

Introduction to CUDA Programming

Lecture 6: Profiling & Tuning Applications

高性能计算机研究中心

Introduction

- **Why is my application running slow?**
- **Work it out on paper**
- **Instrument code**
- **Profile it**
 - NVIDIA Visual Profiler
 - Works with CUDA, needs some tweaks to work with OpenCL
 - nvprof – command line tool, can be used with MPI applications

Identifying Performance Limiters

- **CPU: Setup , data movement**
- **GPU: Bandwidth, compute or latency limited**
- **Number of instructions for every byte moved**
 - ~3.6 : 1 on Fermi
 - ~6.4 : 1 on Kepler
- **Algorithmic analysis gives a good estimate**
- **Actual code is likely different**
 - Instructions for loop control, pointer math, etc.
 - Memory access patterns
 - How to find out?
 - Use the profiler (quick, but approximate)
 - Use source code modification (takes more work)

Analysis with Source Code Modification

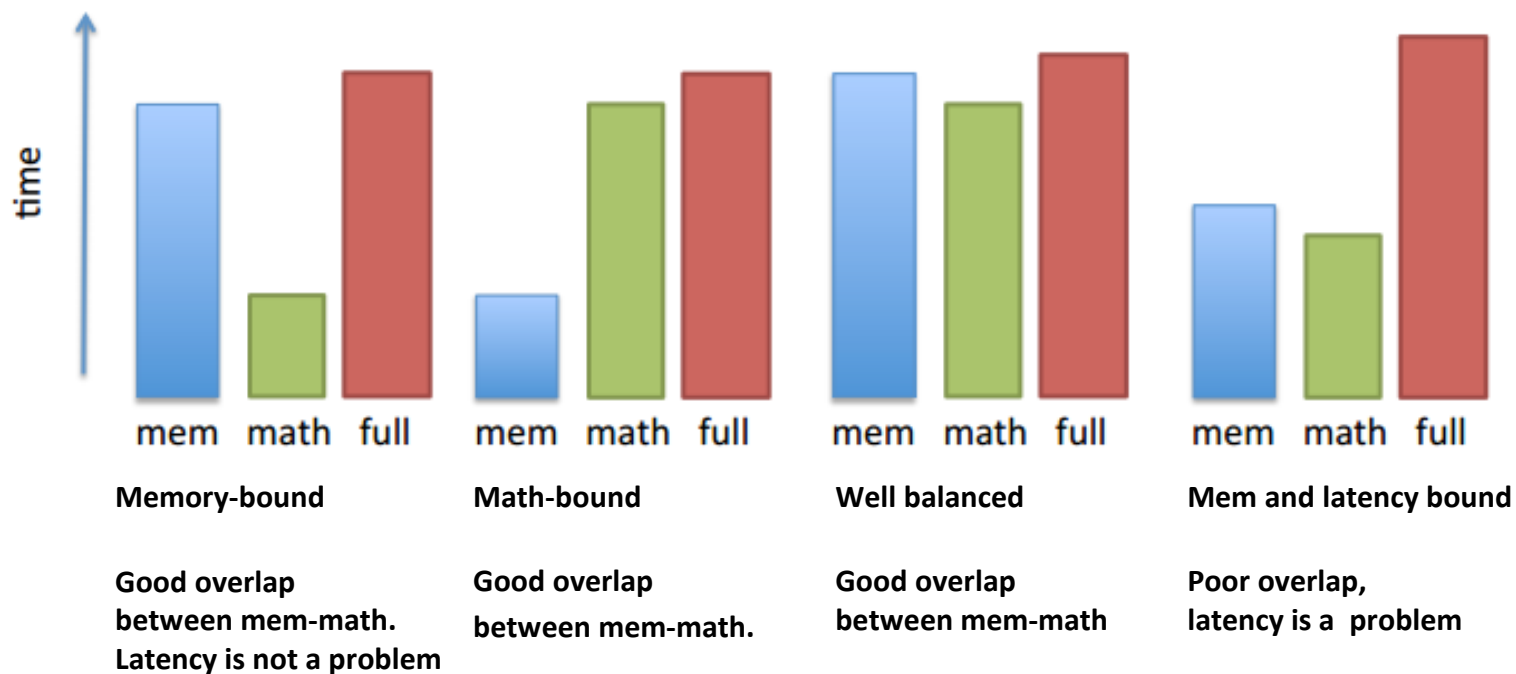
- **Time memory-only and math-only versions**
 - Not so easy for kernels with data-dependent control flow
 - Good to estimate time spent on accessing memory or executing instructions
- **Shows whether kernel is memory or compute bound**
- **Put an “if” statement depending on kernel argument around math/mem instructions**
 - Use dynamic shared memory to get the same occupancy

Analysis with Source Code Modification

```
__global__ void kernel(float *a) {  
    int idx = threadIdx.x + blockDim.x + blockIdx.x;  
    float my_a;  
    my_a = a[idx];  
  
    for(int i = 0; i < 100; i++)  
        my_a = sinf(my_a + I * 3.14f);  
  
    a[idx] = my_a;  
}
```

```
__global__ void kernel(float *a, int prof) {  
    int idx = threadIdx.x + blockDim.x + blockIdx.x;  
    float my_a;  
  
    if (prof & 1)  
        my_a = a[idx];  
    if (prof & 2)  
        for (int i = 0; i < 100; i++)  
            my_a = sinf(my_a + I * 3.14f);  
    if (prof & 1)  
        a[idx] = my_a;  
}
```

