GPU Performance Nuggets

Simon Garcia de Gonzalo & Carl Pearson

PhD Students, IMPACT Research Group
Advised by Professor Wen-mei Hwu

Jun. 15, 2016

grcdgnz2@illinois.edu
pearson@illinois.edu

GPU Performance Programming

GPU Performance questions from Blue Waters users

1) Can I speed up my code on an XK node with a CUDA implementation

2) Is my CUDA implementation “fast” / why isn’t it faster?

These questions have answers, and you can answer them!

Outline of this talk:

- Introduce a pair of NVIDIA performance tools available on Blue Waters
  - What the GPU memory hierarchy provides for your application
  - Can memory hierarchy optimization go too far? A Blue Waters case study.

`nvprof`: collect (or view) profiling data

```
aprun nvprof          \
  -o timeline.nvp \
  ./my-cuda-app
```

`timeline.nvprof`

Timeline of CUDA runtime calls, kernel execution times, etc. ~No run time overhead

```
aprun nvprof              \
  --analysis-metrics \
  -o analysis.nvp     \
   ./my-cuda-app
```

`analysis.nvprof`

Detailed performance data for each kernel execution. Large run time overhead

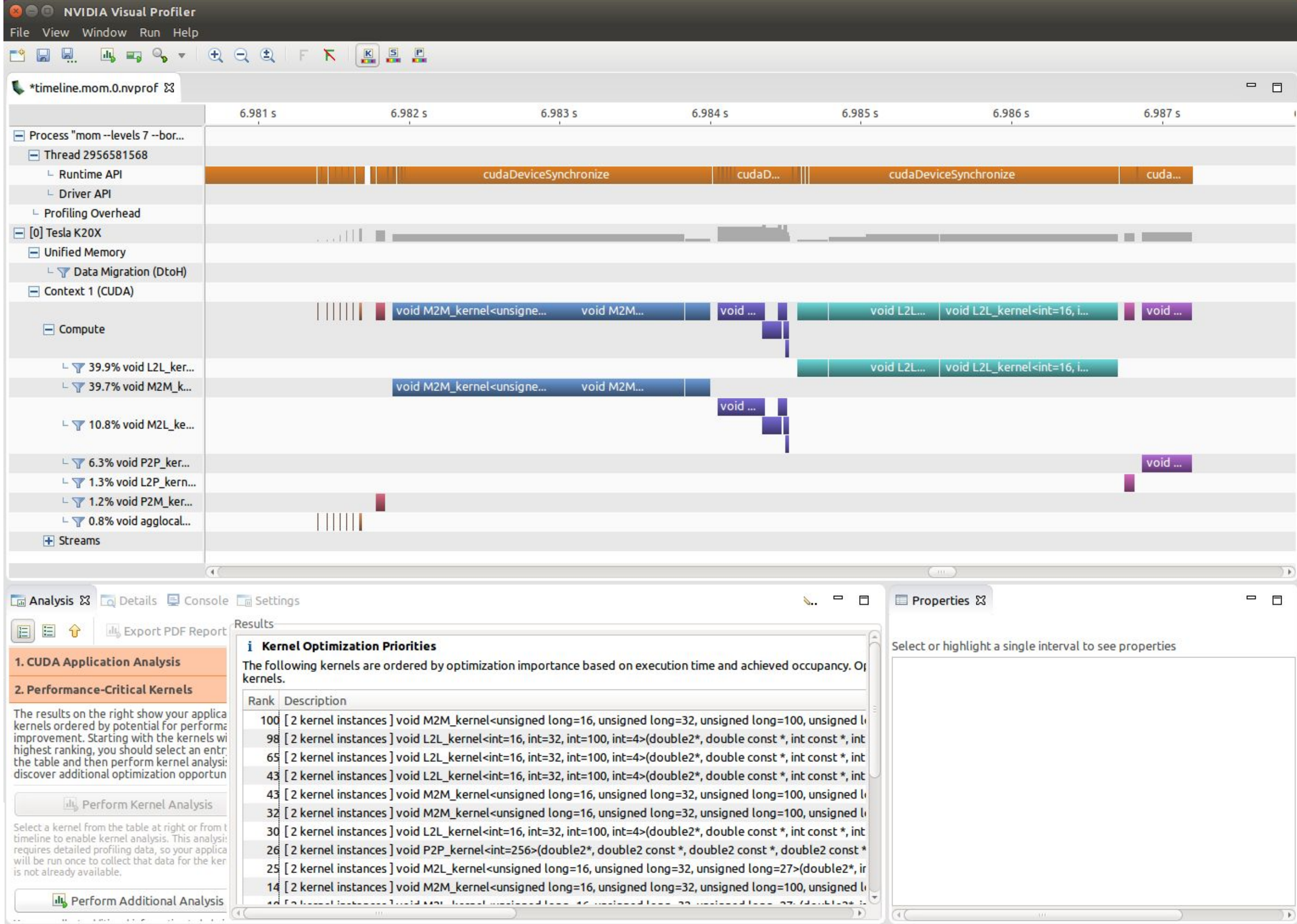
Aside: `nvprof` and `MPI`

Prevent MPI runtime from forking the app into a separate process, which hides it from nvprof

```
PMI_NO_FORK=1                                   \
aprun nvprof                                    \
  -o timeline.%q{ALPS_APP_PE}.nvprof \
  ./my-cuda-mpi-app
```

