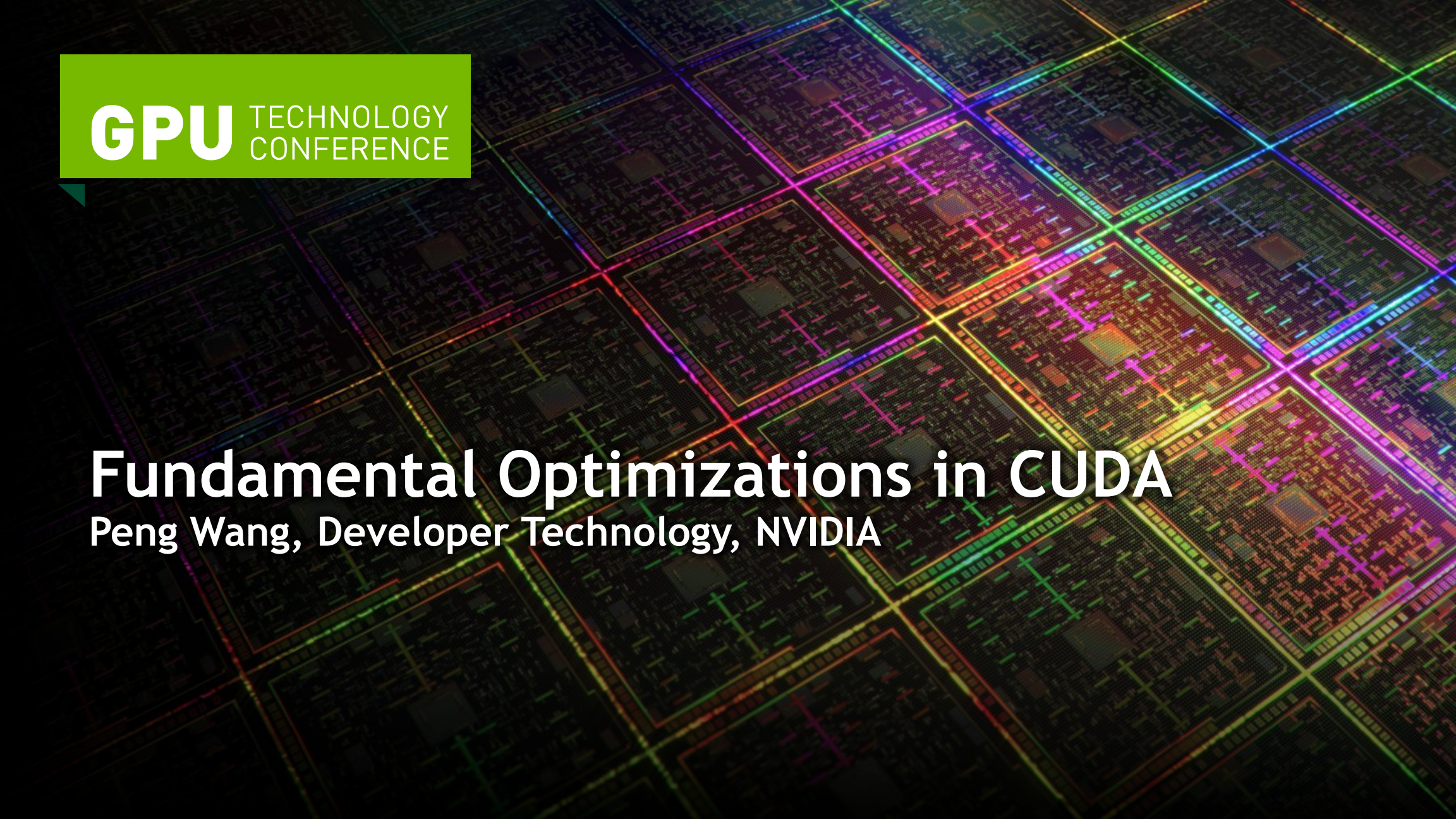
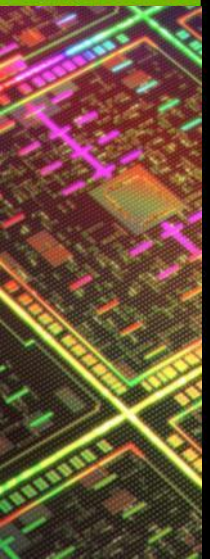


Fundamental Optimizations in CUDA

Peng Wang, Developer Technology, NVIDIA

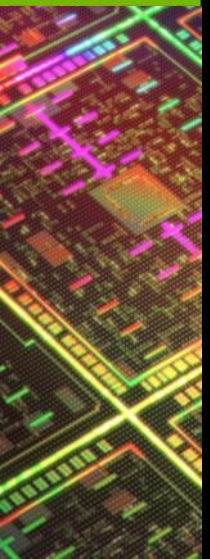


Optimization Overview

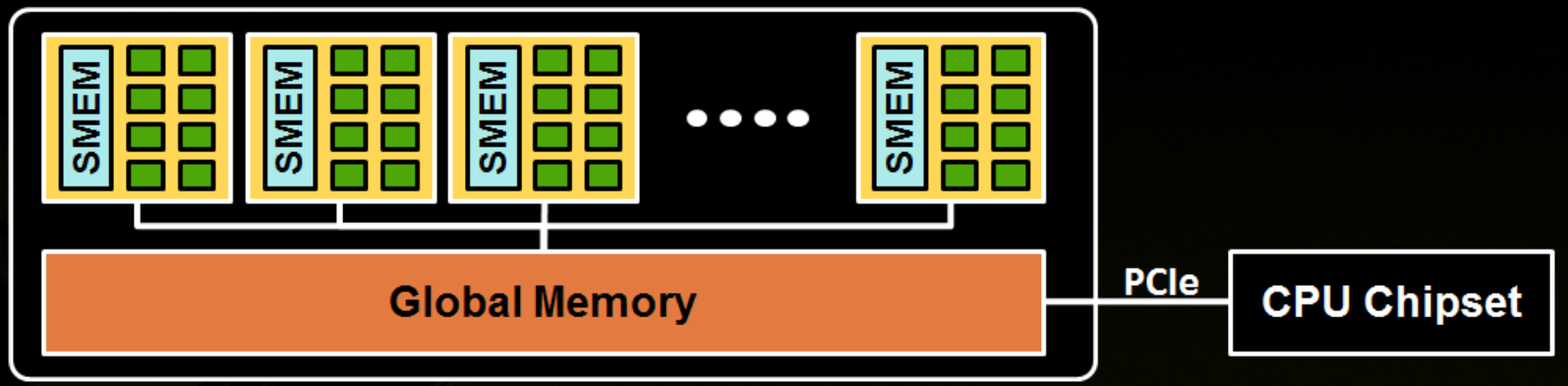
- GPU architecture
- Kernel optimization
  - Memory optimization
  - Latency optimization
  - Instruction optimization
- CPU-GPU interaction optimization
  - Overlapped execution using streams

Optimization Overview

- GPU architecture
- Kernel optimization
  - Memory optimization
  - Execution configuration
  - Instruction optimization
- CPU-GPU interaction optimization
  - Overlapped execution using streams



