Advanced Topics in CUDA

Cliff Woolley, NVIDIA
Developer Technology Group

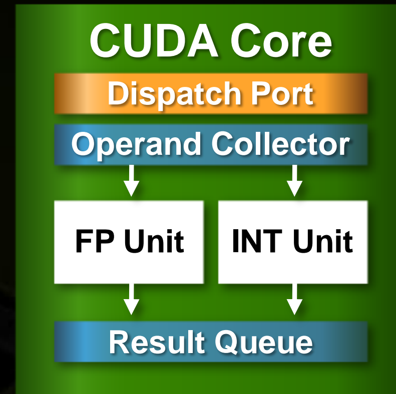
© NVIDIA Corporation 2011

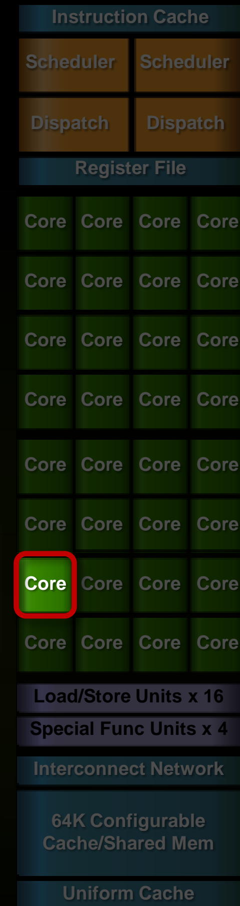
RECAP: SCHEDULING

© NVIDIA Corporation 2011

GPU Architecture – Fermi: CUDA Core

- Floating point & Integer unit
  - IEEE 754-2008 floating-point standard
  - Fused multiply-add (FMA) instruction for both single and double precision
- Logic unit
- Move, compare unit
- Branch unit



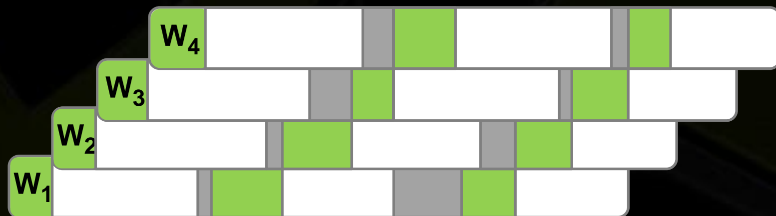


© NVIDIA Corporation 2011

Low Latency or High Throughput?

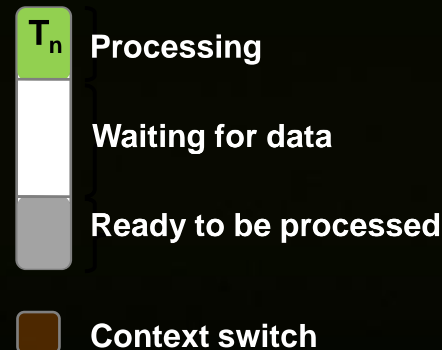
- CPU architecture must minimize latency within each thread
- GPU architecture hides latency with computation from other thread warps

GPU Stream Multiprocessor – High Throughput Processor



CPU core – Low Latency Processor

Computation Thread/Warp



© NVIDIA Corporation 2011

GPU Latency Hiding: Warp Switching

Warp Scheduler	Warp Scheduler
Instruction Dispatch Unit	Instruction Dispatch Unit
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