Unique profile output file per MPI rank

nvvp


NVIDIA Visual Profiler
File View Window Run Help
*timeline.mom.0.nvprof
6.981 s
6.982 s
6.983 s
6.984 s
6.985 s
6.986 s
6.987 s
Process "mom --levels 7 --bor...
Thread 2956581568
Runtime API
Driver API
Profiling Overhead
[0] Tesla K20X
Unified Memory
Data Migration (DtoH)
Context 1 (CUDA)
Compute
39.9% void L2L_ker...
39.7% void M2M_k...
10.8% void M2L_ke...
6.3% void P2P_ker...
1.3% void L2P_kern...
1.2% void P2M_ker...
0.8% void agglocal...
Streams
cudaDeviceSynchronize
cudaD...
cuda...
void M2M_kernel<unsigne...
void M2M...
void ...
void L2L...
void L2L_kernel<int=16, i...
Analysis
Details
Console
Settings
Export PDF Report
Results
1. CUDA Application Analysis
2. Performance-Critical Kernels
The results on the right show your applica kernels ordered by potential for performa improvement. Starting with the kernels wi highest ranking, you should select an entr the table and then perform kernel analysi discover additional optimization opportun
Perform Kernel Analysis
Select a kernel from the table at right or from t timeline to enable kernel analysis. This analysi requires detailed profiling data, so your applica will be run once to collect that data for the ker is not already available.
Perform Additional Analysis
Kernel Optimization Priorities
The following kernels are ordered by optimization importance based on execution time and achieved occupancy. O kernels.
Rank
Description
100 [2 kernel instances] void M2M_kernel<unsigned long=16, unsigned long=32, unsigned long=100, unsigned l
98 [2 kernel instances] void L2L_kernel<int=16, int=32, int=100, int=4>(double2*, double const *, int const *, int
65 [2 kernel instances] void L2L_kernel<int=16, int=32, int=100, int=4>(double2*, double const *, int const *, int
43 [2 kernel instances] void L2L_kernel<int=16, int=32, int=100, int=4>(double2*, double const *, int const *, int
43 [2 kernel instances] void M2M_kernel<unsigned long=16, unsigned long=32, unsigned long=100, unsigned l
32 [2 kernel instances] void M2M_kernel<unsigned long=16, unsigned long=32, unsigned long=100, unsigned l
30 [2 kernel instances] void L2L_kernel<int=16, int=32, int=100, int=4>(double2*, double const *, int const *, int
26 [2 kernel instances] void P2P_kernel<int=256>(double2*, double2 const *, double2 const *, double2 const *
25 [2 kernel instances] void M2L_kernel<unsigned long=16, unsigned long=32, unsigned long=27>(double2*, i
14 [2 kernel instances] void M2M_kernel<unsigned long=16, unsigned long=32, unsigned long=100, unsigned l
Properties
Select or highlight a single interval to see properties

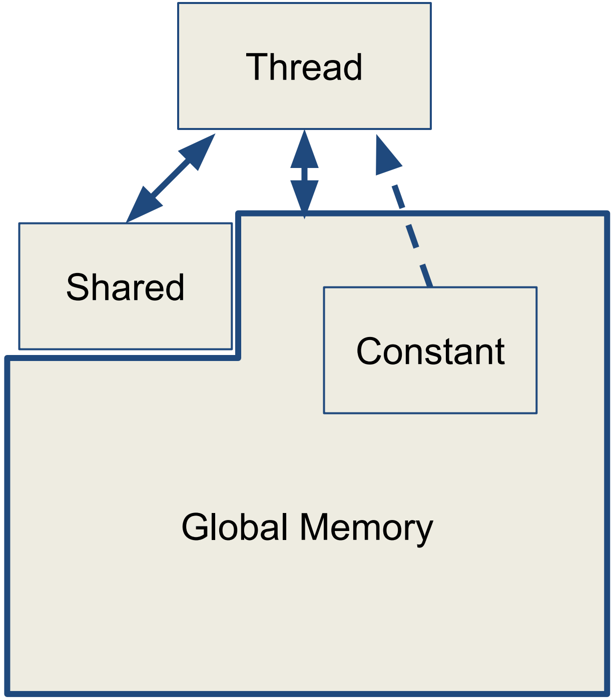
K20X Peak Memory Bandwidth

BLUE WATERS
SUSTAINED PETASCALE COMPUTING

Accelerator	Peak Single-Precision Rate (TFLOPS)	Peak Global Memory Bandwidth (GB/s)	FLOPS / word
C2070 (Fermi)	1.03	144	28.7
K20X (Kepler)	**3.94**	**250**	**63.0**
M40 (Maxwell)	5.83	288	80.9
P100 (Pascal)	9.52	720 (!!!)	52.9

(most) GPU kernels are limited by memory before compute

CUDA Compute Capability 3.5 Memory Model



Thread-Private Memory

48KB Shared Memory

6GB Global Memory

Shared Memory:

Accelerate predictable repeated access to data.