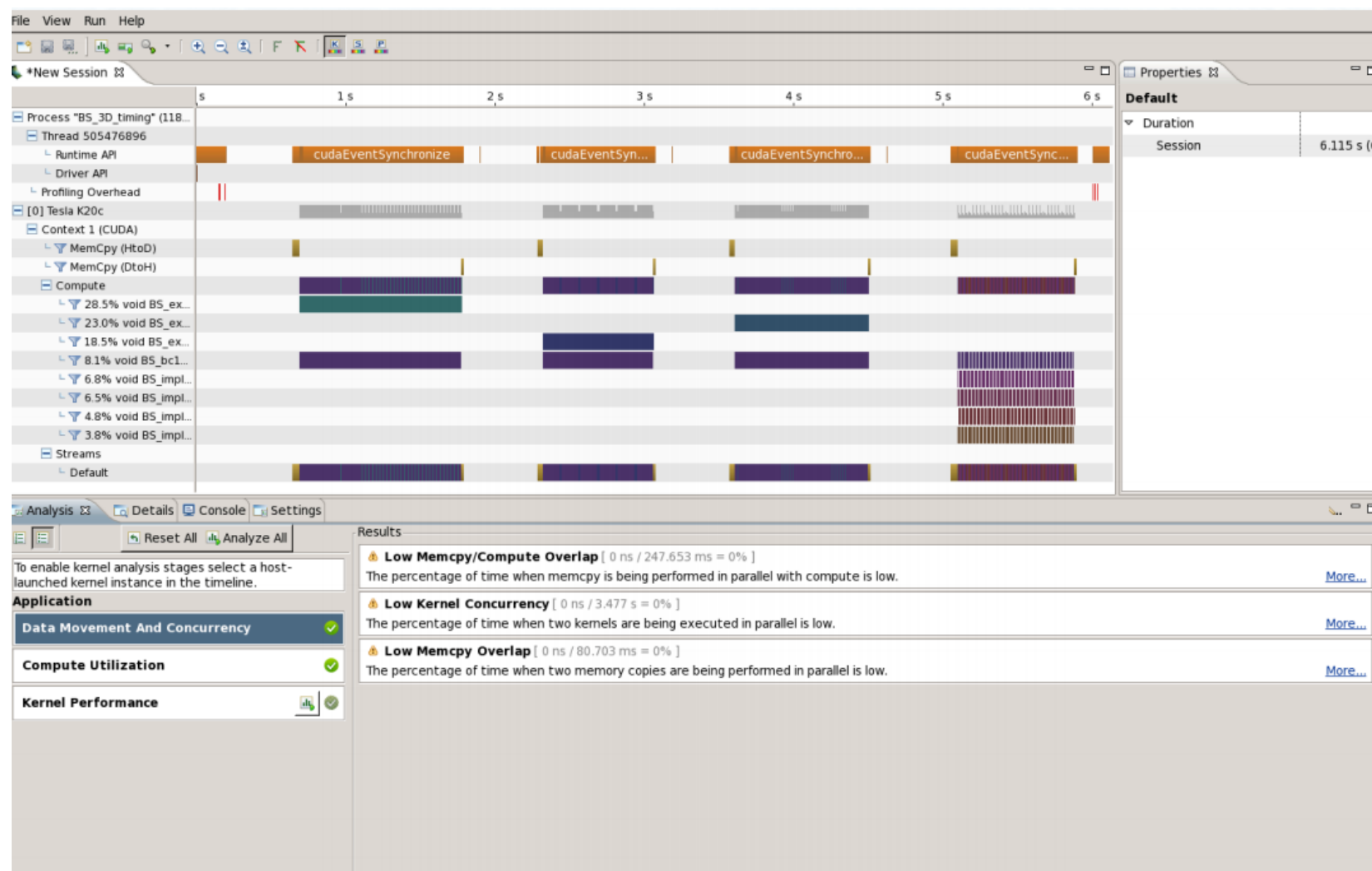
Example scenarios



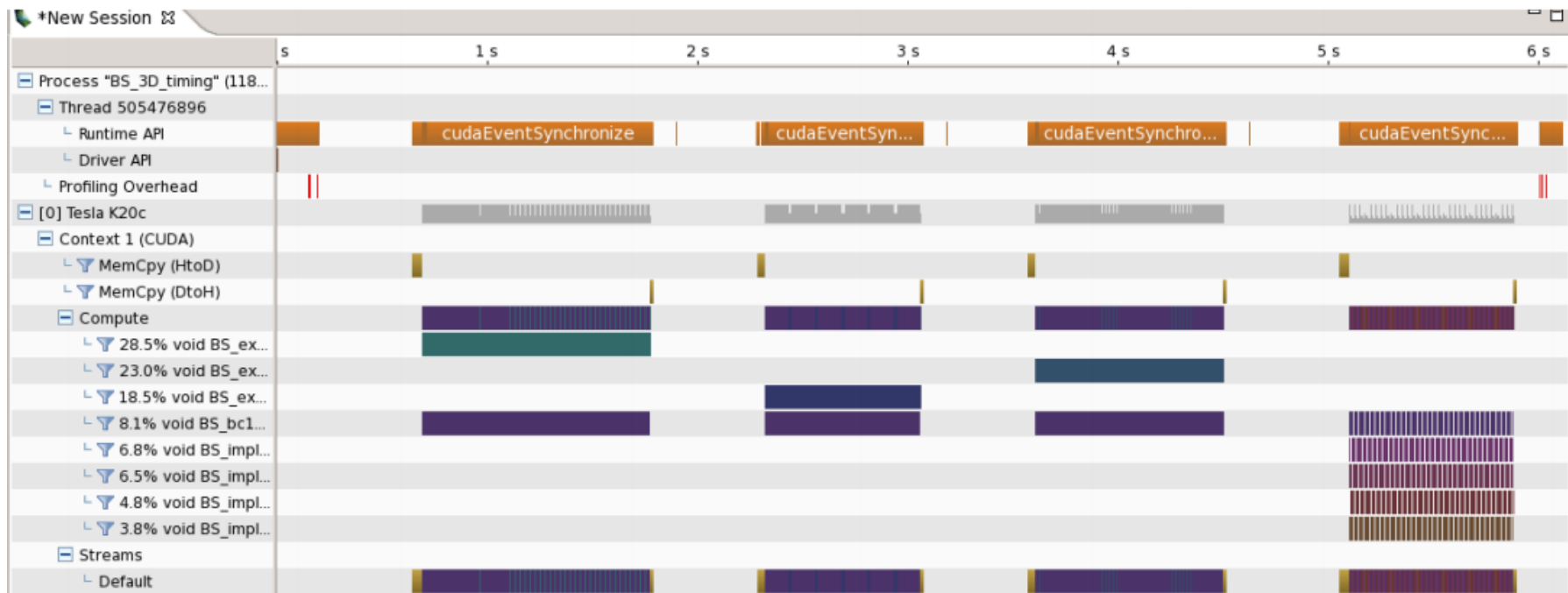
NVIDIA Visual Profiler

- **Launch with “nvvp”**
- **Collects metrics and events during execution**
 - Calls to the CUDA API
 - Overall application:
 - Memory transfers
 - Kernel launches
 - Kernels
 - Occupancy
 - Computation efficiency
 - Memory bandwidth efficiency
- **Requires deterministic execution!**

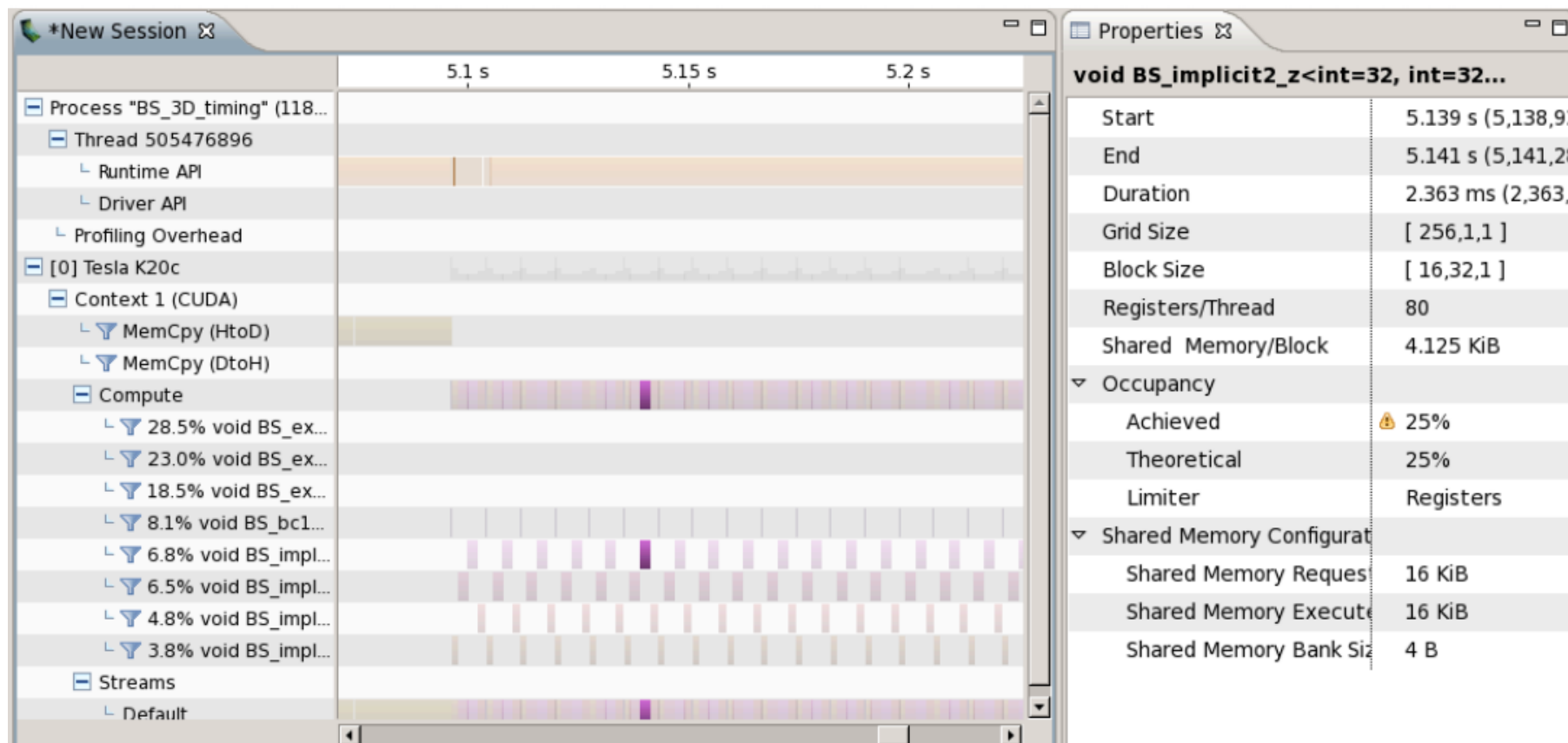
Visual Profiler



The timeline



Kernel properties



Analysis – Guided & Unguided

Guided

Unguided

Results of the analysis

The screenshot displays the NVIDIA Nsight system analysis tool interface. The top navigation bar includes tabs for Analysis, Details, Console, and Settings. Below the navigation bar, there are icons for Analysis, Details, Console, and Settings, and a button labeled 'Export PDF Report'. The main content area is divided into two sections. The left section, titled '1. CUDA Application Analysis' and '2. Check Overall GPU Usage', contains a paragraph explaining the analysis results and a button labeled 'Examine Individual Kernels'. The right section, titled 'Results', contains a list of analysis results, each with a warning icon, a title, a value in brackets, and a description. The results are: 'Low Memcpy/Compute Overlap' (0 ns / 247.653 ms = 0%), 'Low Kernel Concurrency' (0 ns / 3.477 s = 0%), and 'Low Memcpy Overlap' (0 ns / 80.703 ms = 0%). Each result has a 'More...' link. Below the results, there is a section titled 'i Compute Utilization' with an information icon and a paragraph explaining the device timeline.