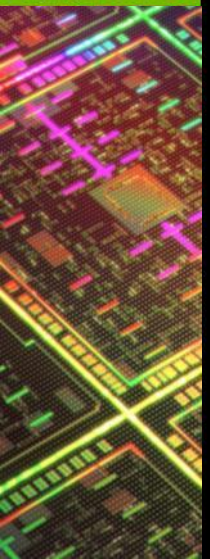
GPU High Level View

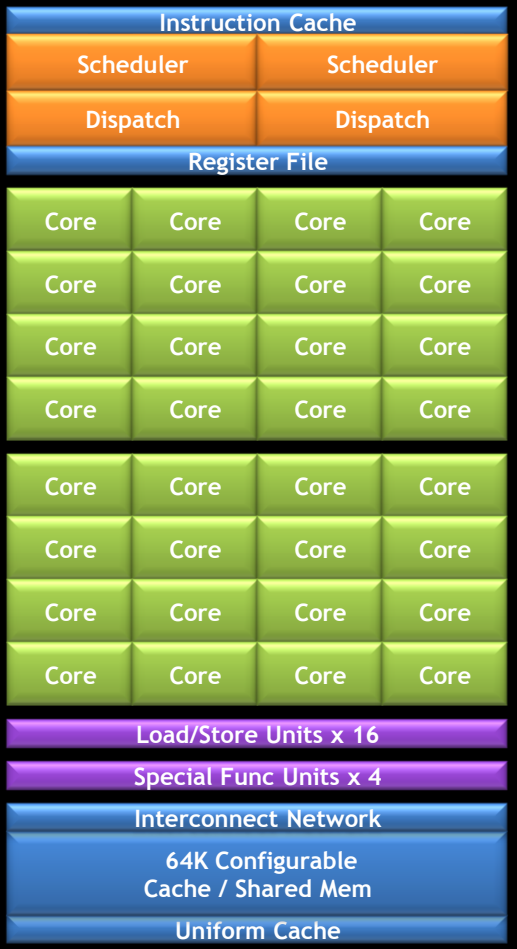


- Streaming Multiprocessor
- Global memory

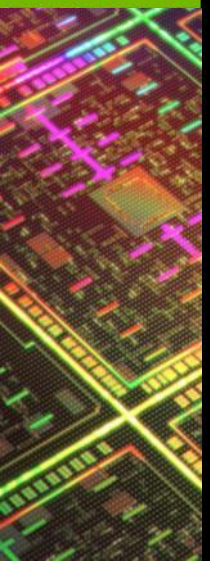
Fermi Multiprocessor

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





GPU and Programming Model



Software

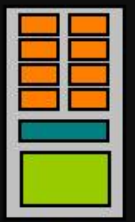
Thread

GPU

CUDA Core

Threads are executed by scalar processors

Thread Block



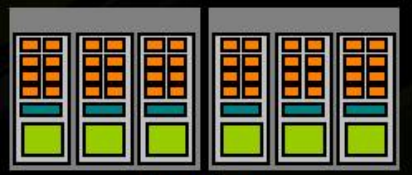
Multiprocessor

Thread blocks are executed on multiprocessors

Thread blocks do not migrate

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources

Grid

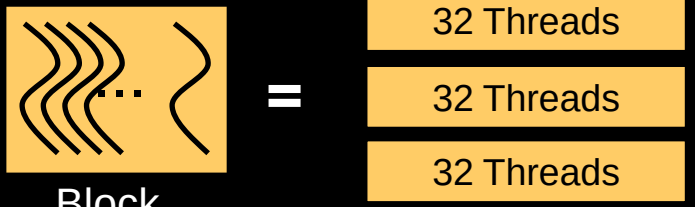


Device

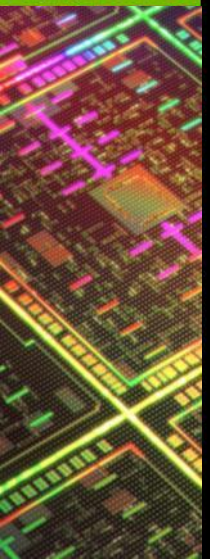
A kernel is launched as a grid of thread blocks

Up to 16 kernels can execute on a device at one time

Warp and SIMT



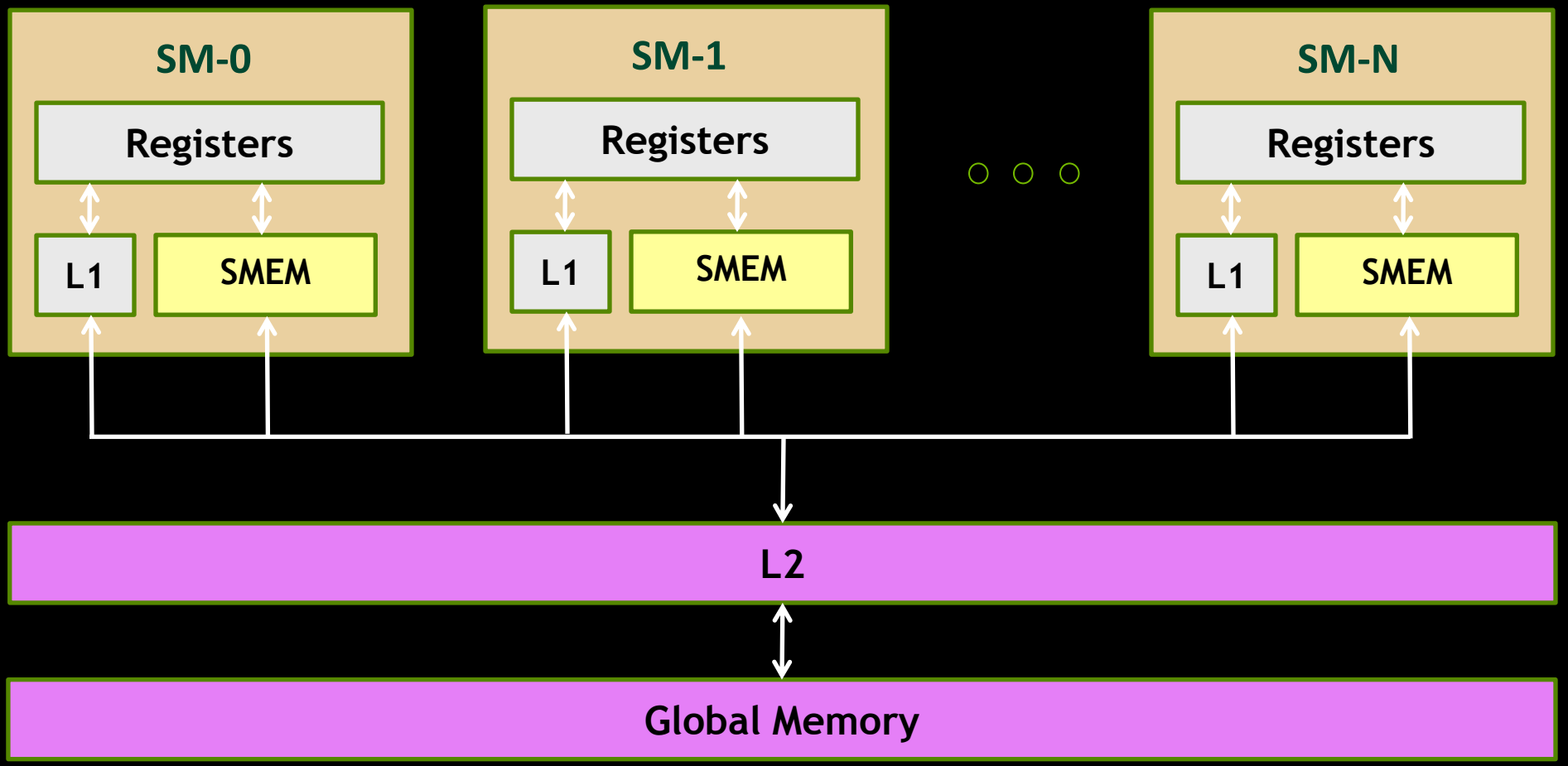
- Blocks divide into groups of 32 threads called warps
- Warps are basic scheduling units
  - Context switching is free
- A lot of warps can hide memory latency
- Warps always perform the same instruction (SIMT)
  - Each thread **CAN** execute its own code path

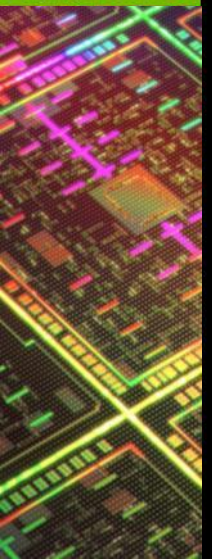


Fermi Memory Hierarchy

- Register
  - Spills to local memory
- Caches
  - Shared memory
  - L1 cache
  - L2 cache
  - Constant cache
  - Texture cache
- Global memory

Fermi Memory Hierarchy Review



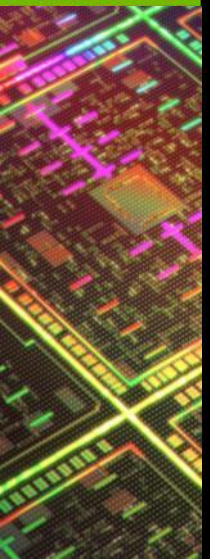


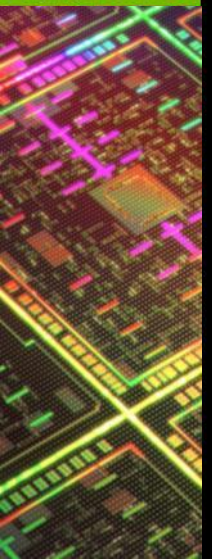
General Optimization Strategies: Measurement