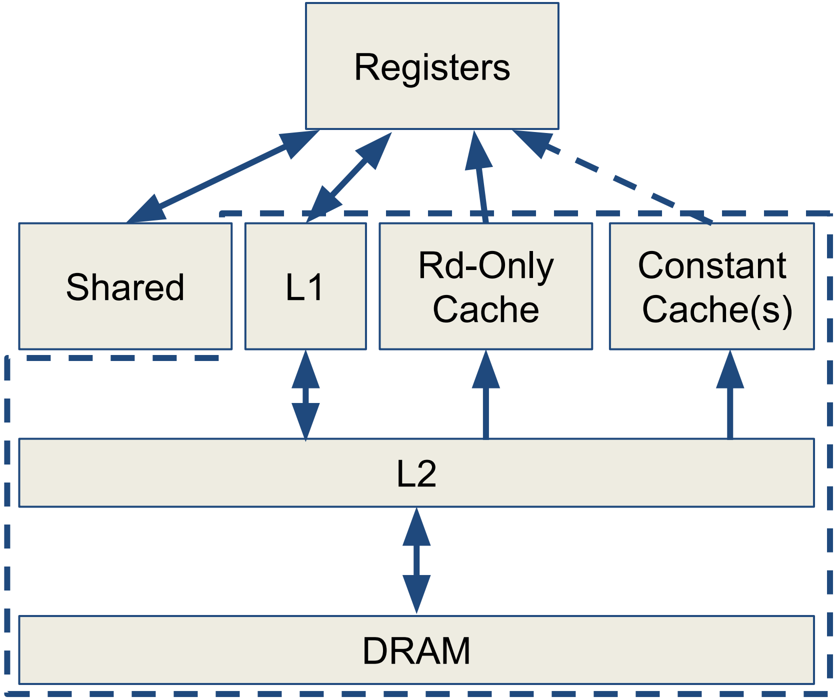
Constant Memory:

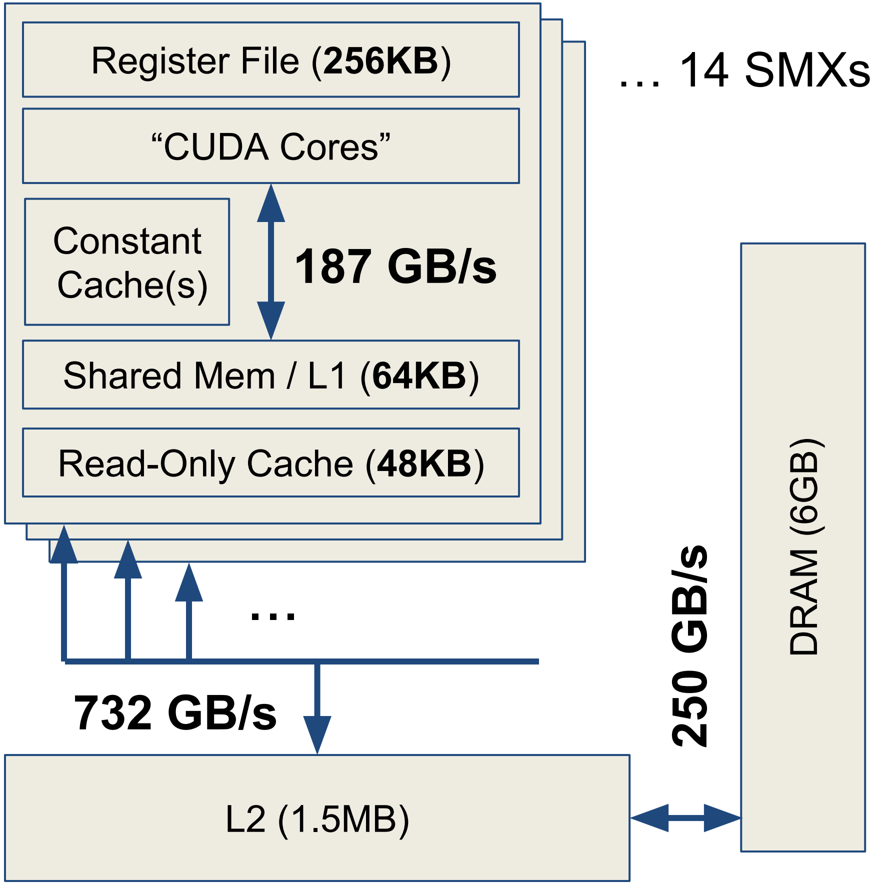
High bandwidth access to read-only data

Global Memory:

Data used by GPU kernels must be here

K20x Memory Subsystem





Use the memory hierarchy to reduce the DRAM FLOPS/word ratio

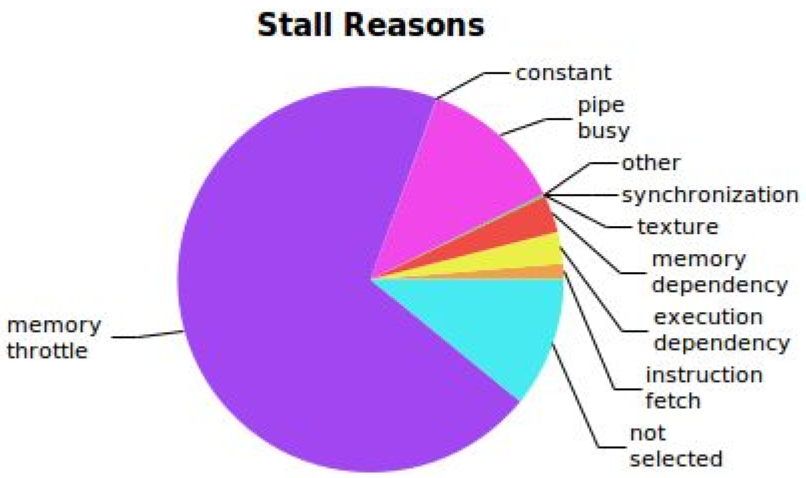
Different memories for different data

L1 Cache	Register spills / stack data
L2 Cache	Global data locality across thread blocks
Read-Only Cache	Unaligned, random, read-only, 2D prefetch
Constant Cache	Aligned, uniform, read-only, “very small”
Shared Memory	Predictable locality within a thread block
DRAM	Aligned, consecutive access by consecutive threads

