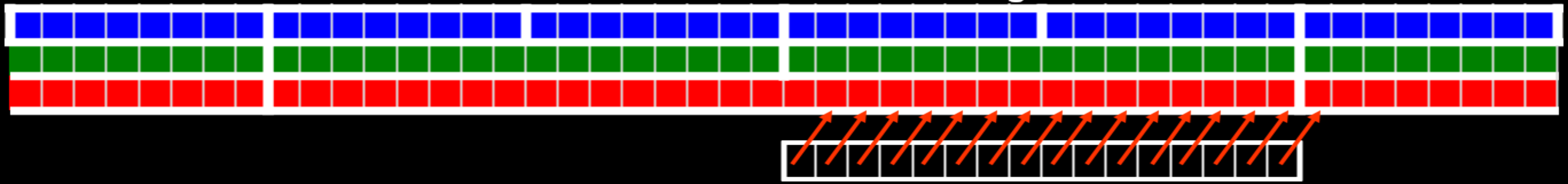
- Coalescing for each half-warp (16 threads)
- Possible GPU transaction size: 32B, 64B, or 128B
- Reduce transaction size when possible
  - Find the segment that contains the address requested
  - If only half of the segments are used, reduce the transaction size

Coalescing example

1 transaction - 64B segment



2 transactions - 64B and 32B segments



1 transaction - 128B segment