Master Educational Program
“Information technology in applications”

Mathematical computations with GPUs

CUDA program optimization

Alexey A. Romanenko
arom@ccfit.nsu.ru
Novosibirsk State University

Workflow

- Gathering information
  - memory bandwidth
  - instruction throughput
  - latencies
  - all above
- Studying limitation factors ordered by priority
  - Measure
  - Analyses
  - Optimization

Profiling

- Environment variables (old style)
  - CUDA_PROFILE=1
  - CUDA_PROFILE_LOG="cuda_profile_%p_%d.log"
  - CUDA_PROFILE_CONFIG
  - CUDA_PROFILE_CSV=1
- Environment variables (modern style)
  - COMPUTE_PROFILE …

Profiling. Configure file

* timestamp
* gridsize
* threadblocksize
* dynsmemperblock
* stasmemperblock
* regperthread
* memtransferdir
* memtransfersize
* streamid

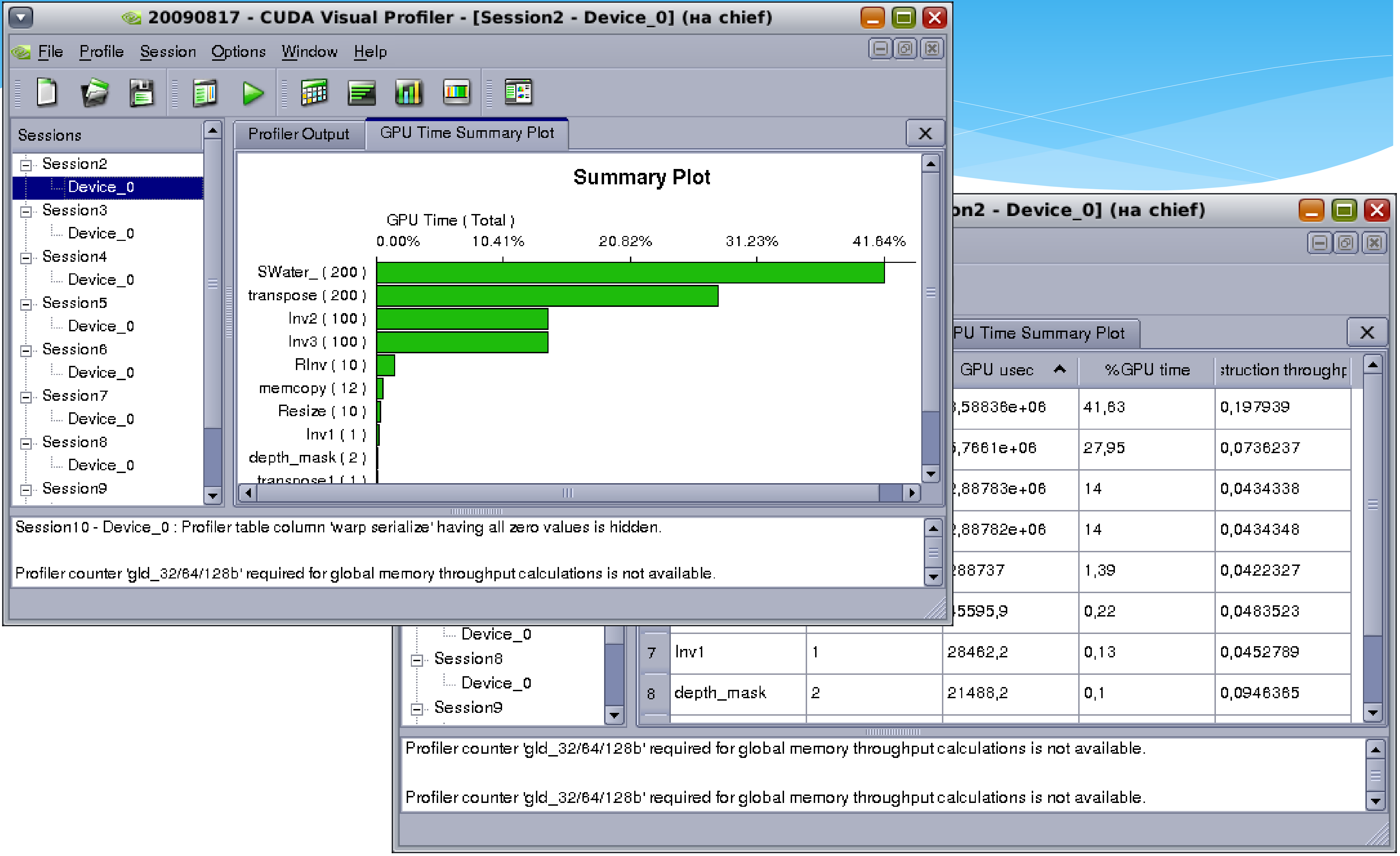
Profiling.
Configure file(CUDA 2.3)