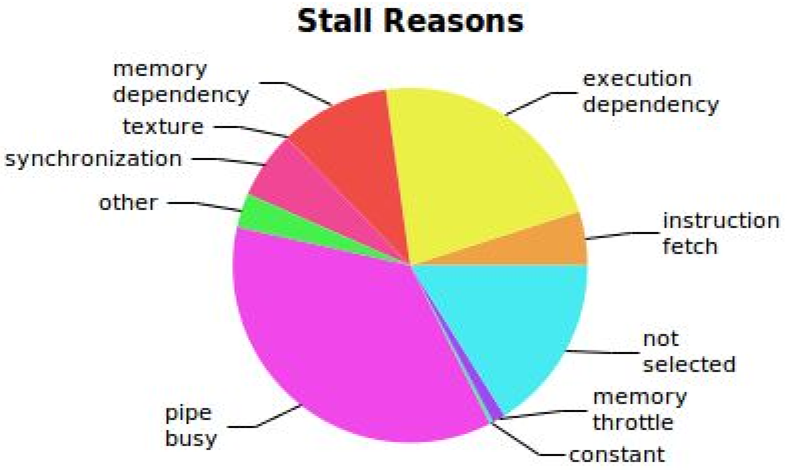
If most of your memory accesses match one of these patterns, good results are possible.

nvvp: Stencil Stall Reasons

Simple



Shared / Constant Memory

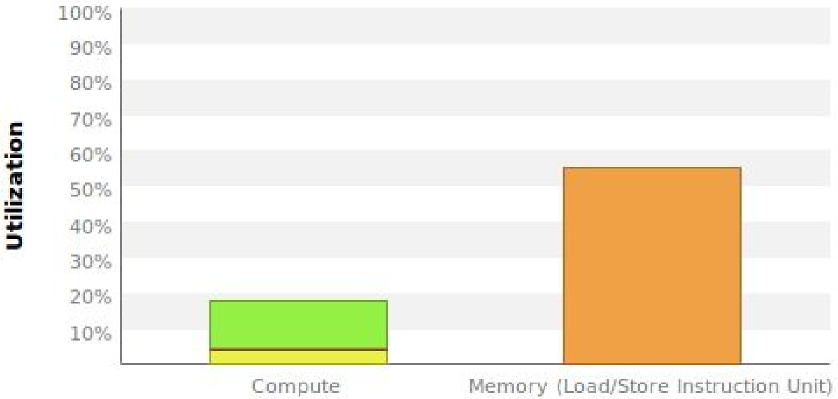


nvvp: Memory Bandwidth (stencil)

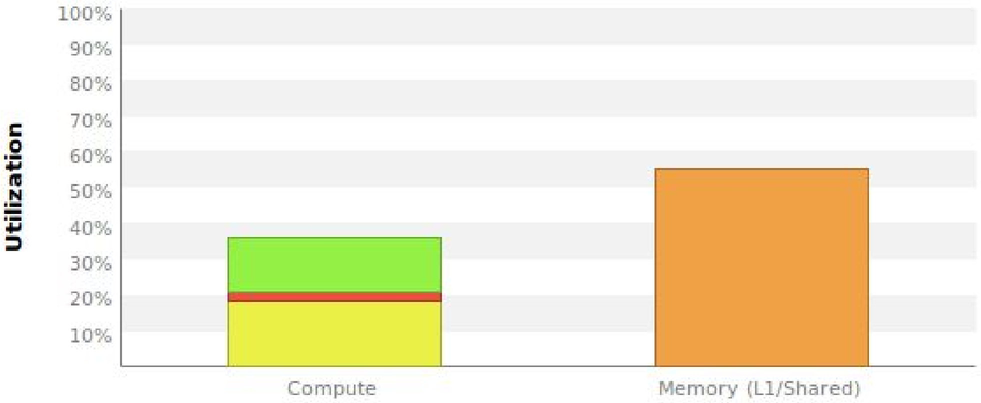
	Simple (GB/s)	Optimized (GB/s)
L1 Global Loads	105.4	34.1
Shared Loads	**0.0**	**1228.4**
Device Memory Reads	4	27.4
Device Memory Writes	3.9	26.1
Speedup	**1**	**7.8**

nvvp: Utilization (stencil)

Simple



Shared / Constant Memory

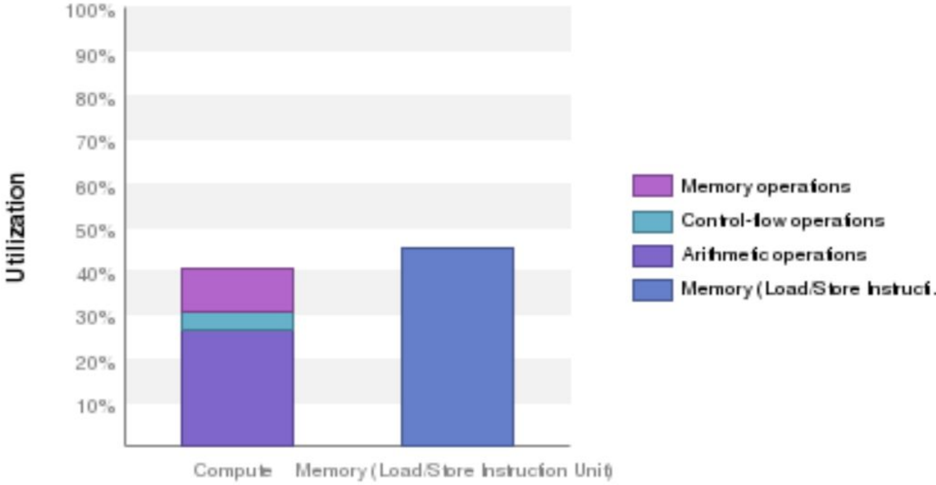


Memory operations   Control-flow operations   Arithmetic operations

nvvp: “Kernel performance is bound by instruction and memory latency!”