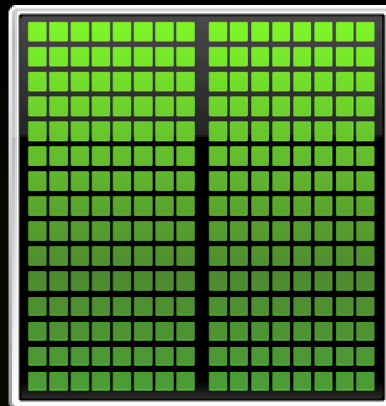
Time

© NVIDIA Corporation 2011

OVERLAPPING PROCESSING WITH DATA TRANSFERS

© NVIDIA Corporation 2011

Available engines

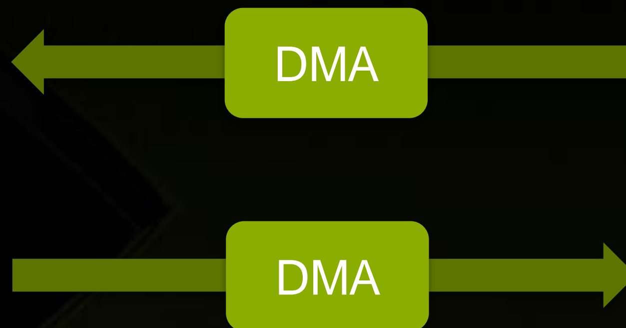


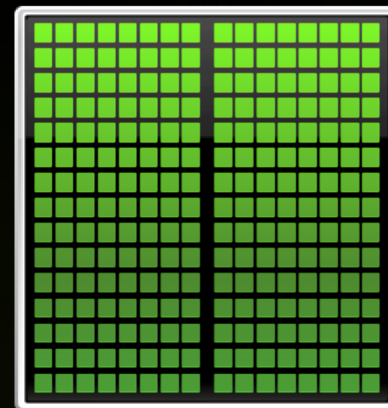
CPU

GPU

© NVIDIA Corporation 2011

Available engines





CPU

GPU

© NVIDIA Corporation 2011

Standard Processing Flow

CPU

Bridge

CPU Memory

PCIe Bus

GigaThread™

Interconnect

L2

DRAM

1. Copy input data from CPU memory to GPU memory

© NVIDIA Corporation 2011

Standard Processing Flow

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

© NVIDIA Corporation 2011

Standard Processing Flow

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

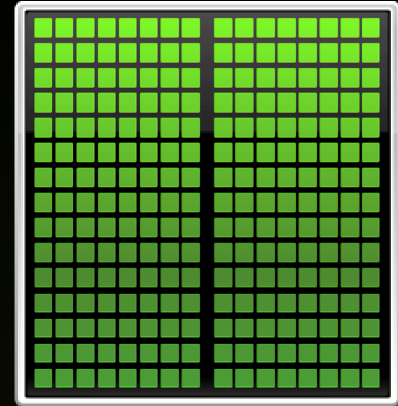
© NVIDIA Corporation 2011

Objective

- Tasks not on the critical path should be hidden
  - i.e. overlapped with other tasks
- Critical path is frequently the transfer of result data from GPU to CPU
  - e.g. path data

© NVIDIA Corporation 2011

Objective: Overlap Processing With Transfers



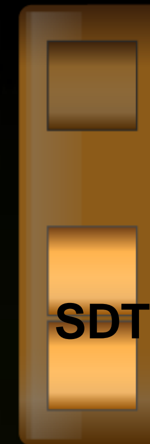
CPU

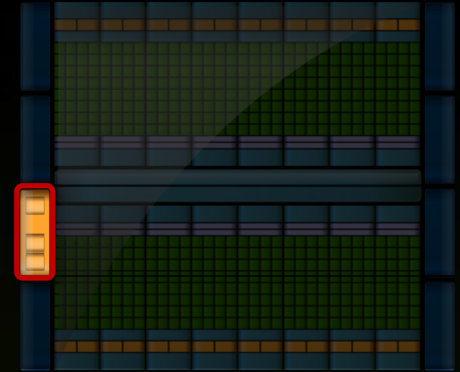
GPU

© NVIDIA Corporation 2011

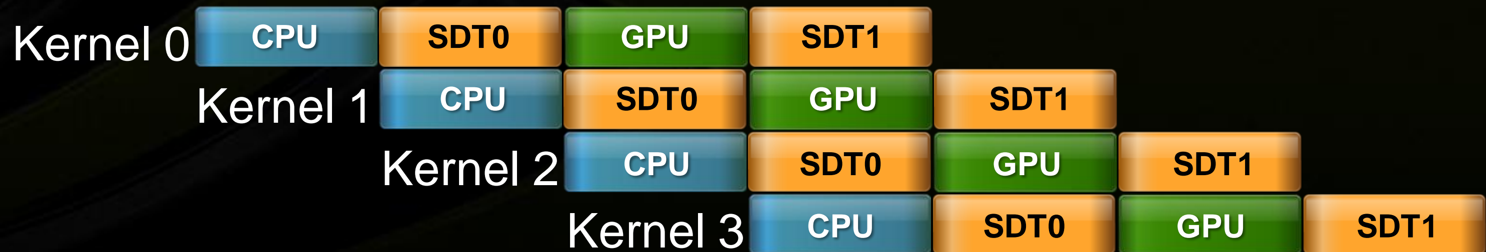
Overlapping Processing With Transfers

- Dual DMA engines
  - Simultaneous CPU→GPU and GPU→CPU data transfer
  - Fully overlapped with CPU and GPU processing time





- Activity Snapshot:



© NVIDIA Corporation 2011

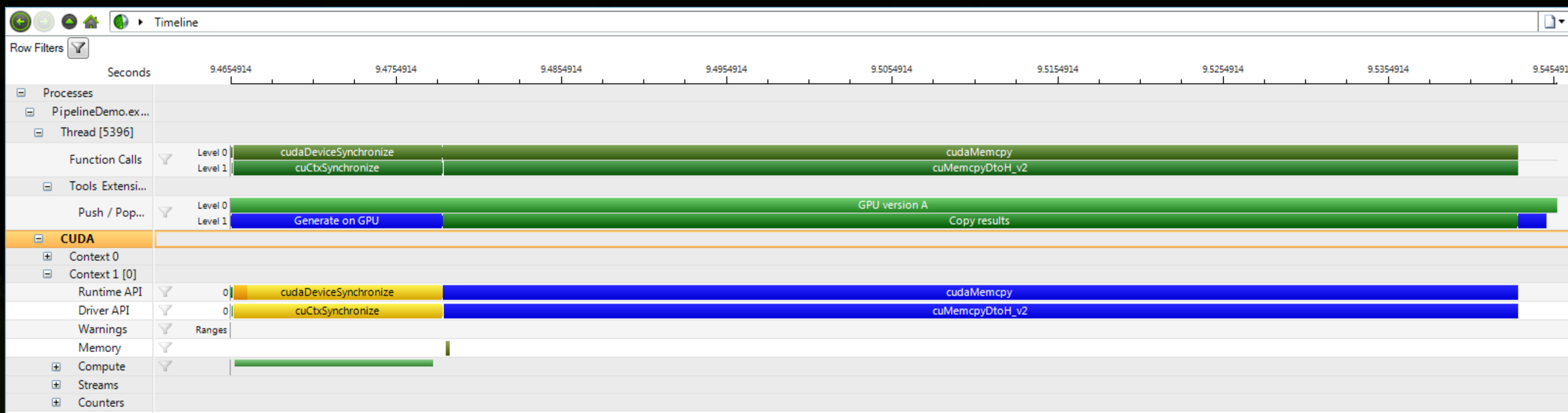
Phase A: No Pipelining

Kernel	Transfer	Post process

- No overlap
  - Kernel runs on GPU
  - Transfer data across PCIe from device to host
  - Post process on CPU