- gld_incoherent
- gld_coherent
- gld_32b / gst_32b
- gld_64b / gst_62b
- gld_128b / gst_128b
- gld_request
- gst_incoherent
- gst_coherent
- gst_request
- local_load
- local_store
- branch
- divergent_branch
- instructions
- warp_serialize
- cta_launched
- gputime
- cputime
- occupancy

Profiling. Output

```
method,gputime,cputime,occupancy
method=[ memcopy ] gputime=[ 5519.680 ]
method=[ memcopy ] gputime=[ 5516.992 ]
method=[ memcopy ] gputime=[ 5517.728 ]
method=[ memcopy ] gputime=[ 5508.288 ]
method=[ __globfunc__Z10depth_maskPfS_fii ] gputime=[ 1791.424 ] cputime=[ 14.000 ]
                                                              occupancy=[ 0.812 ]
method=[ __globfunc__Z10depth_maskPfS_fii ] gputime=[ 1769.920 ] cputime=[ 2.000 ]
                                                              occupancy=[ 0.812 ]
method=[ __globfunc__Z9transposePfS_ii ] gputime=[ 1598.336 ] cputime=[ 2.000 ]
                                                              occupancy=[ 0.812 ]
method=[ __globfunc__Z12Invariants_XPfS_S_S_S_S_ii ] gputime=[ 5061.920 ]
                           cputime=[ 3.000 ]   occupancy=[ 0.812 ]
method=[ __globfunc__Z7SWater_PfS_S_S_S_fS_S_S_ii ] gputime=[ 10506.752 ]
                           cputime=[ 2.000 ]   occupancy=[ 0.406 ]
method=[ __globfunc__Z9transposePfS_ii ] gputime=[ 1599.776 ] cputime=[ 2.000 ]
                                               occupancy=[ 0.812 ]
method=[ __globfunc__Z9transposePfS_ii ] gputime=[ 1613.472 ] cputime=[ 1.000 ]
                                               occupancy=[ 0.812 ]
method=[ __globfunc__Z9transposePfS_ii ] gputime=[ 1609.056 ] cputime=[ 2.000 ]
                                               occupancy=[ 0.812 ]
```

Cudaprof



Session10 - Device_0 : Profiler table column 'warp serialize' having all zero values is hidden.

Profiler counter 'gld_32/64/128b' required for global memory throughput calculations is not available.