- Find out the limiting factor in kernel performance
  - Memory bandwidth bound (memory optimization)
  - Instruction throughput bound (instruction optimization)
  - Latency bound (configuration optimization)
- Measure effective memory/instruction throughput
- Optimize for peak memory/instruction throughput
  - Finding out the bottleneck
  - Typically an iterative process

Optimization Overview

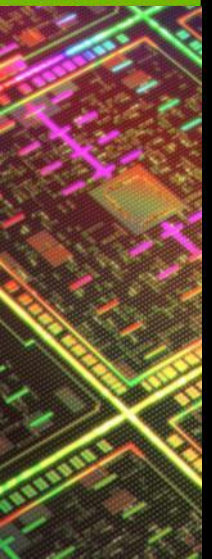
- GPU architecture
- Kernel optimization
  - Memory optimization
  - Latency optimization
  - Instruction optimization
- CPU-GPU interaction optimization
  - Overlapped execution using streams





Memory Optimization

- If the code is memory-bound and effective memory throughput is much lower than the peak
- Purpose: access only data that are absolutely necessary
- Major techniques
  - Improve access pattern to reduce wasted transactions: coalescing
  - Reduce redundant access: shared memory



Coalescing

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

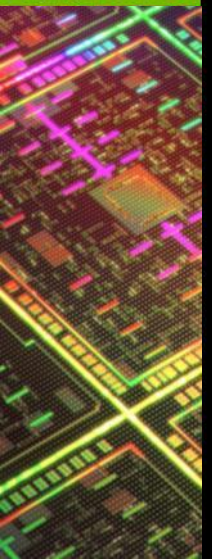
- Coalescing: global memory access from a warp can be coalesced into a single transaction

- Criterion: requests from a warp falling in a L1 cache line, one transaction

 # transaction = # L1 line accessed

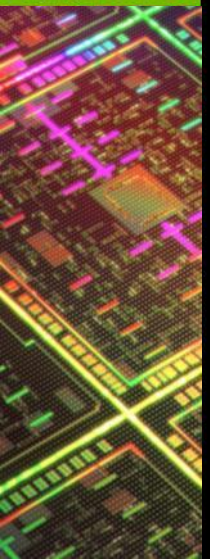
Caching or Non-caching?

- On Fermi, by default all global memory access are cached in L1.
  - L1 can be by-passed by passing “-Xptxas –dlcm=cg” to nvcc: cache only in L2
- If non-cached: same coalescing criterion
  - But transaction size can be reduced to 32B segment



Caching or Non-caching?

- Caching
  - Help on some non-coalesced access, e.g. misaligned
  - May lead to lower performance for some uncoalesced access due to more wasted bandwidth
- Non-caching
  - Reduce wasted bandwidth
  - Leave more space for register spilling



Caching Load

- **Warp requests 32 aligned, consecutive 4-byte words**
- **Addresses fall within 1 cache-line**
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%

Caching Load

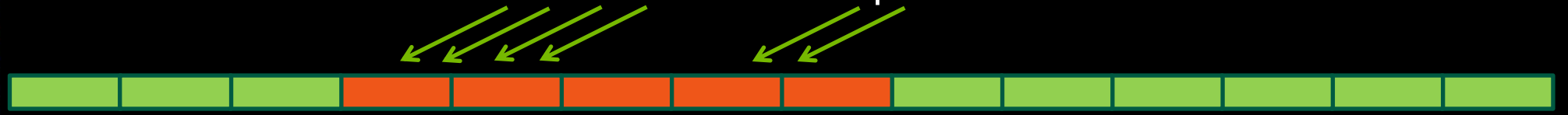
- **Warp requests 32 aligned, permuted 4-byte words**
- **Addresses fall within 1 cache-line**
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%

Caching Load

- **Warp requests 32 misaligned, consecutive 4-byte words**
- **Addresses fall within 2 cache-lines**
  - Warp needs 128 bytes
  - 256 bytes move across the bus on misses
  - Bus utilization: 50%

Non-caching Load

- Warp requests 32 misaligned, consecutive 4-byte words
- Addresses fall within at most 5 segments
  - Warp needs 128 bytes
  - At most 160 bytes move across the bus
  - Bus utilization: at least 80%
    - Some misaligned patterns will fall within 4 segments, so 100% utilization



Caching Load

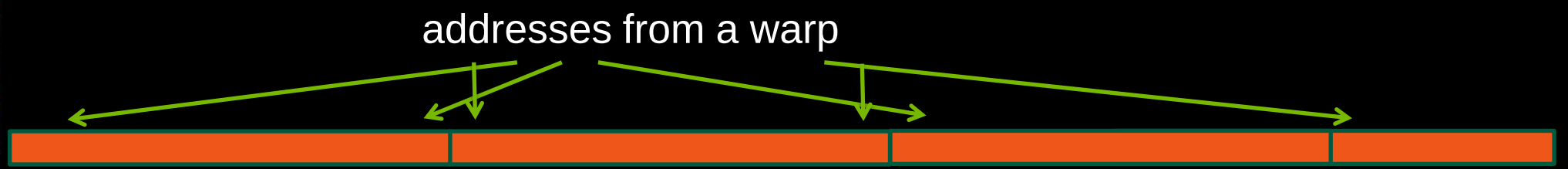
- All threads in a warp request the same 4-byte word
- Addresses fall within a single cache-line
  - Warp needs 4 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 3.125%

Non-caching Load

- All threads in a warp request the same 4-byte word
- Addresses fall within a single segment
  - Warp needs 4 bytes
  - 32 bytes move across the bus on a miss
  - Bus utilization: 12.5%

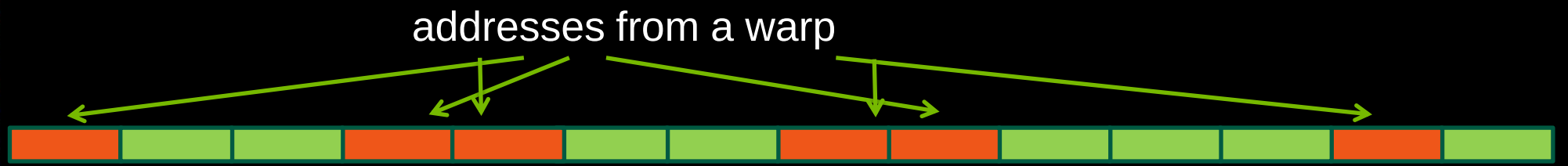
Caching Load

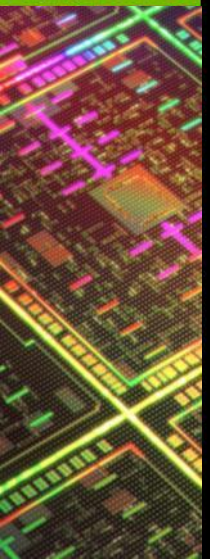
- **Warp requests 32 scattered 4-byte words**
- **Addresses fall within *N* cache-lines**
  - Warp needs 128 bytes
  - *N**128 bytes move across the bus on a miss
  - Bus utilization: 128 / (*N**128)



Non-caching Load

- Warp requests 32 scattered 4-byte words
- Addresses fall within *N* segments
  - Warp needs 128 bytes
  - *N**32 bytes move across the bus on a miss
  - Bus utilization: 128 / (*N**32)