© NVIDIA Corporation 2011

Phase A: No Pipelining



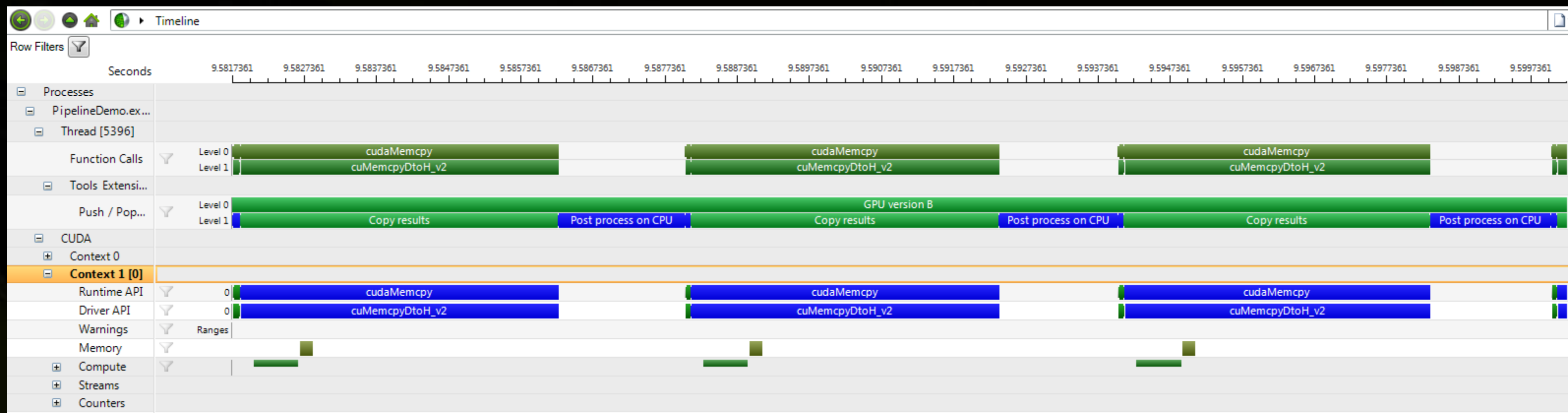
© NVIDIA Corporation 2011

Phase B: Batching

- No overlap
  - Preparation for next phase…

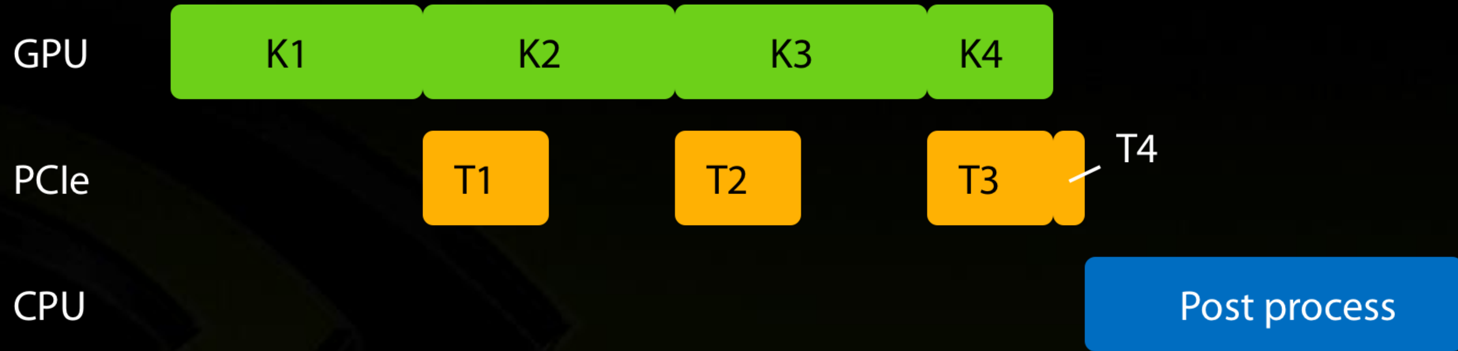
© NVIDIA Corporation 2011

Phase B: Batching



© NVIDIA Corporation 2011

Phase C: Overlap Kernel and Transfer



- Overlap GPU compute with PCIe transfer
- Use streams to specify dependencies
  - K1→T1, K2→T2 etc.

© NVIDIA Corporation 2011

Phase C: Overlap Kernel and Transfer

Timeline
Row Filters
Seconds 9.7402106 9.7403106 9.7404106 9.7405106 9.7406106 9.7407106 9.7408106 9.7409106 9.7410106 9.7411106
Processes
PipelineDemo.ex...
Thread [5396]
Function Calls Level 0: cudaLaunch cudaMem... cudaLaunch cudaMemcpyAs... cudaLaunch cudaMemcpyAs... cudaLaunch cudaMemcpyAs... cudaLaunch cudaMemcpyAs...
Level 1: culaunchKernel cuMem... culaunchKernel cuMemcpyDto... culaunchKernel cuMemcpyDto... culaunchKernel cuMemcpyDto... culaunchKernel cuMemcpyDto...
Tools Extensi...
Push / Pop... Level 0: GPU version C
Level 1: Generate on GPU Copy results Generate on GPU Copy results Generate on GPU Copy results Generate on GPU Copy results Generate on GPU Copy results
CUDA
Context 0
Context 1 [0]
Runtime API 0: cudaLaunch cudaMem... cudaLaunch cudaMemcpyAs... cudaLaunch cudaMemcpyAs... cudaLaunch cudaMemcpyAs... cudaLaunch cudaMemcpyAs...
Driver API 0: culaunchKernel cuMem... culaunchKernel cuMemcpyDto... culaunchKernel cuMemcpyDto... culaunchKernel cuMemcpyDto... culaunchKernel cuMemcpyDto...
Warnings Ranges
Memory 10000KB DeviceToHost
Compute
Streams
Stream 0
Stream 1 kernel
Stream 2 10000KB DeviceToHost
Stream 3
Stream 4
Counters