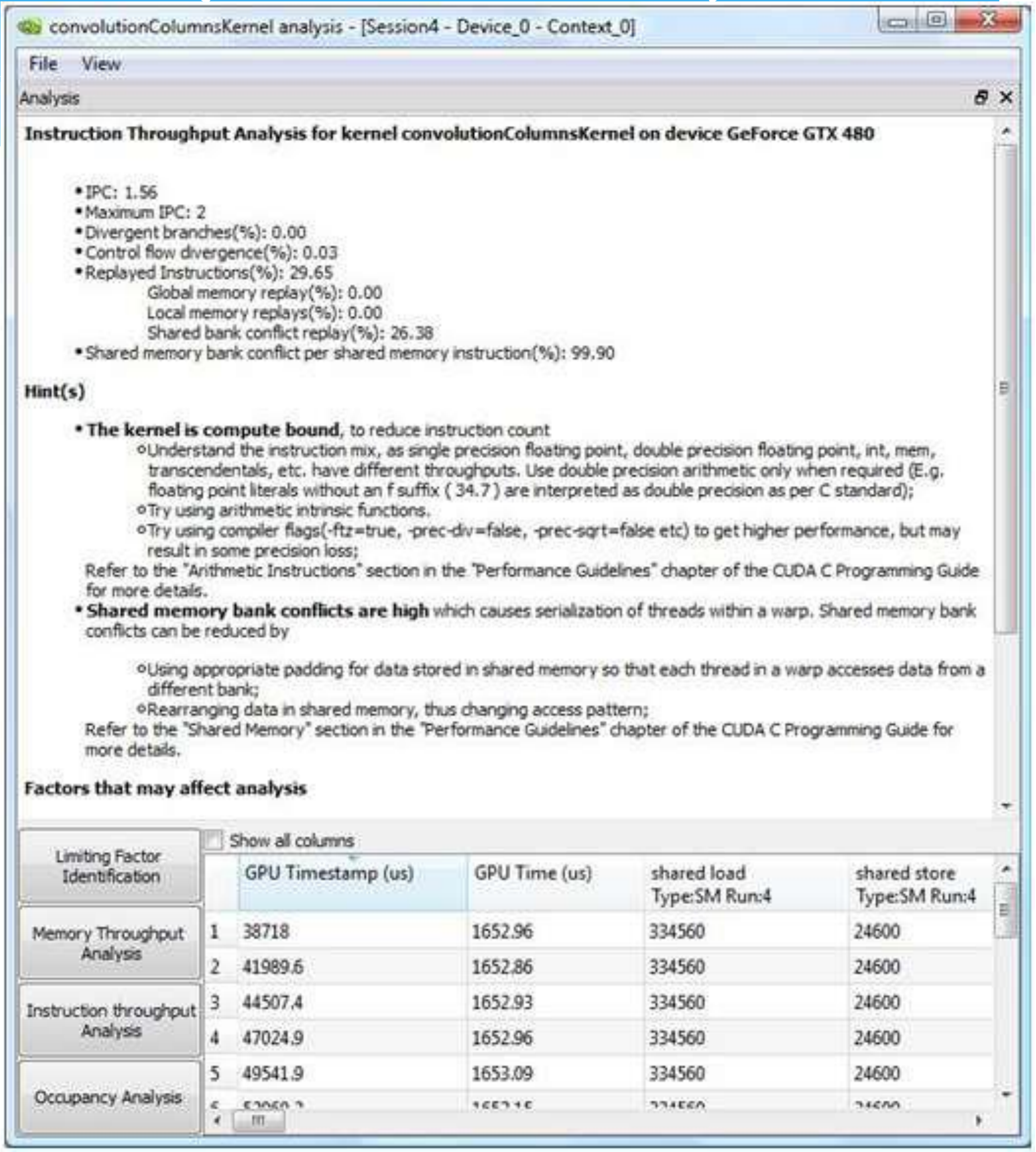
			GPU usec	%GPU time	struction throughp
			,58836e+06	41,63	0,197939
			,7661e+06	27,95	0,0736237
			,88783e+06	14	0,0434338
			,88782e+06	14	0,0434348
			88737	1,39	0,0422327
			5595,9	0,22	0,0483523
7	Inv1	1	28462,2	0,13	0,0452789
8	depth_mask	2	21488,2	0,1	0,0946365

Profiler counter 'gld_32/64/128b' required for global memory throughput calculations is not available.

Profiler counter 'gld_32/64/128b' required for global memory throughput calculations is not available.

Computerprof (CUDA 4.0)

* New user' interface
* Improvements
  * Gives recommendations based on gathered information
  * Automatically detects limitation factors


convolutionColumnsKernel analysis - [Session4 - Device_0 - Context_0]
File
View
Analysis

Instruction Throughput Analysis for kernel convolutionColumnsKernel on device GeForce GTX 480

- IPC: 1.56
- Maximum IPC: 2
- Divergent branches(%): 0.00
- Control flow divergence(%): 0.03
- Replayed Instructions(%): 29.65
  - Global memory replay(%): 0.00
  - Local memory replays(%): 0.00
  - Shared bank conflict replay(%): 26.38
- Shared memory bank conflict per shared memory instruction(%): 99.90

Hint(s)

- **The kernel is compute bound,** to reduce instruction count
  - Understand the instruction mix, as single precision floating point, double precision floating point, int, mem, transcendentals, etc. have different throughputs. Use double precision arithmetic only when required (E.g. floating point literals without an f suffix (34.7) are interpreted as double precision as per C standard);
  - Try using arithmetic intrinsic functions.
  - Try using compiler flags(-ftz=true, -prec-div=false, -prec-sqrt=false etc) to get higher performance, but may result in some precision loss;

  Refer to the "Arithmetic Instructions" section in the "Performance Guidelines" chapter of the CUDA C Programming Guide for more details.
- **Shared memory bank conflicts are high** which causes serialization of threads within a warp. Shared memory bank conflicts can be reduced by
  - Using appropriate padding for data stored in shared memory so that each thread in a warp accesses data from a different bank;
  - Rearranging data in shared memory, thus changing access pattern;