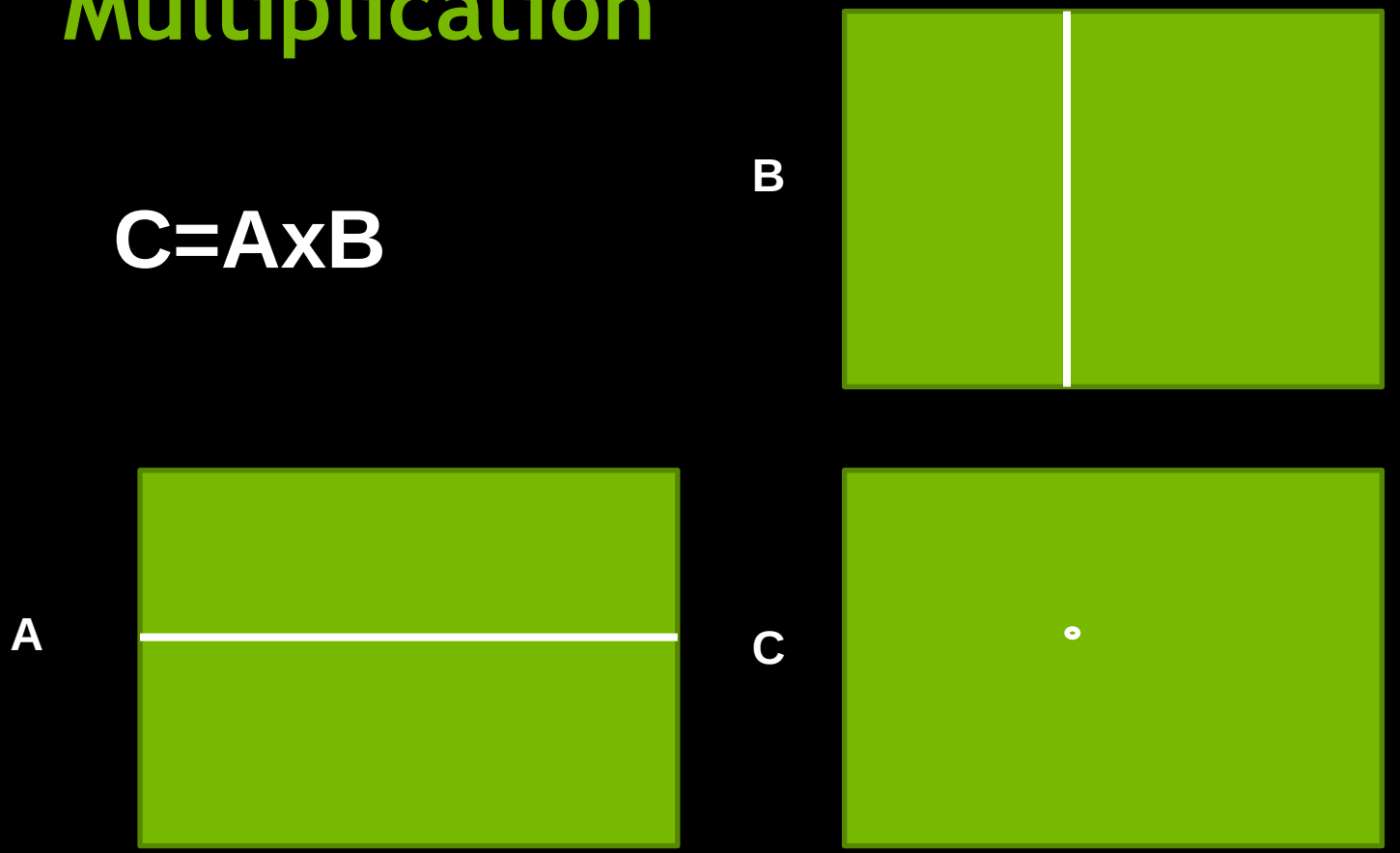


Shared Memory

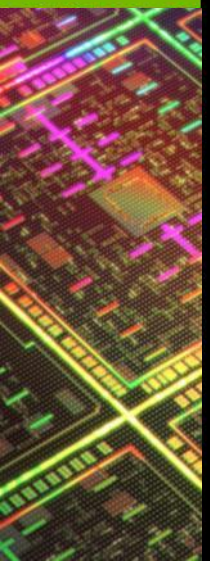
- Low latency: a few cycles
- High throughput: 73.6 GB/s per SM (1.03 TB/s per GPU)
- Main use
  - Inter-block communication
  - User-managed cache to reduce redundant global memory accesses
  - Avoid non-coalesced access

Shared Memory Example: Matrix Multiplication

C=AxB



Every thread corresponds to one entry in C.

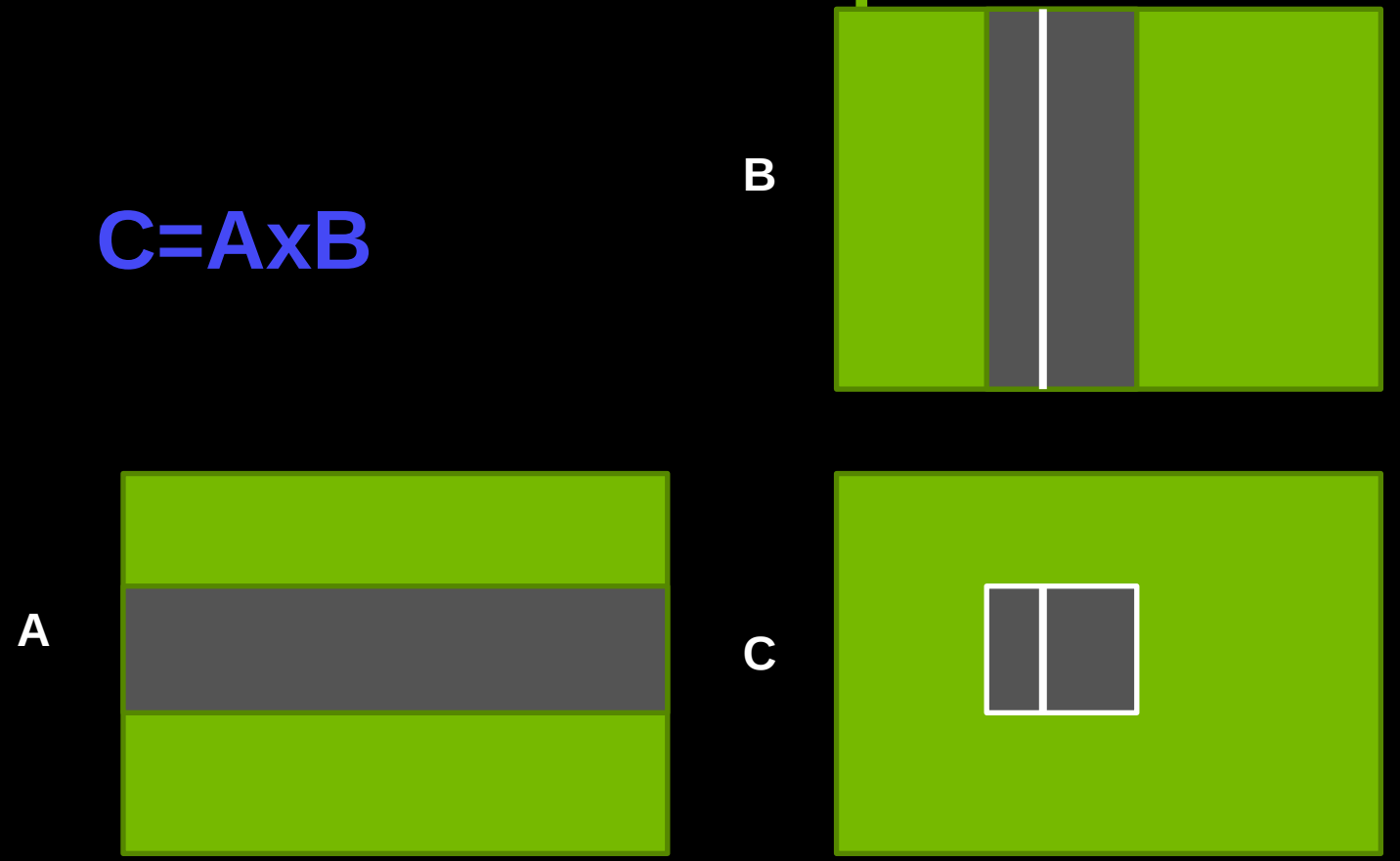


Naive Kernel

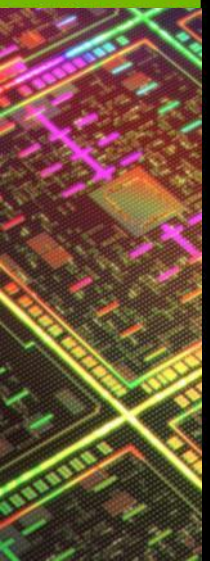
```
__global__ void simpleMultiply(float* a,
                               float* b,
                               float* c,
                               int N)
{
  int row = threadIdx.x + blockIdx.x*blockDim.x;
  int col  = threadIdx.y + blockIdx.y*blockDim.y;
  float sum = 0.0f;
  for (int i = 0; i < N; i++) {
    sum += a[row*N+i] * b[i*N+col];
  }
  c[row*N+col] = sum;
}
```