© NVIDIA Corporation 2011

Phase C: Overlap Kernel and Transfer

Timeline

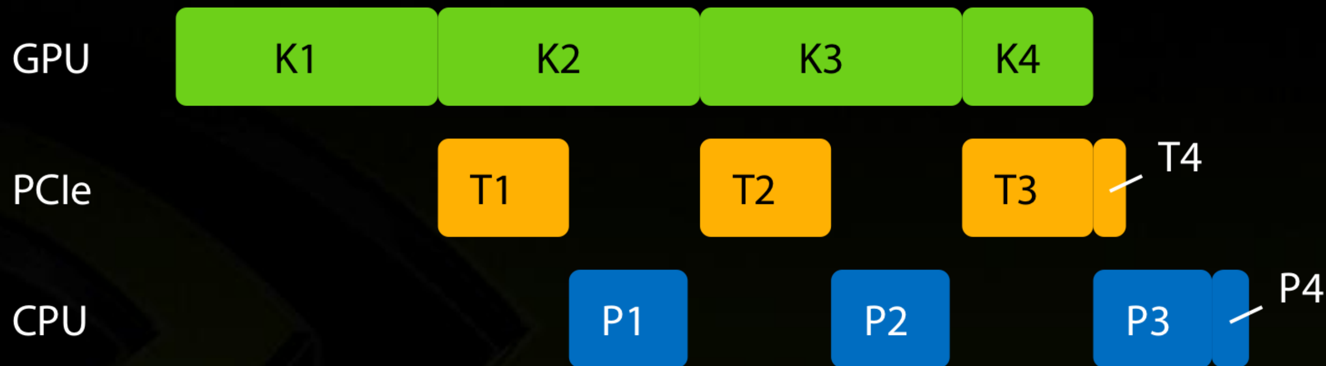
Row Filters

Seconds 9.7371898 9.7391898 9.7411898 9.7431898 9.7451898 9.7471898 9.7491898 9.7511898 9.7531898 9.7551898 9.7571898 9.7591898 9.7611898 9.7631898 9.7651898 9.7671898 9.7691898 9.7711898 9.7731898 9.7751898 9.7771898 9.7791898 9.7811898 9.7831898

Processes
PipelineDemo.ex...
Thread [5396]
Function Calls — Level 0: cudaDeviceSynchronize; Level 1: cuCtxSynchronize
Tools Extensi...
Push / Pop... — Level 0: GPU version C; Level 1: Wait for completion, Post pro...
CUDA
Context 0
Context 1 [0]
Runtime API — 0 cudaDeviceSynchronize
Driver API — 0 cuCtxSynchronize
Warnings — Ranges
Memory — 10000...
Compute
Streams
Stream 0
Stream 1 — 10000...
Stream 2 — 10000...
Stream 3
Stream 4
Counters

© NVIDIA Corporation 2011

Phase D: Overlap Kernel, Transfer and CPU



- Overlap GPU compute, PCIe transfer and CPU compute
- Use streams to specify K/T dependencies
- Use events to specify T/P dependencies
  - T1→P1, T2→P2 etc.

© NVIDIA Corporation 2011

Phase D: Work Dependencies

Compute engine

GPU

PCIe

CPU

Stream order

Copy engine order

Event synchronisation

© NVIDIA Corporation 2011

Phase D: Use Double Buffering

GPU	$\Rightarrow b0_d$	$\Rightarrow b1_d$	$\Rightarrow b0_d$	$\Rightarrow b1_d$		
PCIe		$b0_d \Rightarrow b0_h$	$b1_d \Rightarrow b1_h$	$b0_d \Rightarrow b0_h$	$b1_d \Rightarrow b1_h$	
CPU			$b0_h \Rightarrow$	$b1_h \Rightarrow$	$b0_h \Rightarrow$	$b1_h \Rightarrow$
	iter 0	iter 1	iter 2		stage0_batchBase > numSims	stage1_batchBase > numSims

© NVIDIA Corporation 2011

Phase D: Overlap Kernel, Transfer and CPU

Timeline

Row Filters

Seconds: 9.8420345, 9.8440345, 9.8460345, 9.8480345, 9.8500345, 9.8520345, 9.8540345, 9.8560345, 9.8580345, 9.8600345, 9.8620345, 9.8640345, 9.8660345, 9.8680345, 9.8700345

- Processes
  - PipelineDemo.ex...
    - Thread [5396]
      - Function Calls — Level 0: cudaEventSynchron... ; Level 1: cuEventSynchronize
      - Tools Extensi...
        - Push / Pop... — Level 0: GPU version D ; Level 1: Wait for data, Post pr..., Post proce..., Post process...
    - CUDA
      - Context 0
      - Context 1 [0]
        - Runtime API — 0: cudaEventSynchron...
        - Driver API — 0: cuEventSynchronize
        - Warnings — Ranges
        - Memory — 10000KB D...
        - Compute
        - Streams
          - Stream 0
          - Stream 1
          - Stream 2
          - Stream 3 — kernel, 10000KB D...
          - Stream 4 — kernel, 10000KB D...
        - Counters