Latency limited kernels

- Characterized by having both low compute utilization and low memory utilization
- Low GPU occupancy is the main factor in this type of limitation.
- Unlike latency oriented CPUs, GPUs need a large degree of ILP to hide instruction latency.
- Common issue for highly optimized kernels that overuse limited resources that lowers possible achievable occupancy.



Resources that limit occupancy

- The following table contain the resources that are most likely to cause low occupancy

Accelerator	Maximum Threads per SM	Maximum Blocks per SM	Shared Memory per SM	Maximum Registers per Threads
C2070 (Fermi)	1536	8	48KB	63
K20X (Kepler)	2048	16	48KB	255
M40 (Maxwell)	2048	32	96KB	255
P100 (Pascal)	2048	32	64KB	255

BLUE WATERS
SUSTAINED PETASCALE COMPUTING

Case study: Reducing share memory

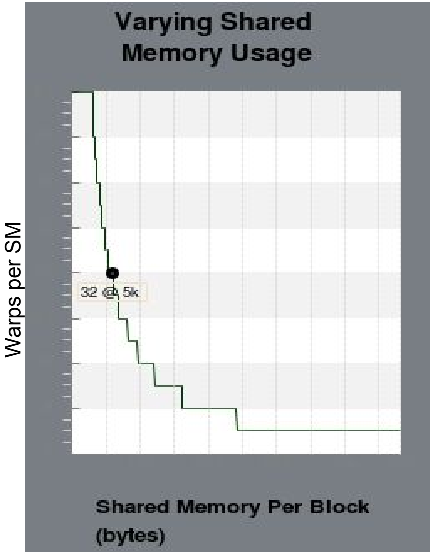
- 5.97KB of shared memory per block was being used
- Tesla K20X is configured to have 48KB of shared memory per SMX
- Each SMX was limited to simultaneously execute only 8 blocks (32 warps) out of the possible 16 block (64 warps)
- What to do:

```
//__shared__ CudaVector3D acc[THREADS_PER_BLOCK_PART];
// __shared__ cudatype pot[THREADS_PER_BLOCK_PART];
// __shared__ cudatype idt2[THREADS_PER_BLOCK_PART];
CudaVector3D acc;
cudatype pot;
cudatype idt2;
```

  - Shuffle instruction for reduction

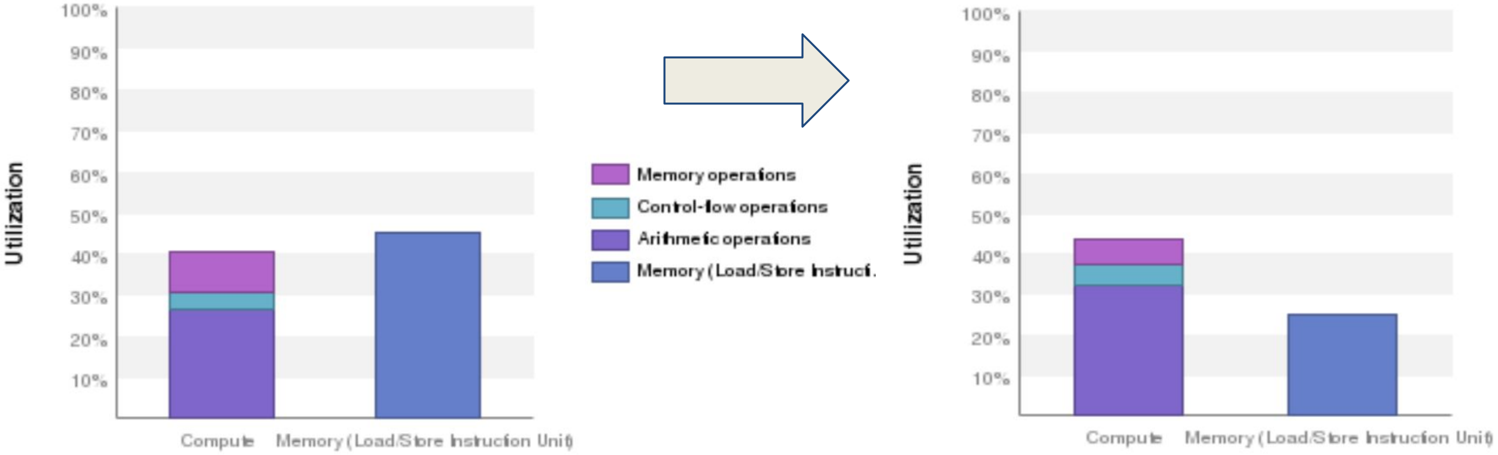
```
sumx += __shfl_down(sumx, offset, NODES_PER_BLOCK_PART);
sumy += __shfl_down(sumy, offset, NODES_PER_BLOCK_PART);
sumz += __shfl_down(sumz, offset, NODES_PER_BLOCK_PART);
poten += __shfl_down(poten, offset, NODES_PER_BLOCK_PART);
```

  - Some __syncthreads() can be removed due to threads not having to wait for all threads to read or write to shared memory



Case study: Reducing share memory

- By using less shared memory we lowered the memory utilization as expected but did not improve the compute utilization…. We are still Latency limited!



- Register usage could be the limiting resources.