Blocked Matrix Multiplication

C=AxB



Data reuse in the blocked version



Blocked and cached kernel

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

  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  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];
    __syncthreads();
    for (int i = k; i < k+TILE_DIM; i++)
      sum += aTile[threadIdx.y][i]* bTile[i][threadIdx.x];
  }
  c[row*N+col] = sum;
}
```

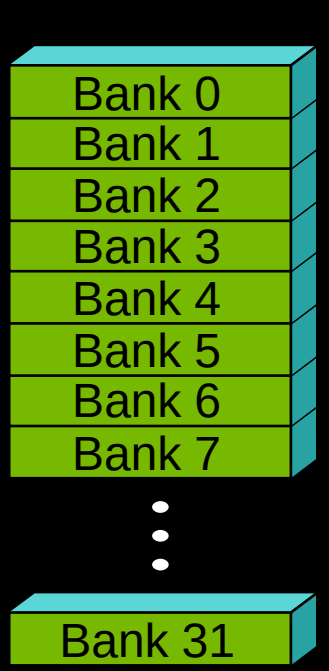
Performance Results

M=N=K=512

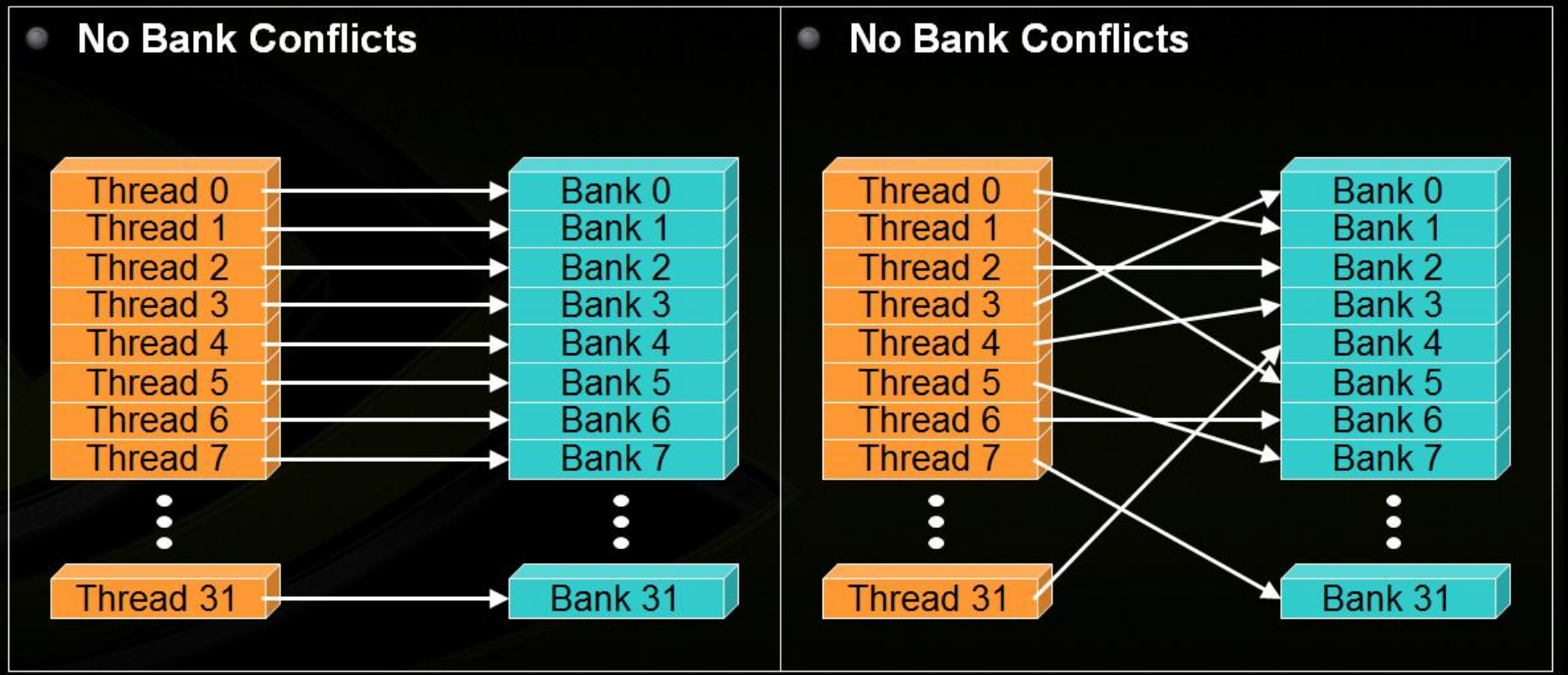
Optimization	C1060	C2050
A, B in global	12 Gflop/s	57 Gflop/s
A, B in shared	125 Gflop/s	181 Gflop/s

Bank Conflicts

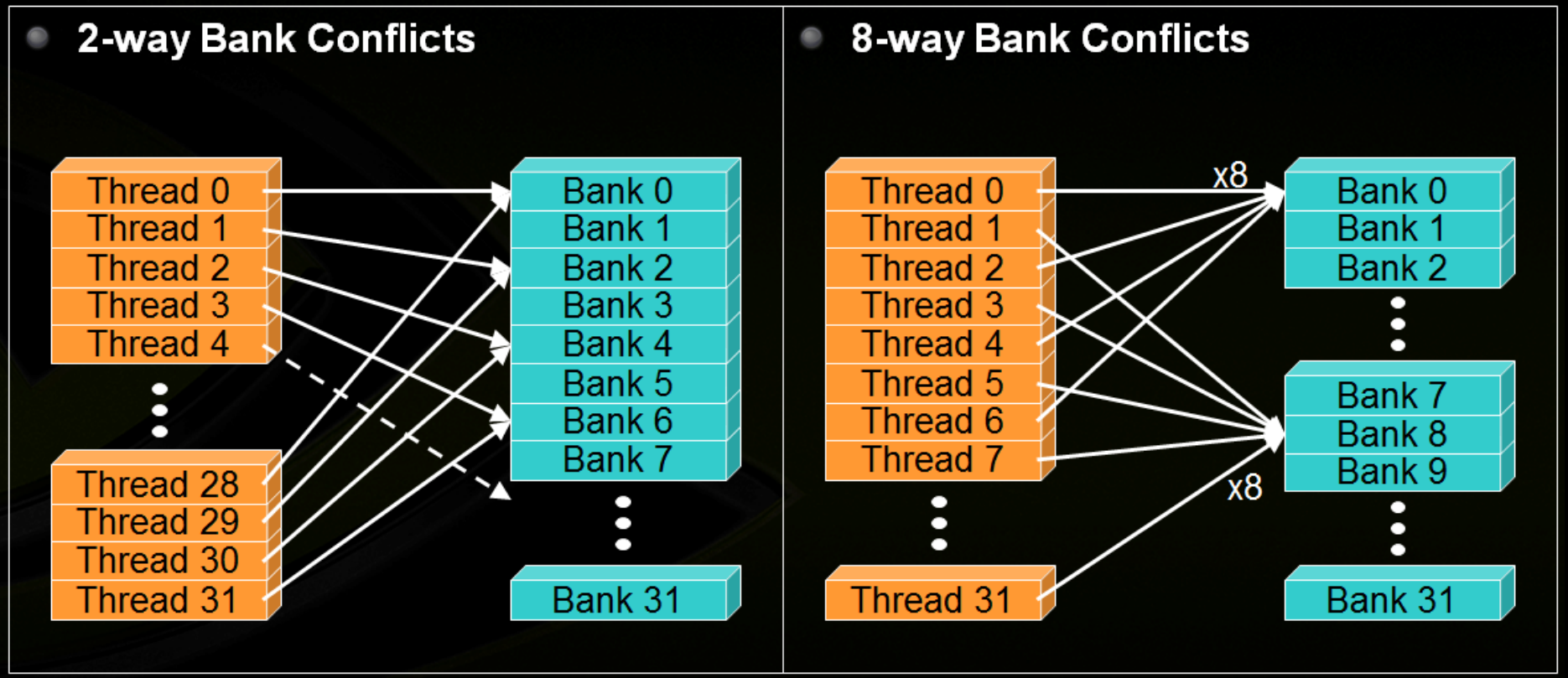
- Shared memory is divided into banks
  - Successive 32-bit words assigned to successive banks
  - Number of banks = 32 (Fermi)
- Bank conflict: two R/W fall in the same bank, the access will be serialized.
- Special cases
  - If all threads in a warp access the same word, one broadcast. Fermi can also do multi-broadcast.
  - If reading continuous byte/double, no conflict on Fermi

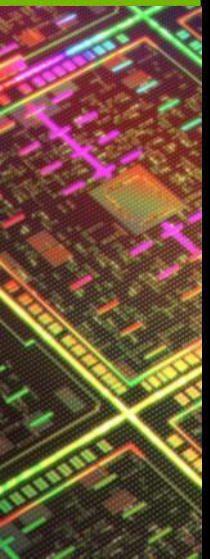


Bank Access Examples



Bank Access Examples



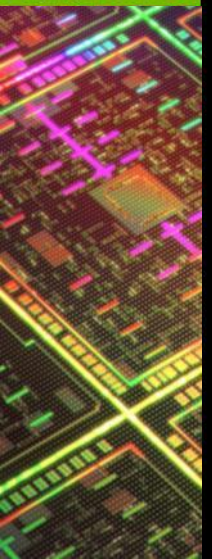


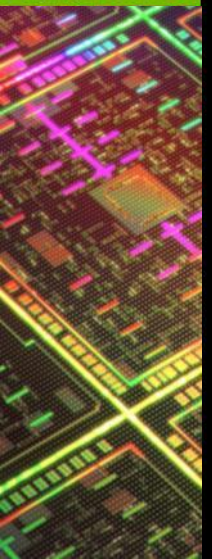
Optimizing Bank Conflict

- Measure whether it matters
  - Change SMEM reads to the same value to see the impact
- Avoiding bank conflict
  - Change address patterns
  - Padding
    - Use array[N_BANK][N_BANK+1]

Memory Optimizations

- Strive for perfect coalescing
  - Transpose the data structure, e.g. AOS to SOA
  - Padding
  - Change parallelization scheme: 1-thread-per-task to 1-warp-per-task?
- Use shared memory to reduce global memory access, avoid non-coalesced access
- Bound to texture cache for unpredictable uncoalesced access
- Use constant cache if all threads in a warp will access the same constant data



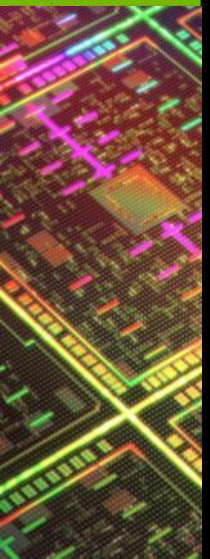


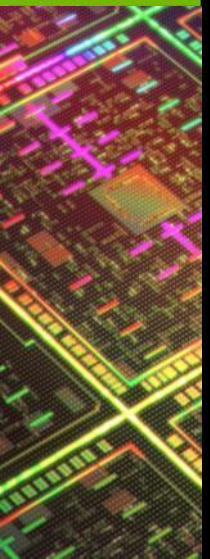
Global Memory Throughput Metric

- 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, shared memory, etc, measuring perf as you go

Optimization Overview

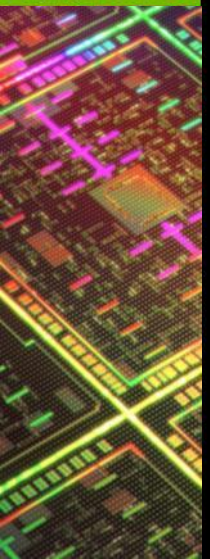
- GPU architecture
- Kernel optimization
  - Memory optimization
  - Latency optimization
  - Instruction optimization
- CPU-GPU interaction optimization
  - Overlapped execution using streams