© NVIDIA Corporation 2011

Phase D: Overlap Kernel, Transfer and CPU

Timeline

Row Filters

Seconds

9.8520277 9.8521277 9.8522277 9.8523277 9.8524277 9.8525277 9.8526277 9.8527277 9.8528277 9.8529277 9.8530277 9.8531277 9.8532277 9.8533277 9.8534277 9.8535277 9.8536277 9.8537277 9.8538277 9.8539277 9.8540277

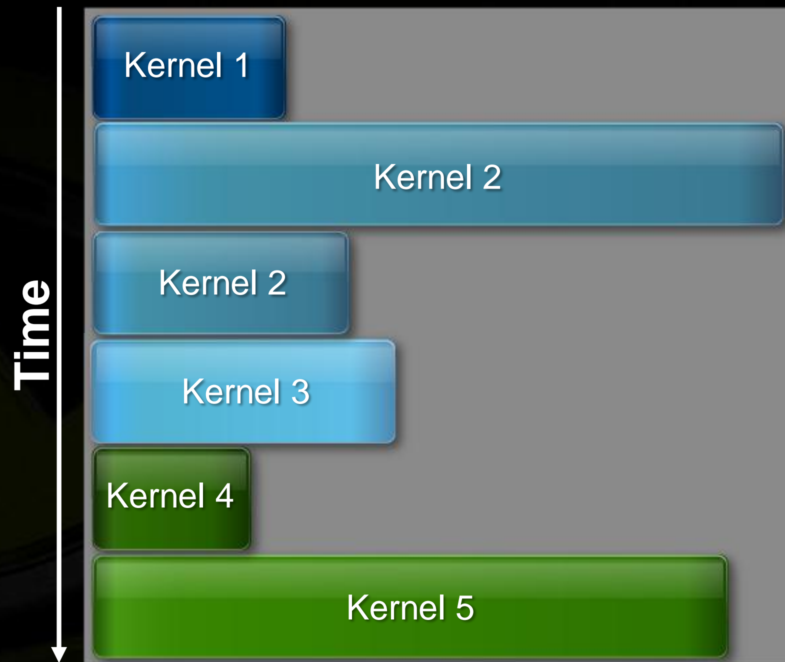
Processes
PipelineDemo.ex...
Thread [5396]
Function Calls — Level 0: cudaLaun... — Level 1: cuLaunc...
Tools Extensi...
Push / Pop... — Level 0: GPU version D — Level 1: Generate on... Wait for... Post process on CPU Gener...
CUDA
Context 0
Context 1 [0]
Runtime API — 0 — cudaLaun...
Driver API — 0 — cuLaunc...
Warnings — Ranges
Memory — 10000KB DeviceToHost
Compute
Streams
Stream 0
Stream 1
Stream 2
Stream 3 — kernel
Stream 4 — 10000KB DeviceToHost
Counters

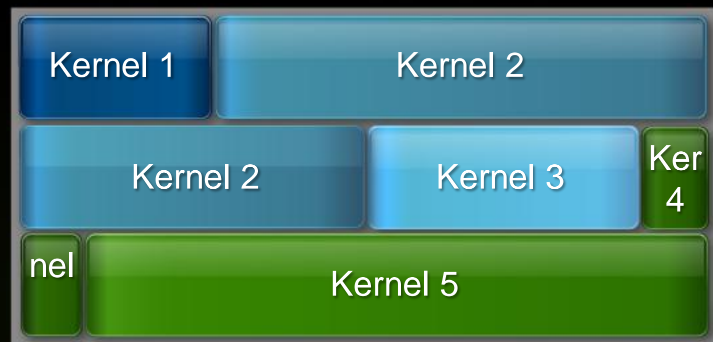
© NVIDIA Corporation 2011

CONCURRENT KERNELS

© NVIDIA Corporation 2011

Concurrent Kernel Execution





Sequential Kernel Execution

Parallel Kernel Execution

© NVIDIA Corporation 2011

ANALYSIS-DRIVEN OPTIMIZATION

© NVIDIA Corporation 2011

Performance Optimization Process

- **Use appropriate performance metric for each kernel**
  - For example, Gflops/s don’t make sense for a bandwidth-bound kernel
- **Determine what limits kernel performance**
  - Memory throughput
  - Instruction throughput
  - Latency
  - Combination of the above
- **Address the limiters in the order of importance**
  - Determine how close to the HW limits the resource is being used
  - Analyze for possible inefficiencies
  - Apply optimizations
    - Often these will just fall out from how HW operates