Analysis Details Console Settings

Export PDF Report

1. CUDA Application Analysis

2. Check Overall GPU Usage

The analysis results on the right indicate potential problems in how your application is taking advantage of the GPU's available compute and data movement capabilities. You should examine the information provided with each result to determine if you can make changes to your application to increase GPU utilization.

Examine Individual Kernels

You can also examine the performance of individual kernels to expose additional optimization opportunities.

Results

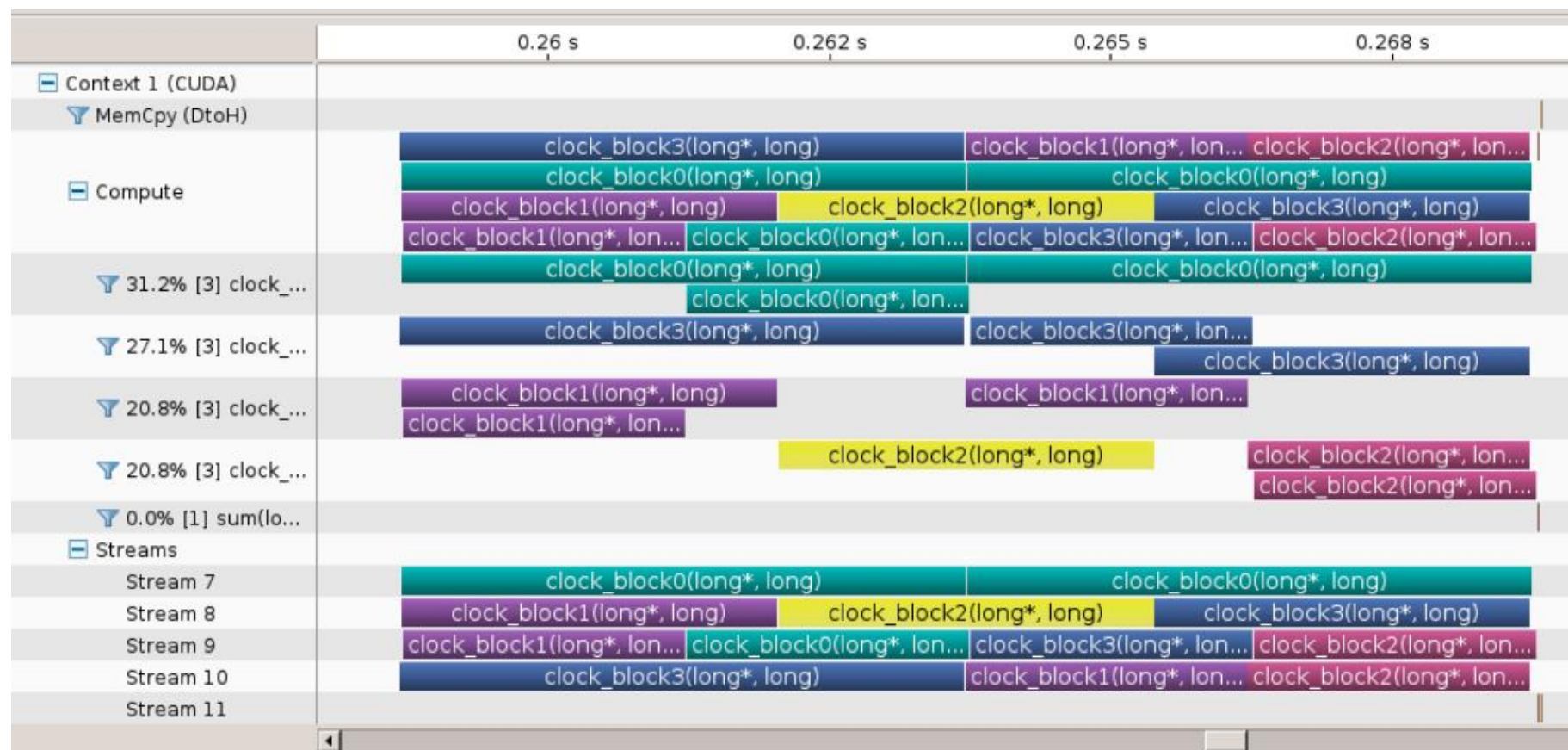
- Low Memcpy/Compute Overlap** [0 ns / 247.653 ms = 0%]
The percentage of time when memcpy is being performed in parallel with compute is low. [More...](#)
- Low Kernel Concurrency** [0 ns / 3.477 s = 0%]
The percentage of time when two kernels are being executed in parallel is low. [More...](#)
- Low Memcpy Overlap** [0 ns / 80.703 ms = 0%]
The percentage of time when two memory copies are being performed in parallel is low. [More...](#)

i Compute Utilization

The device timeline shows an estimate of the amount of the total compute capacity being used by the kernels executing on the device.

Visual Profiler Demo

Concurrent kernels



Metrics vs Events

Device: Tesla K20c ▼

Metrics | Events

- ▼ ☐ Memory
 - ☐ Requested Global Load Throughput
 - ☐ Requested Global Store Throughput
 - ☐ Device Memory Read Throughput
 - ☐ Device Memory Write Throughput
 - ☐ Global Store Throughput
 - ☐ Global Load Throughput
 - ☐ Shared Memory Efficiency
 - ☐ Global Memory Load Efficiency
 - ☐ Global Memory Store Efficiency
 - ☐ Local Memory Overhead
 - ☐ Requested Non-Coherent Global Load Throughput
 - ☐ Local Memory Load Transactions Per Request
 - ☐ Local Memory Store Transactions Per Request
 - ☐ Shared Memory Load Transactions Per Request
 - ☐ Shared Memory Store Transactions Per Request
 - ☐ Global Load Transactions Per Request