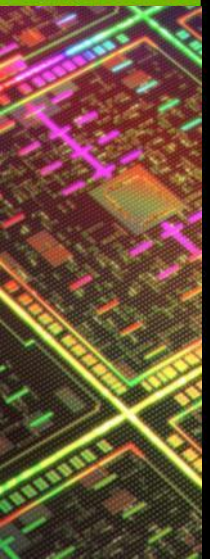
Latency Optimization

- When the code is latency bound
  - Both the memory and instruction throughputs are far from the peak
- Latency hiding:
  - Instructions are issued in order
  - A thread blocks when one of the operands isn't ready
  - Latency is hidden by switching threads
    - GMEM: 400-800 cycles
    - Arithmetic: 18-22 cycles
- Purpose: have enough concurrency to hide latency
- Major techniques: increase concurrency
  - Adjust resource usage to increase active warps (TLP)



Grid/Block Size Heuristics

- # of blocks >> # of SM > 100 to scale well to future device
- Block size should be a multiple of 32 (warp size)
- Minimum: 64. I generally use 128 or 256. But use whatever is best for your app.
- Depends on the problem, do experiments!



Occupancy

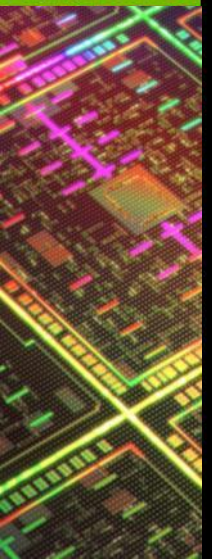
- Occupancy: ratio of active warps per SM to the maximum number of allowed warps
  - Maximum number: 48 in Fermi

- We need the occupancy to be high enough to hide latency

- Occupancy is limited by resource usage

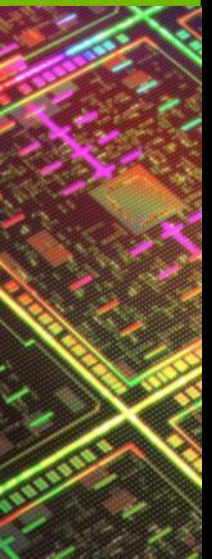
Dynamical Partitioning of SM Resources

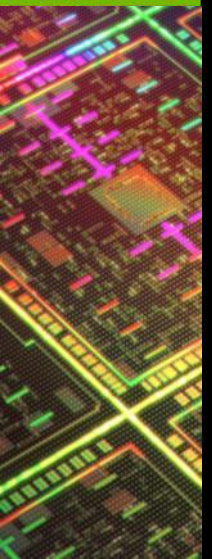
- Shared memory is partitioned among blocks
- Registers are partitioned among threads: <= 63
- Thread block slots: <= 8
- Thread slots: <= 1536
- Any of those can be the limiting factor on how many threads can be launched at the same time on a SM
- If adding a single instruction leads to significant perf drop, occupancy is the primary suspect



Latency Hiding Occupancy Calculation

- Assume global memory takes 400 cycles, we need 400/2 = 200 arithmetic instructions to hide the latency.
- Assume the code has 8 independent arithmetic instructions for every one global memory access. Thus 200/8~26 warps would be enough (54% occupancy).
- Lessons:
  - Required occupancy depends on BOTH architecture and application
  - In this example , beyond 54%, higher occupancy won't lead to further performance increase.

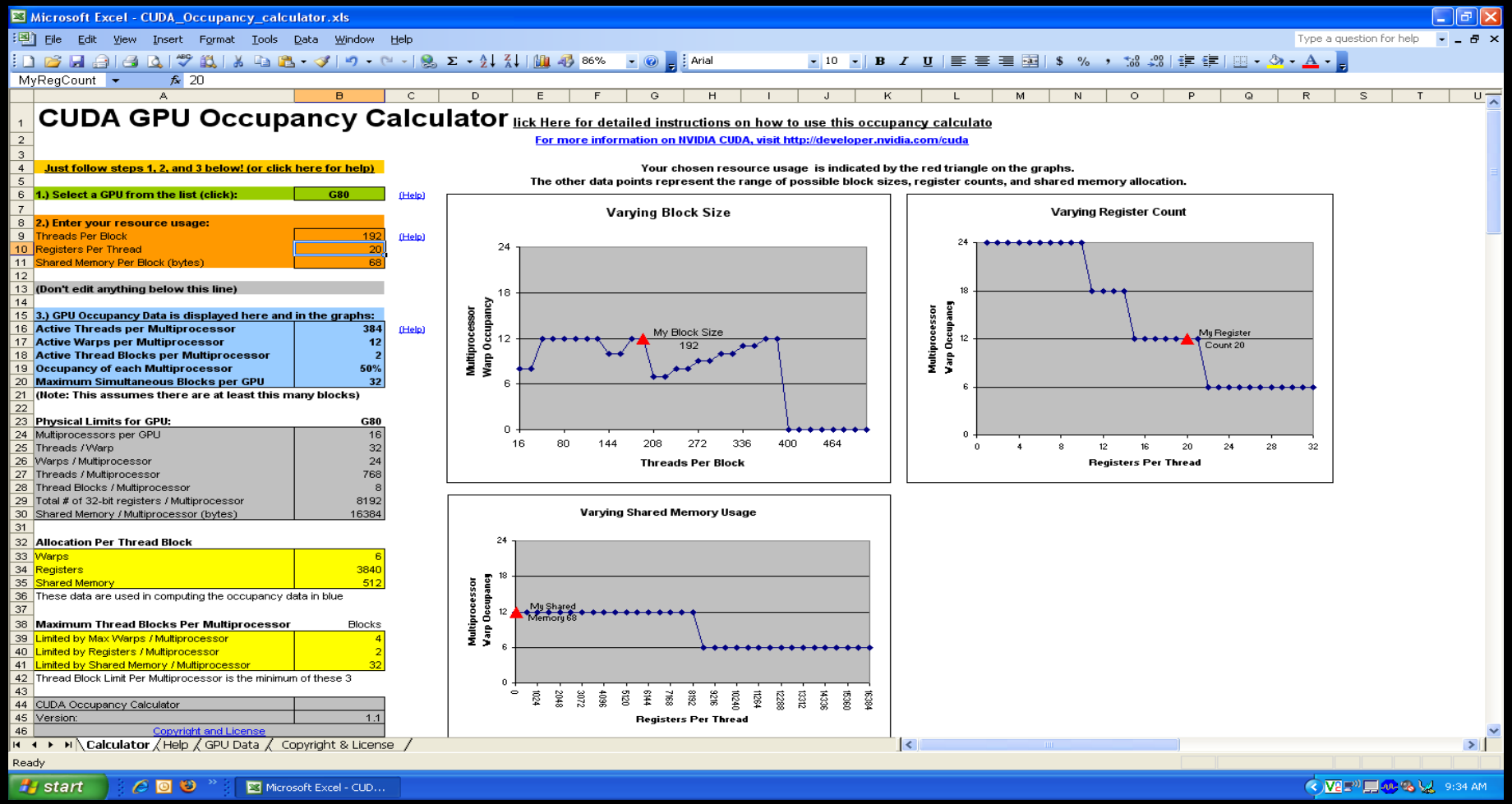




Occupancy Optimizations

- Know the current occupancy
  - Visual profiler
  - --ptxas-options=-v: output resource usage info; input to Occupancy Calculator
- Adjust resource usage to increase occupancy
  - Change block size
  - Limit register usage
    - Compiler option –maxrregcount=n: per file