© NVIDIA Corporation 2011

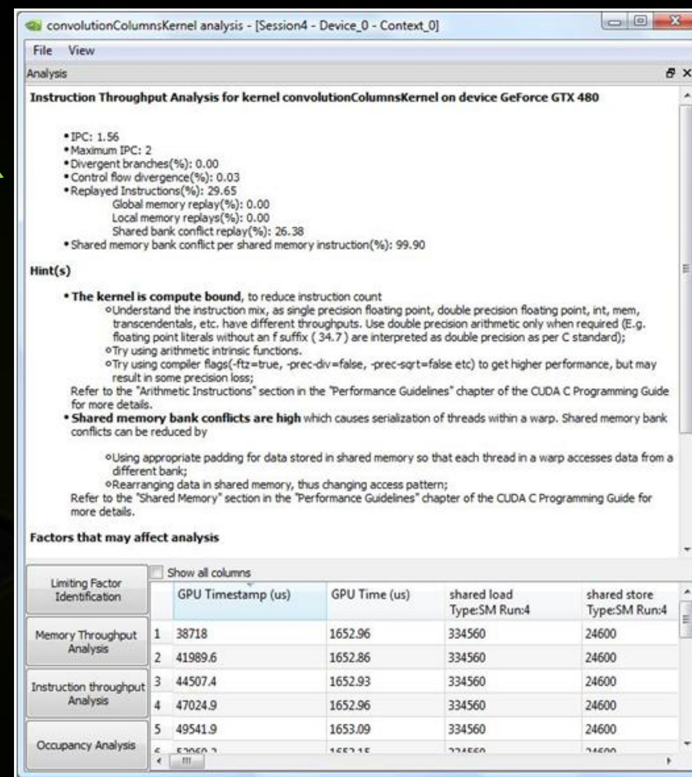
New in CUDA Toolkit 4.0: Automated Performance Analysis using Visual Profiler

Summary analysis & hints

- Per-Session
- Per-Device
- Per-Context
- Per-Kernel

New UI for kernel analysis

- Identify the limiting factor
- Analyze instruction throughput
- Analyze memory throughput
- Analyze kernel occupancy



© NVIDIA Corporation 2011

Notes on profiler

- Most counters are reported per Streaming Multiprocessor (SM)
  - Not entire GPU
  - Exceptions: L2 and DRAM counters
- A single run can collect a few counters
  - Multiple runs are needed when profiling more counters
    - Done automatically by the Visual Profiler
    - Have to be done manually using command-line profiler
- Counter values may not be exactly the same for repeated runs
  - Threadblocks and warps are scheduled at run-time
  - So, “two counters being equal” usually means “two counters within a small delta”
- See the profiler documentation for more information

© NVIDIA Corporation 2011

ANALYSIS-DRIVEN OPTIMIZATION: IDENTIFYING PERF LIMITERS

© NVIDIA Corporation 2011

Limited by Bandwidth or Arithmetic?

- **Perfect instructions:bytes ratio for Fermi C2050:**
  - ~4.5 : 1 with ECC on
  - ~3.6 : 1 with ECC off
  - These assume fp32 instructions, throughput for other instructions varies
- **Algorithmic analysis:**
  - Rough estimate of arithmetic to bytes ratio
- **Code likely uses more instructions and bytes than algorithm analysis suggests:**
  - Instructions for loop control, pointer math, etc.
  - Address pattern may result in more memory fetches
  - Two ways to investigate:
    - Use the profiler (quick, but approximate)
    - Use source code modification (more accurate, more work intensive)

© NVIDIA Corporation 2011

A Note on Counting Global Memory Accesses

- **Load/store instruction count can be lower than the number of actual memory transactions**
  - Address pattern, different word sizes
- **Counting requests from L1 to the rest of the memory system makes the most sense**
  - Caching-loads: count L1 misses
  - Non-caching loads and stores: count L2 read requests
    - Note that L2 counters are for the entire chip, L1 counters are per SM
- **Some shortcuts, assuming "coalesced" address patterns:**
  - One 32-bit access instruction -> one 128-byte transaction per warp
  - One 64-bit access instruction -> two 128-byte transactions per warp
  - One 128-bit access instruction -> four 128-byte transactions per warp