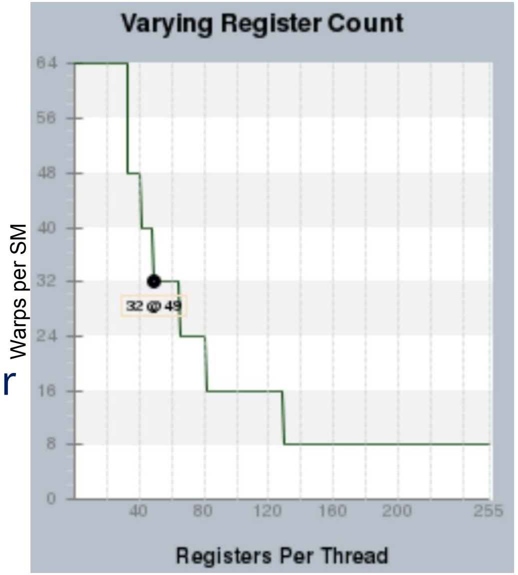
Case study: Reducing registers usage

- 56 registers per thread was being used or 14336 registers per block
- Tesla K20X is configured to have up to 65536 registers per SMX
- Each SMX was limited to simultaneously execute only 4 blocks (32 warps) out of the possible 16 block (64 warps)
- No direct way of controlling register usage, but we can help the compiler to do a better job.
- What to do:

  __launch_bounds__(**maxThreadsPerBlock**, minBlockPerMultiProc)

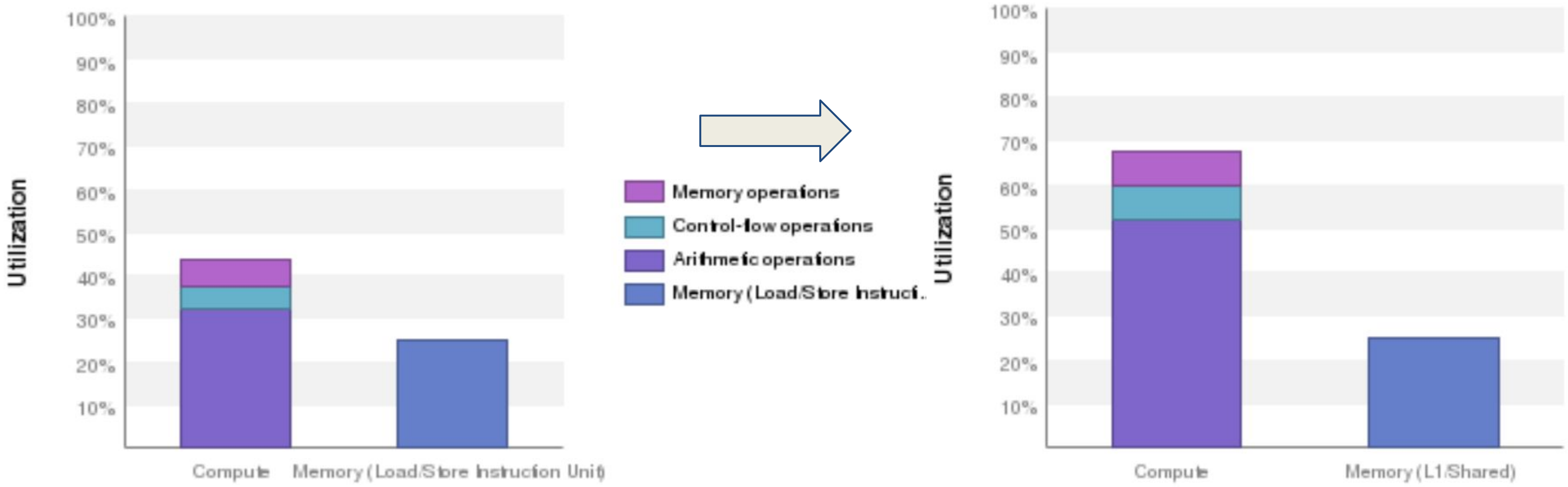
- The compiler will derive the number of register it needs per threads to be able to handle minBlockPerMultiProc***maxThreadsPerBlock** per SMX.

  NUM_REG ↓ LOCAL_MEM ↑ NUM_INSTRUCTIONS ↑



Case study: Reducing registers usage

- Register usage decreased from 56 to 24 thus utility rose to approximately 70%



- Further reducing register usage causes spilling onto global memory adversely affecting execution time!

What does it all mean in terms of speedup

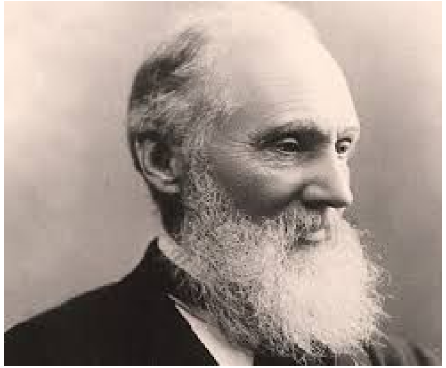
- Two kernels from ChaNGa, N-Body Cosmological application, (Prof. Thomas Quinn, University of Washington) :
  - particleGravityComputation
  - nodeGravityComputation

- Both kernels are non-trivial and highly optimized making use of shared memory.

- After described latency optimizations:
  - particleGravityComputation
    - Utilization improved from about 40% to 70%
    - 1.66x speedup
  - nodeGravityComputation
    - Utilization improved from about 30% to 60%
    - 2.11x speedup

Takeaways

- Writing a CUDA kernels is becoming easier, but getting good performance is not.
- Know the tools you have available. Profiling is key to performance
- Fitting your application to the GPU memory hierarchy is critical for performance
- Resources are not infinite, optimization without thinking about resources sizes can hurt performance.