    - __launch_bounds__: per kernel
    - Use template to reduce register usage
  - Dynamical allocating shared memory

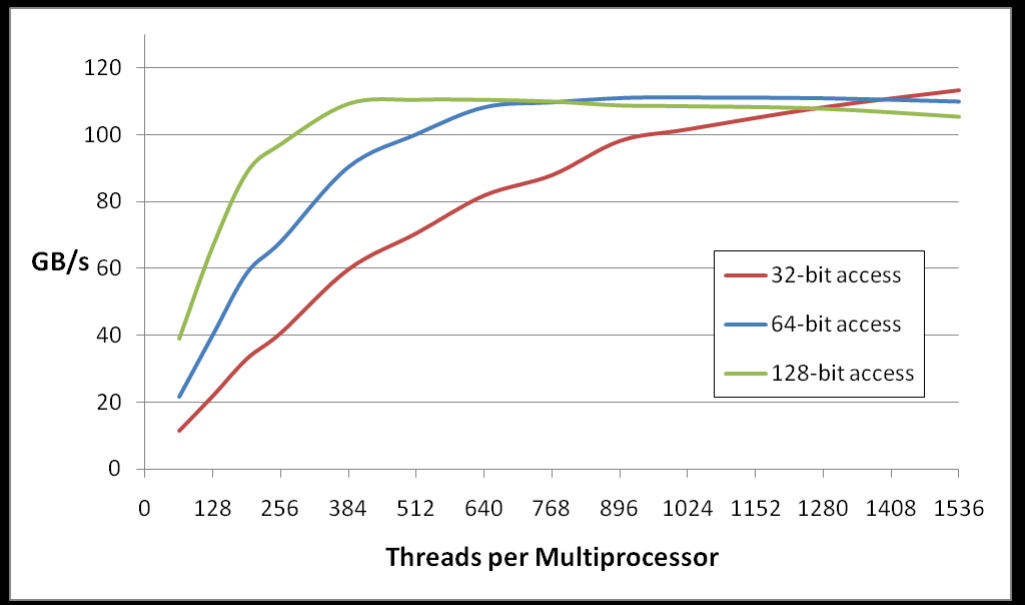
Occupancy Calculator



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

Increase ILP of Each Thread

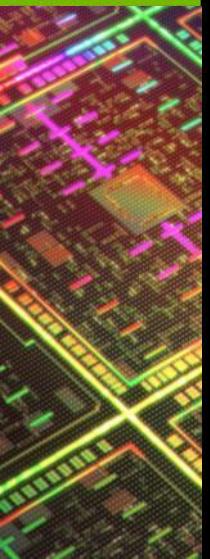
- Load by itself doesn't stall execution
- Increment a 64M element array
  - Two accesses per thread (load then store, but they are dependent)
    - Thus, each warp (32 threads) has one outstanding transaction at a time



Several independent smaller accesses have the same effect as one larger one.

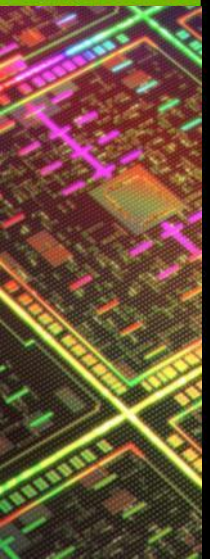
For example:

Four 32-bit ~= one 128-bit



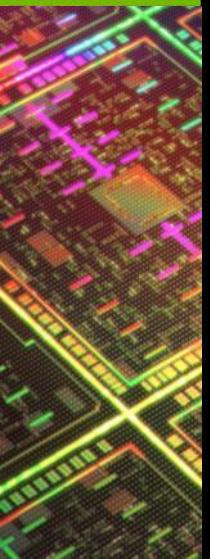
Optimization Overview

- GPU architecture
- Kernel optimization
  - Memory optimization
  - Latency optimization
  - Instruction optimization
- CPU-GPU interaction optimization
  - Overlapped execution using streams



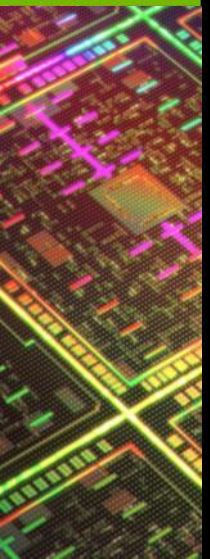
Instruction Optimization

- 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
- Purpose: reduce instruction count
  - Use less instructions to get the same job done
- Major techniques
  - Use high throughput instructions
  - Reduce wasted instructions: branch divergence, bank conflict, etc.



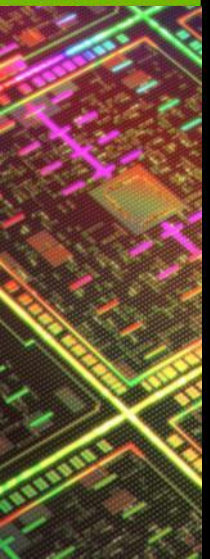
Fermi Arithmetic Instruction Throughputs

- Throughputs of common instructions
  - 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)”



Reduce Instruction Count

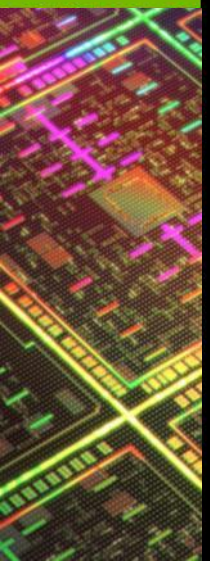
- Avoid automatic conversion of double to float
  - Adding “f” to floating literals (e.g. 1.0f) because the default is double
- Fermi default: -ftz=false, -prec-div=true, -prec-sqrt=true for IEEE compliance
- Fast math functions
  - Two types of runtime math library functions
    - func(): slower but higher accuracy (5 ulp or less)
    - __func(): fast but lower accuracy (see prog. guide for full details)
  - -use_fast_math: forces every func() to __func ()

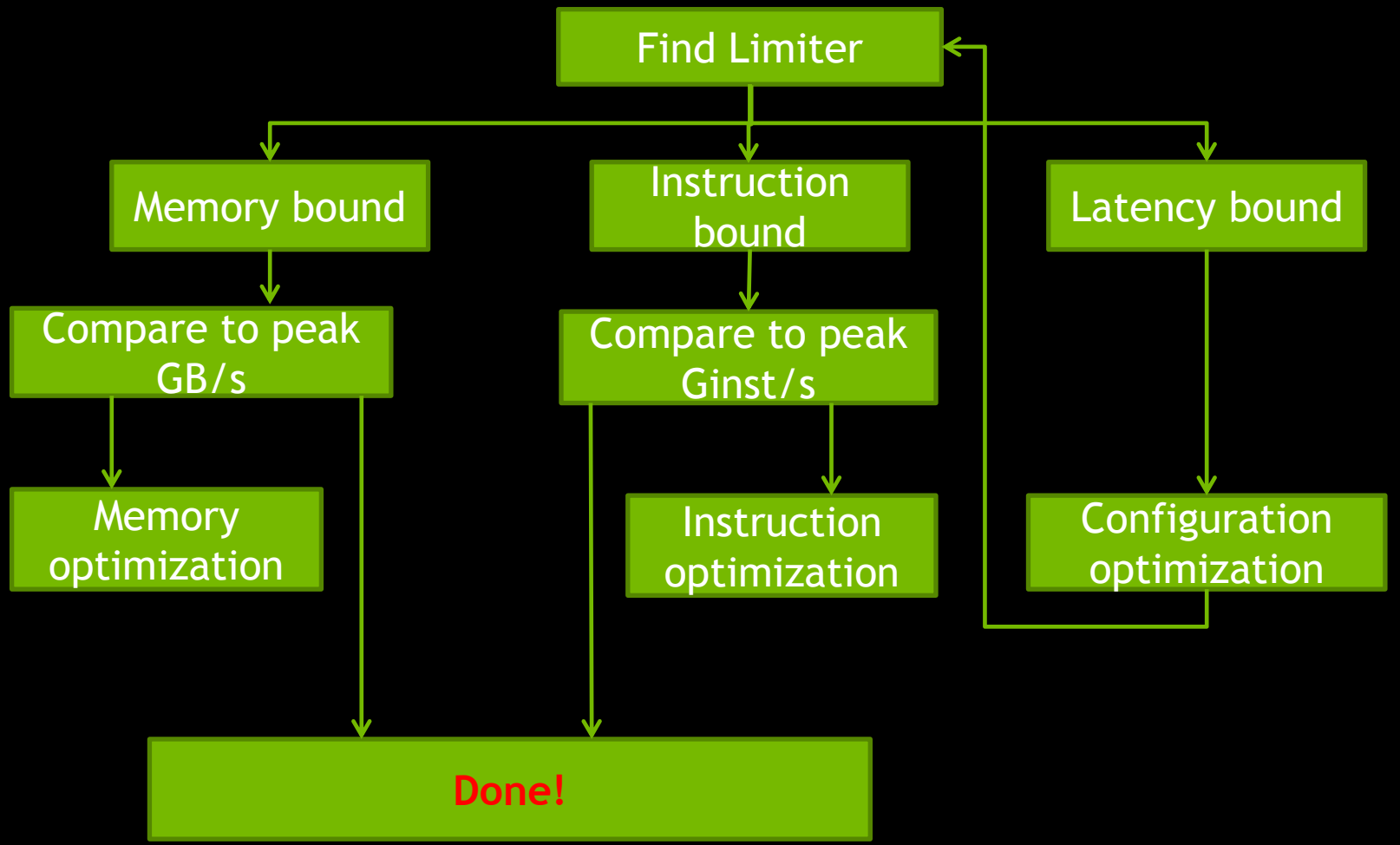


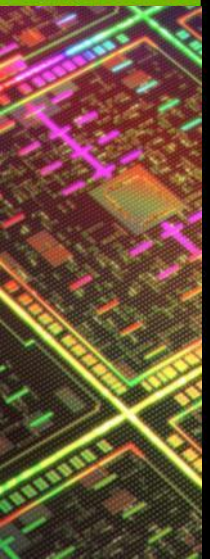
Control Flow

- Divergent branches:
  - Threads within a single warp take different paths
  - Example with divergence:
    - `if (threadIdx.x > 2) {...} else {...}`
    - Branch granularity < warp size
  - Divergence inside a warp is processed by turning off the inactive threads