Device: Tesla K20c ▼

Metrics | Events

- ▼ ☐ Instruction
 - ☐ elapsed_cycles_sm
 - ☐ warps launched
 - ☐ threads launched
 - ☐ Instructions executed
 - ☐ Instructions issued 1
 - ☐ Instructions issued 2
 - ☐ thread inst executed
 - ☐ active cycles
 - ☐ active warps
 - ☐ sm cta launched
 - ☐ not_predicated_off_thread_inst_executed
- ▼ ☐ Memory
 - ☐ fb subp0 read sectors
 - ☐ fb subp1 read sectors
 - ☐ fb subp0 write sectors

How to “use” the profiler

- **Understand the timeline**
 - Where and when is your code
 - Add annotations to your application
 - NVIDIA Tools Extension (markers, names, etc.)
- **Find “obvious” bottlenecks**
- **Focus profiling on region of interest**
- **Dive into it**

Checklist

■ **cudaDeviceSynchronize()**

- Most API calls (e.g. kernel launch) are asynchronous
- Overhead when launching kernels
- Get rid of cudaDeviceSynchronize() to hide this latency
- Timing: events or callbacks in CUDA 5.0

■ **Cache config 16/48, 32/32 or 48/16 kB L1/shared (default is 48k shared!)**

- cudaSetDeviceCacheConfig
- cudaFuncSetCacheConfig
- Check if shared memory usage is a limiting factor

Checklist

Occupancy

- **Max 1536 threads or 8 blocks per SM on Fermi (2048/16 for Kepler)**
- **Limited amount of registers and shared memory**
 - Max 63 registers/thread, rest is spilled to global memory (255 for K20 Keplers)
 - You can explicitly limit it (-maxrregcount=xx)
 - 48kB/32kB/16kB shared/L1: don't forget to set it
- **Visual Profiler tells you what is the limiting factor**
- **In some cases though, it is faster if you don't maximise it (see Volkov paper) -> Autotuning!**

Verbose compile

■ Add `-Xptxas=-v`

ptxas info : Compiling entry function '_Z10fem_kernelPiS_' for 'sm_20'

ptxas info : Function properties for _Z10fem_kernelPiS_

856 bytes stack frame, 980 bytes spill stores, 1040 bytes spill loads

ptxas info : Used 63 registers, 96 bytes cmem[0]

■ Feed into Occupancy Calculator

Checklist

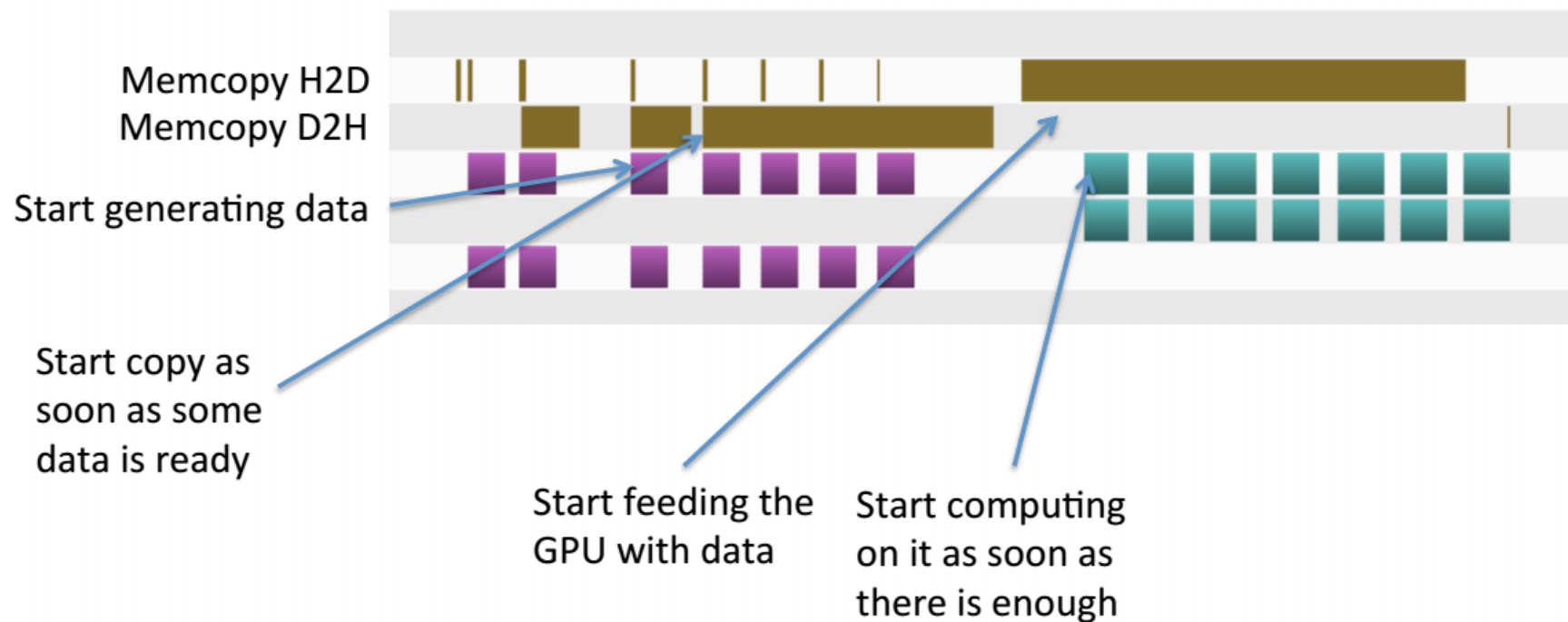
■ Precision mix (e. g. 1.0 vs 1.0f) –cuobjdump

- F2F.F64.F32 (6* the cost of a mul1ply)
- IEEE standard: always convert to higher precision
- Integer multiplications are now expensive (6*)