Lord Kelvin

"To measure is to know"

"If you can not measure it, you can not improve it"

Resources

- nvprof and nvvp:
  - https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/
  - https://devblogs.nvidia.com/parallelforall/cudacasts-episode-19-cuda-6-guided-performance-analysis-visual-profiler/

- Latency limited kernels:
  - https://nvlabs.github.io/moderngpu/performance.html

- Shuffle instructions:
  - https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-kepler-shuffle/
  - https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/

- Launch_bounds qualifier:
  - https://nvlabs.github.io/moderngpu/performance.html#launchbounds

- Teaching kits:
  - https://developer.nvidia.com/teaching-kits

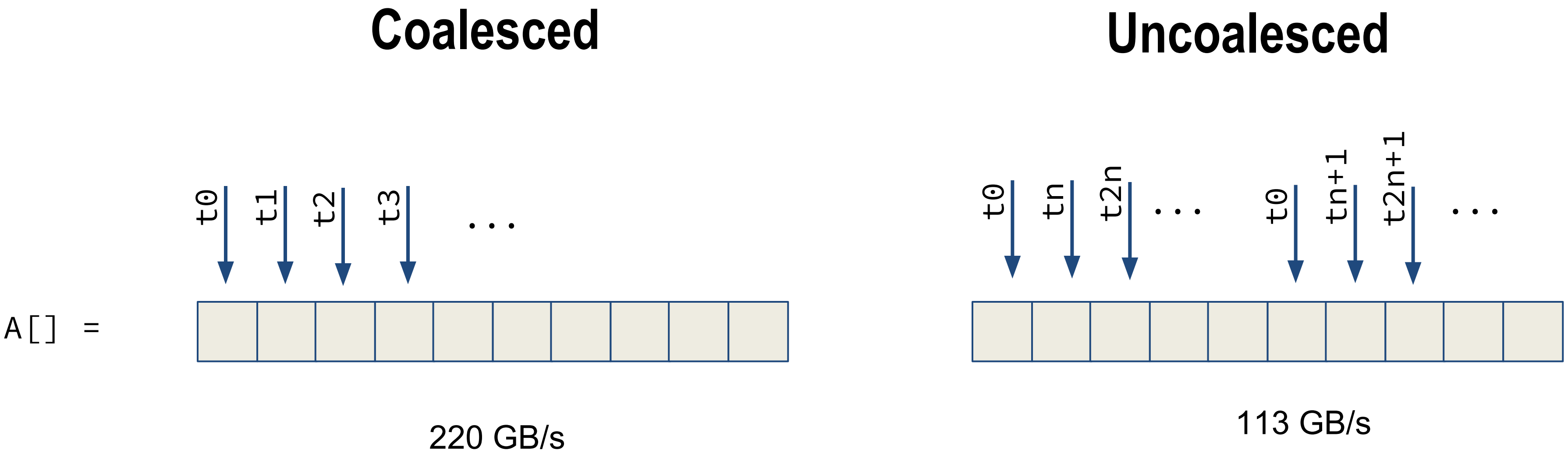
End

pearson@illinois.edu
grcdgnz2@illinois.edu

GPU Performance Programming

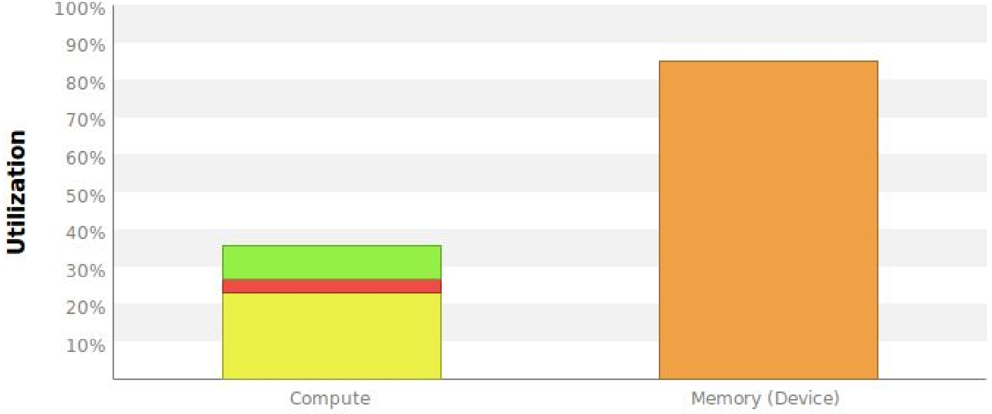
- Common
  - Latency-limited
  - Memory-bandwidth-limited
- Less Common
  - Compute-resource limited
  - Not enough parallelism