    - Different if-else branches are both executes: serialized
- Different warps can execute different code with no impact on performance
- Avoid diverging within a warp
  - Example without divergence:
    - `if (threadIdx.x / WARP_SIZE > 2) {...} else {...}`
    - Branch granularity is a whole multiple of warp size

Kernel Optimization Workflow

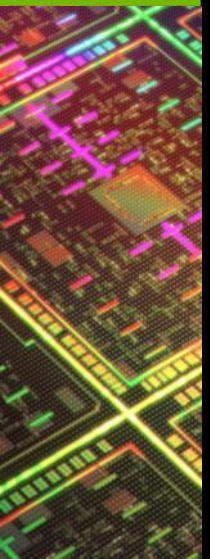






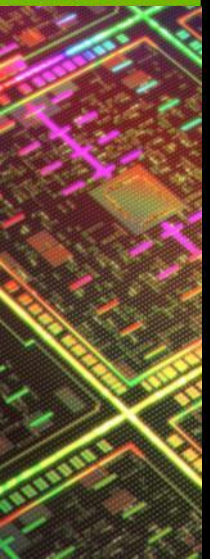
Optimization Overview

- GPU architecture
- Kernel optimization
  - Memory optimization
  - Latency optimization
  - Instruction optimization
- CPU-GPU interaction optimization
  - Overlapped execution using streams



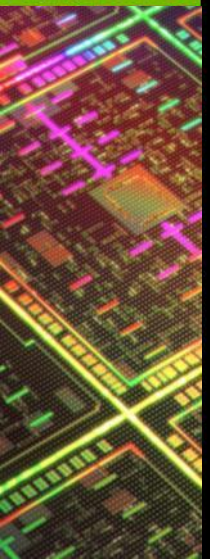
Minimizing CPU-GPU data transfer

- 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 (C2050)
- Minimize transfer
  - Intermediate data directly on GPU
  - Recompute
  - 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



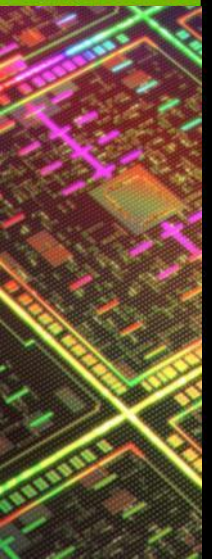
Streams and Async API

- Default API:
  - Kernel launches are asynchronous with CPU
  - Memcopies (D2H, H2D) block CPU thread
  - CUDA calls are serialized by the driver
- Streams and async functions provide:
  - Memcopies (D2H, H2D) asynchronous with CPU
  - Ability to concurrently execute a kernel and a memcopy
  - Concurrent kernel in Fermi
- Stream = sequence of operations that execute in issue-order on GPU
  - Operations from different streams can be interleaved
  - A kernel and memcopy from different streams can be overlapped



Pinned (non-pageable) memory

- Pinned memory enables:
  - memcopies asynchronous with CPU & GPU
- Usage
  - cudaHostAlloc / cudaFreeHost
    - instead of malloc / free
  - Additional flags if pinned region is to be shared between lightweight CPU threads
- Note:
  - pinned memory is essentially removed from virtual memory
  - cudaHostAlloc is typically very expensive

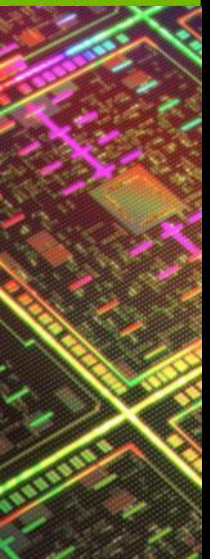


Overlap kernel and memory copy

- Requirements:
  - D2H or H2D memcopy from pinned memory
  - Device with compute capability ≥ 1.1 (G84 and later)
  - Kernel and memcopy in different, non-0 streams
- Code:

```
cudaStream_t   stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync( dst, src, size, dir, stream1 );
kernel<<<grid, block, 0, stream2>>>(…);
```

potentially overlapped



Summary

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