■ cudaMemcpy

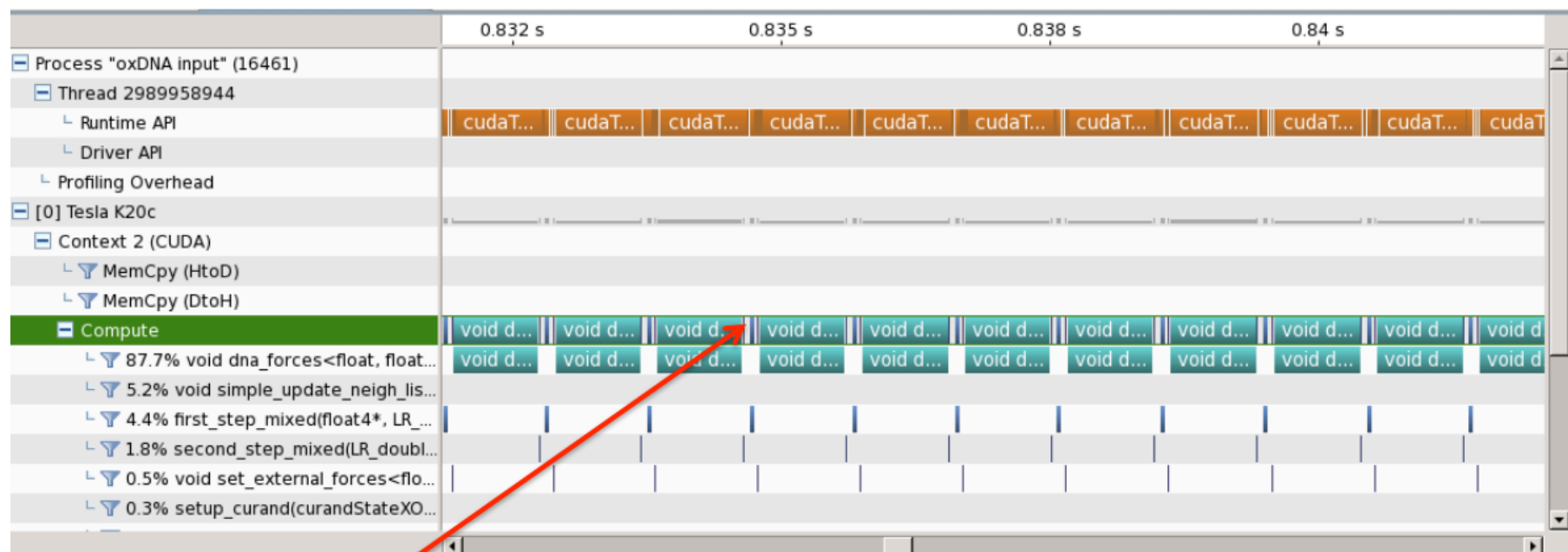
- Introduces explicit synchronisation, high latency
- Is it necessary?
 - May be cheaper to launch a kernel which immediately exits
- Could it be asynchronous? (Pin the memory!)