© NVIDIA Corporation 2011

Analysis with Profiler

- **Profiler counters:**
  - instructions_issued, instructions_executed
    - Both incremented by 1 per warp
    - "issued" includes replays, "executed" does not
  - gld_request, gst_request
    - Incremented by 1 per warp for each load/store instruction
    - Instruction may be counted if it is "predicated out"
  - l1_global_load_miss, l1_global_load_hit, global_store_transaction
    - Incremented by 1 per **L1 line** (line is 128B)
  - uncached_global_load_transaction
    - Incremented by 1 per group of 1, 2, 3, or 4 transactions
    - Better to look at L2_read_request counter (incremented by 1 per 32B transaction; **per GPU, not per SM**)
- **Compare:**
  - 32 * instructions_issued /* 32 = warp size */
  - 128B * (global_store_transaction + l1_global_load_miss)

© NVIDIA Corporation 2011

Analysis with Modified Source Code

- **Time memory-only and math-only versions of the kernel**
  - Easier for codes that don't have data-dependent control-flow or addressing
  - Gives you good estimates for:
    - Time spent accessing memory
    - Time spent in executing instructions
- **Compare the times taken by the modified kernels**
  - Helps decide whether the kernel is mem or math bound
- **Compare the sum of mem-only and math-only times to full-kernel time**
  - Shows how well memory operations are overlapped with arithmetic
  - Can reveal latency bottleneck

© NVIDIA Corporation 2011

Some Example Scenarios

time

mem	math	full

Memory-bound

Good mem-math overlap: latency likely not a problem

(assuming memory throughput is not low compared to HW theory)

mem	math	full

Math-bound

Good mem-math overlap: latency likely not a problem

(assuming instruction throughput is not low compared to HW theory)

mem	math	full

Balanced

Good mem-math overlap: latency likely not a problem

(assuming memory/instr throughput is not low compared to HW theory)

mem	math	full

Memory and latency bound

Poor mem-math overlap: latency *is* a problem

© NVIDIA Corporation 2011

Source Modification

- **Memory-only:**
  - Remove as much arithmetic as possible
    - Without changing access pattern
    - Use the profiler to verify that load/store instruction count is the same
- **Store-only:**
  - Also remove the loads (to compare read time vs. write time)
- **Math-only:**
  - Remove global memory accesses
  - Need to trick the compiler:
    - Compiler throws away all code that it detects as not contributing to stores
    - Put stores inside conditionals that always evaluate to `false`
      - Condition should depend on the value about to be stored (prevents other optimizations)
      - Condition outcome should not be known to the compiler

© NVIDIA Corporation 2011

Source Modification for Math-only

```
__global__ void fwd_3D( ..., int flag)
{
        ...
    value = temp + coeff * vsq;
    if( 1 == value * flag )
        g_output[out_idx] = value;
}
```

If you compare only the flag, the compiler may move the computation into the conditional as well

© NVIDIA Corporation 2011

Source Modification and Occupancy

- **Removing pieces of code is likely to affect register count**
  - This could increase occupancy, skewing the results
  - See slide 23 to see how that could affect throughput
- **Make sure to keep the same occupancy**
  - Check the occupancy with profiler before modifications
  - After modifications, if necessary add shared memory to match the unmodified kernel's occupancy

```
kernel<<< grid, block, smem, ...>>>(...)
```

© NVIDIA Corporation 2011

Case Study: Limiter Analysis

- 3DFD of the wave equation, fp32
- Time (ms):
  - Full-kernel: 35.39
  - Mem-only: 33.27
  - Math-only: 16.25
- Instructions issued:
  - Full-kernel: 18,194,139
  - Mem-only: 7,497,296
  - Math-only: 16,839,792
- Memory access transactions:
  - Full-kernel: 1,708,032
  - Mem-only: 1,708,032
  - Math-only: 0

- **Analysis:**
  - Instr:byte ratio = ~2.66
    - 32*18,194,139 / 128*1,708,032
  - Good overlap between math and mem:
    - 2.12 ms of math-only time (13%) are not overlapped with mem
  - App memory throughput: 62 GB/s
    - HW theory is 114 GB/s, so we're off
- **Conclusion:**
  - Code is memory-bound
  - Latency could be an issue too
  - Optimizations should focus on memory throughput first
    - math contributes very little to total time (2.12 out of 35.39ms)

© NVIDIA Corporation 2011

Summary: Limiter Analysis

- **Rough algorithmic analysis:**
  - How many bytes needed, how many instructions
- **Profiler analysis:**
  - Instruction count, memory request/transaction count
- **Analysis with source modification:**
  - Memory-only version of the kernel
  - Math-only version of the kernel
  - Examine how these times relate and overlap

© NVIDIA Corporation 2011

Questions?

© NVIDIA Corporation 2011