nvvp: Coalesced and Uncoalesced Accesses

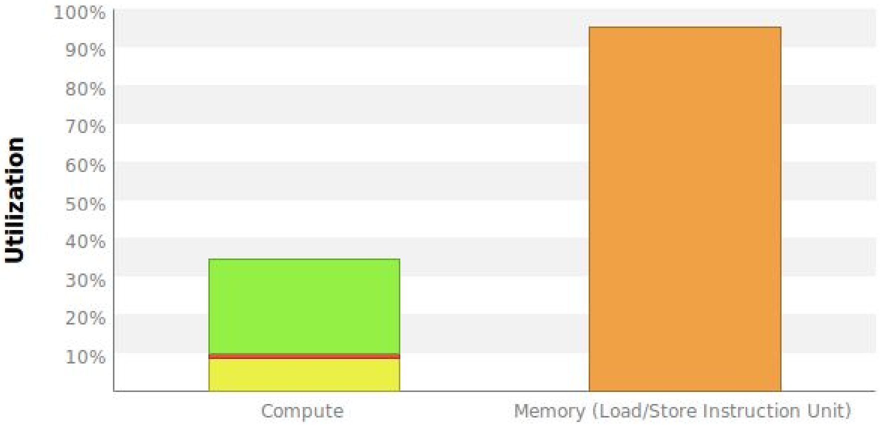


nvvp: Coalesced and Uncoalesced Accesses

Coalesced



Uncoalesced

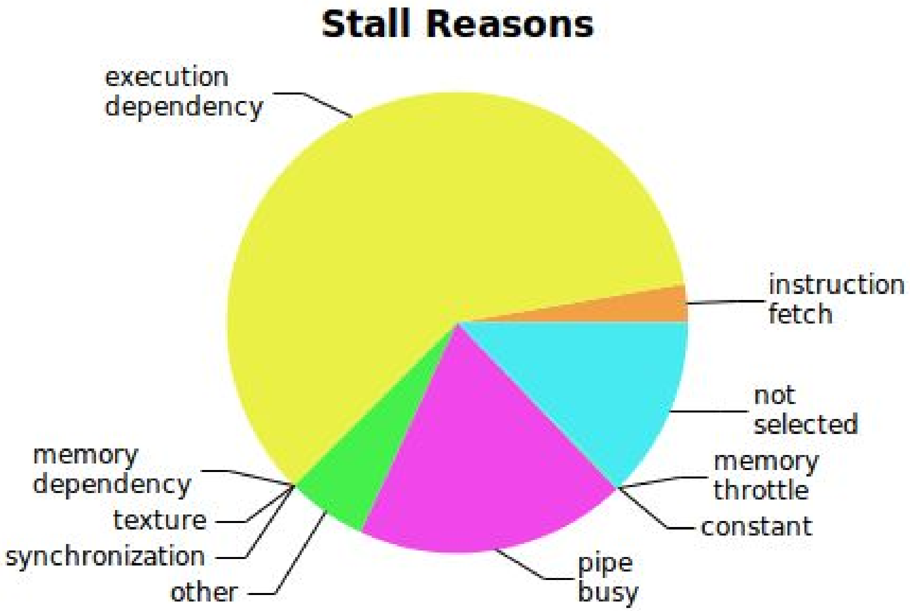


Memory operations Control-flow operations Arithmetic operations

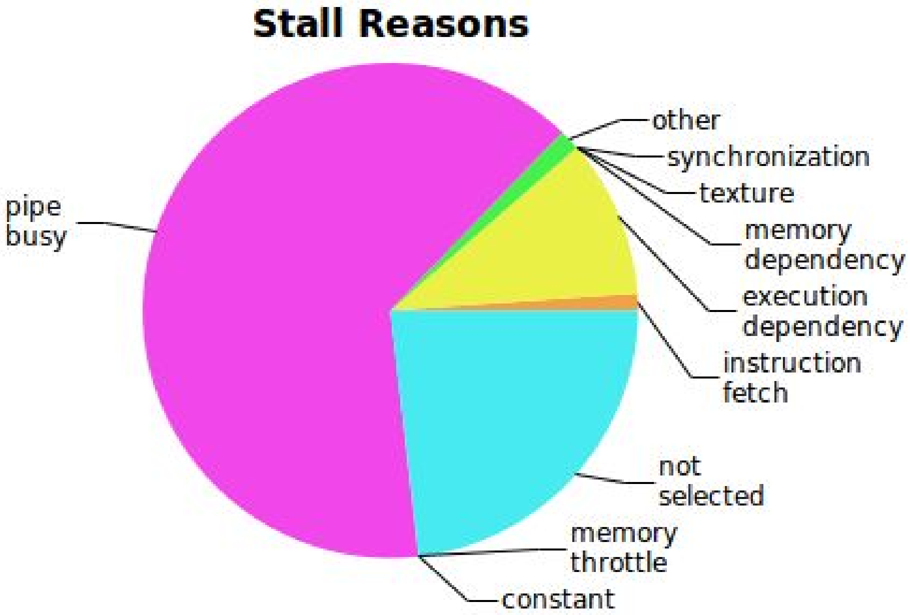
nvvp: "The performance of the kernel is most likely being limited by the memory system"

nvvp: Coalesced and Uncoalesced Accesses

Coalesced



Uncoalesced



nvvp: Coalesced and Uncoalesced Accesses

	Coalesced (GB/s)	Uncoalesced (GB/s)
L1 Cache Writes	184.621	277.179
Device Memory Reads	0.009	26.293 (?)
Device Memory Writes	220.102	113.181
Device Memory Total	220.111	139.474

nvvp: Coalesced and Uncoalesced Accesses

Global Memory Alignment and Access Pattern

Memory bandwidth is used most efficiently when each global memory load and store has proper alignment and access pattern.

Optimization: Select each entry below to open the source code to a global load or store within the kernel with an inefficient alignment or access pattern. For each load or store improve the alignment and access pattern of the memory access. More...

Line / File	vector_write.cu - /mnt/a/u/sciteam/cpearson/cuda-test/vector-write
66	Global Store L2 Transactions/Access = 32, Ideal Transactions/Access = 8 [16777216 L2 transactions for 524288 total executions]

```
63:  const int i = blockDim.x * blockIdx.x + threadIdx.x;
64:  const int j = blockDim.y * blockIdx.y + threadIdx.y;
65:  if (j < SIZE_X && i < SIZE_Y) {
66:    dst[i * SIZE_X + j] = val;     // row-major
67:  }
```