Asynchronous Memcopy

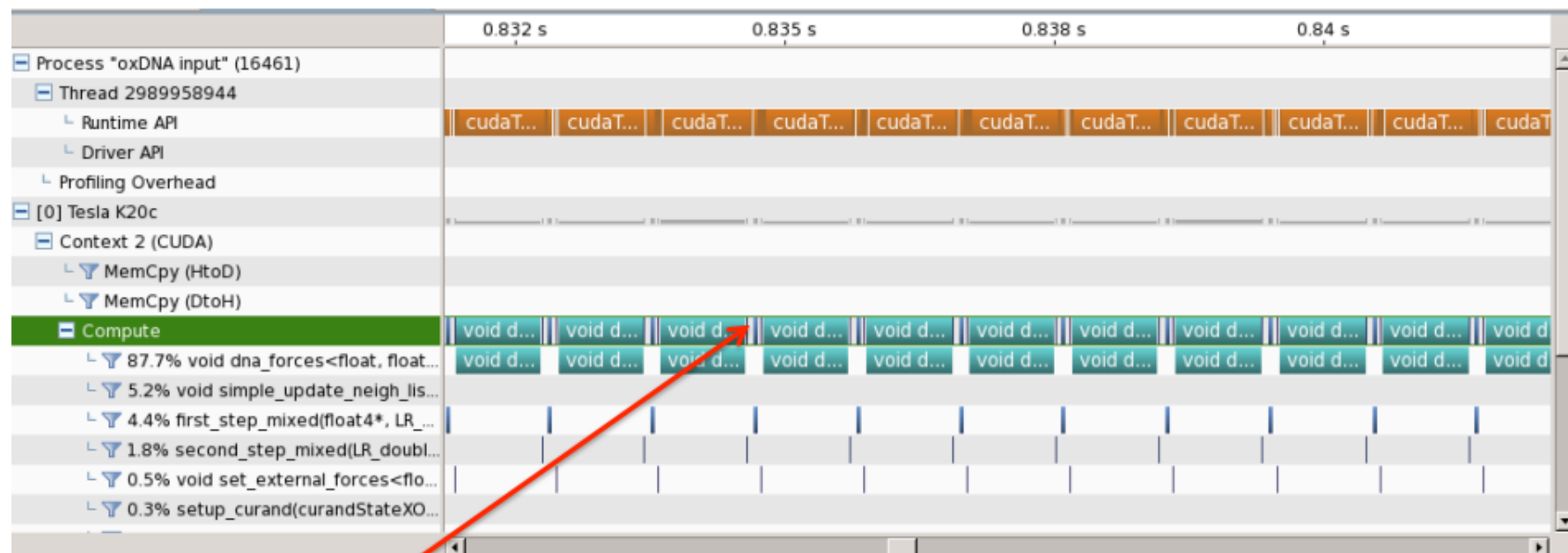


Case study

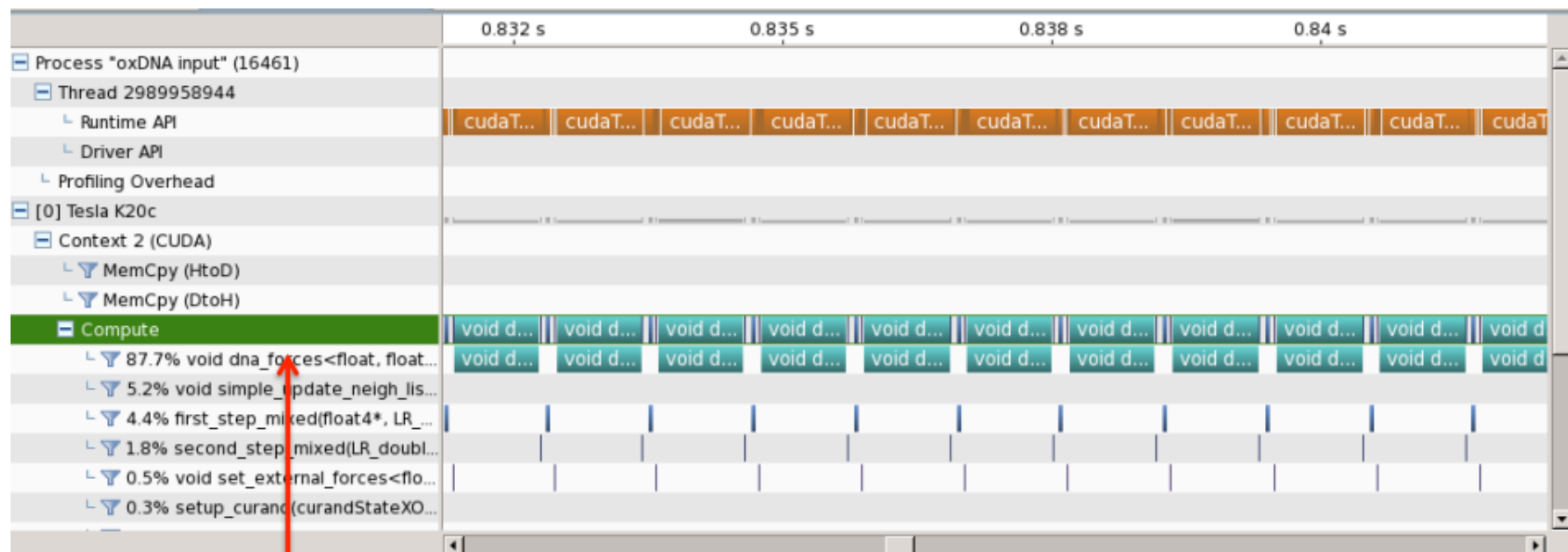
- **Molecular Dynamics**
- **~10000 atoms**
- **Short-range interaction**
 - Verlet lists
- **Very long simulation time**
 - Production code runs for ~1 month



- Gaps between kernels – get rid of `cudaDeviceSynchronize()` – “free” 8% speedup



- Gaps between kernels – get rid of `cudaDeviceSynchronize()` – “free” 8% speedup
- None of the kernels use shared memory – set L1 to 48k – “free” 10% speedup



- Gaps between kernels – get rid of `cudaDeviceSynchronize()` – “free” 8% speedup
- None of the kernels use shared memory – set L1 to 48k – “free” 10% speedup
- dna_forces is 81% of runtime

void dna_forces<float, float4>(float4*, ...	
Start	835.753 ms (835)
End	836.546 ms (836)
Duration	792.429 μ s
Grid Size	[121,1,1]
Block Size	[64,1,1]
Registers/Thread	110
Shared Memory/Block	0 B
Efficiency	
Global Load Efficiency	23.4%
Global Store Efficiency	100%
Shared Efficiency	n/a
Warp Execution Efficiency	24.1%
Non-Predicated Warp Ex	23.1%
Occupancy	
Achieved	16.4%
Theoretical	25%
Limiter	Registers
Shared Memory Configurati	
Shared Memory Reques	16 KiB
Shared Memory Execute	16 KiB
Shared Memory Bank Siz	4 B

Kernel analysis

- Fairly low runtime
 - Launch latency
- Few, small blocks
 - Tail
- Low theoretical
- Occupancy
 - 110 registers/thread
 - Even lower achieved ...
- L1 configuration
 - Analyze all

Kernel analysis

Name	Value
Global Load Efficiency	23.4%
Global Store Efficiency	100%
Global Load Throughput	52.14 GB/s
Global Store Throughput	0.65 GB/s

- **Memory**
- **Low efficiency**
- **But a very low total utilization (53 GB/s)**
- **Not really a problem**

Kernel analysis

Name	Value
Warp execution efficiency	24.1%
Issue Slot Utilization	32%

Instruction

- **Very high branch divergence**
 - Threads in a warp doing different things
 - SIMD – all branches executed sequentially
 - Need to look into the code
- **Rest is okay**

Kernel analysis

Name	Value
Theoretical	25%
Achieved	16.4%
Limiter	Block Size or Registers

Occupancy

- **Low occupancy**
- **Achieved much lower than theoretical**
 - Load imbalance, tail
- **Limiter is blocksize**
 - In this case doesn't help, there are already too few blocks
- **Structural problem**
 - Need to look into the code

Structural problems

1 thread per atom

- 10k atoms – too few threads
- Force computation with each neighbor
 - Redundant computations
 - Different number of neighbors – divergence

“Interaction” based computation

- Exploit symmetry
- Lots more threads, unit work per thread
- Atomic increment of values, only if non-0
- 4.3x speed up for force calculations, 2.5x overall

Memory-bound kernels

- What can you do if a kernel is memory-bound?

- Access pattern

- Profiler “Global Load/Store Efficiency”
- Struct of Arrays vs. Array of Structs



- Fermi cache: every memory transaction is 128 Bytes
- Rule of thumb: Get high occupancy to get close to theoretical bandwidth

nvprof

- **Command-line profiling tool**
- **Text output (CSV)**
 - CPU, GPU activity, trace
 - Event collection (no metrics)
- **Headless profile collection**
 - Can be used in a distributed setting
 - Visualise results using the Visual Profiler

Usage

- `nvprof [nvprof_args] <app> [app_args]`

Time(%),Time,Calls,Avg,Min,Max,Name

,us,,us,us,us,

58.02,104.2260,2,52.11300,52.09700,52.12900,"op_cuda_update()"

18.92,33.98600,2,16.99300,16.73700,17.24900,"op_cuda_res()"

18.38,33.02400,18,1.83400,1.31200,3.77600,"[CUDA memcpy HtoD]"

4.68,8.41600,3,2.80500,2.49600,2.97600,"[CUDA memcpy DtoH]"

- **Use `--query-events` to get a list of events you can profile**
- **Use `--query-metrics` and `--analysis-metrics` to get metrics (new in CUDA 5.5)**

Distributed Profiling

- `mpirun [mpirun args] nvprof -o out.%p -
profile-child-processes [nvprof args] <app>
[app args]`
 - Will create `out.PID#0`, `out.PID#1` ... files for different processes (based on process ID)
- **Import into Visual Profiler**
 - File/Import nvprof Profile

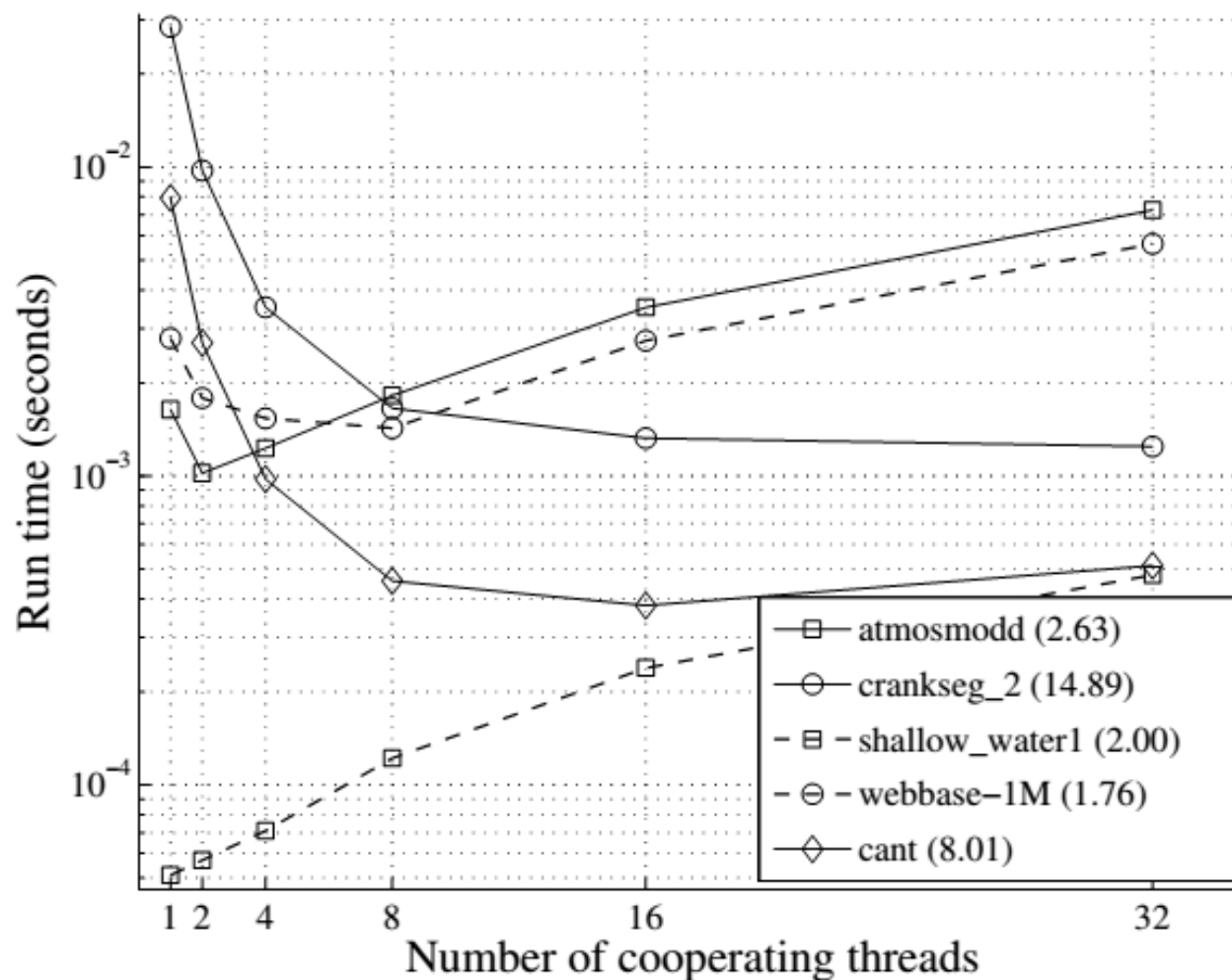
Auto-tuning

- **Several parameters that affect performance**
 - Block size
 - Amount of work per block
 - Application specific
- **Which combination performs the best?**
- **Auto-tuning with Flamingo**
 - #define/ read the sizes , recompile/rerun combinations

Auto-tuning Case Study

- **Thread cooperation on sparse matrix-vector product**
 - Multiple threads doing partial dot product on the row
 - Reduction in shared memory
- **Auto-tune for different matrices**
 - Difficult to predict caching behavior
 - Develop a heuristic for cooperation vs. average row length

Auto-tuning Case Study



Overview

- **Performance limiters**
 - Bandwidth, computations, latency
- **Using the Visual Profiler**
- **“Checklist”**
- **Case Study: molecular dynamics code**
- **Command-line profiling (MPI)**
- **Auto-tuning**