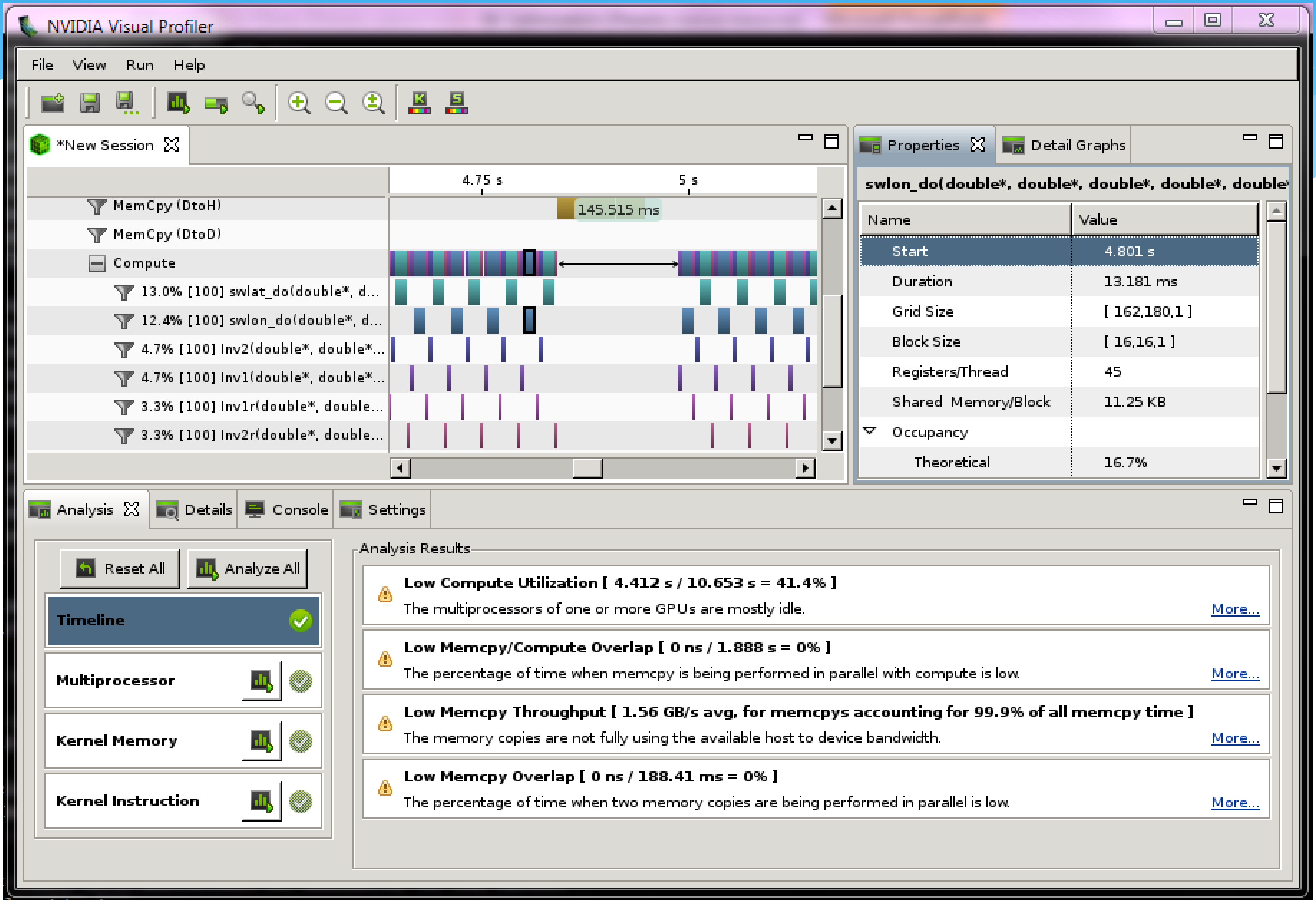
  Refer to the "Shared Memory" section in the "Performance Guidelines" chapter of the CUDA C Programming Guide for more details.

Factors that may affect analysis

Limiting Factor Identification
Memory Throughput Analysis
Instruction throughput Analysis
Occupancy Analysis
Show all columns

	GPU Timestamp (us)	GPU Time (us)	shared load Type:SM Run:4	shared store Type:SM Run:4
1	38718	1652.96	334560	24600
2	41989.6	1652.86	334560	24600
3	44507.4	1652.93	334560	24600
4	47024.9	1652.96	334560	24600
5	49541.9	1653.09	334560	24600
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

NVVP (CUDA 4.1)



swlon_do(double*, double*, double*, double*, double

Name	Value
Start	4.801 s
Duration	13.181 ms
Grid Size	[162,180,1]
Block Size	[16,16,1]
Registers/Thread	45
Shared Memory/Block	11.25 KB
Occupancy	
Theoretical	16.7%

Analysis Results

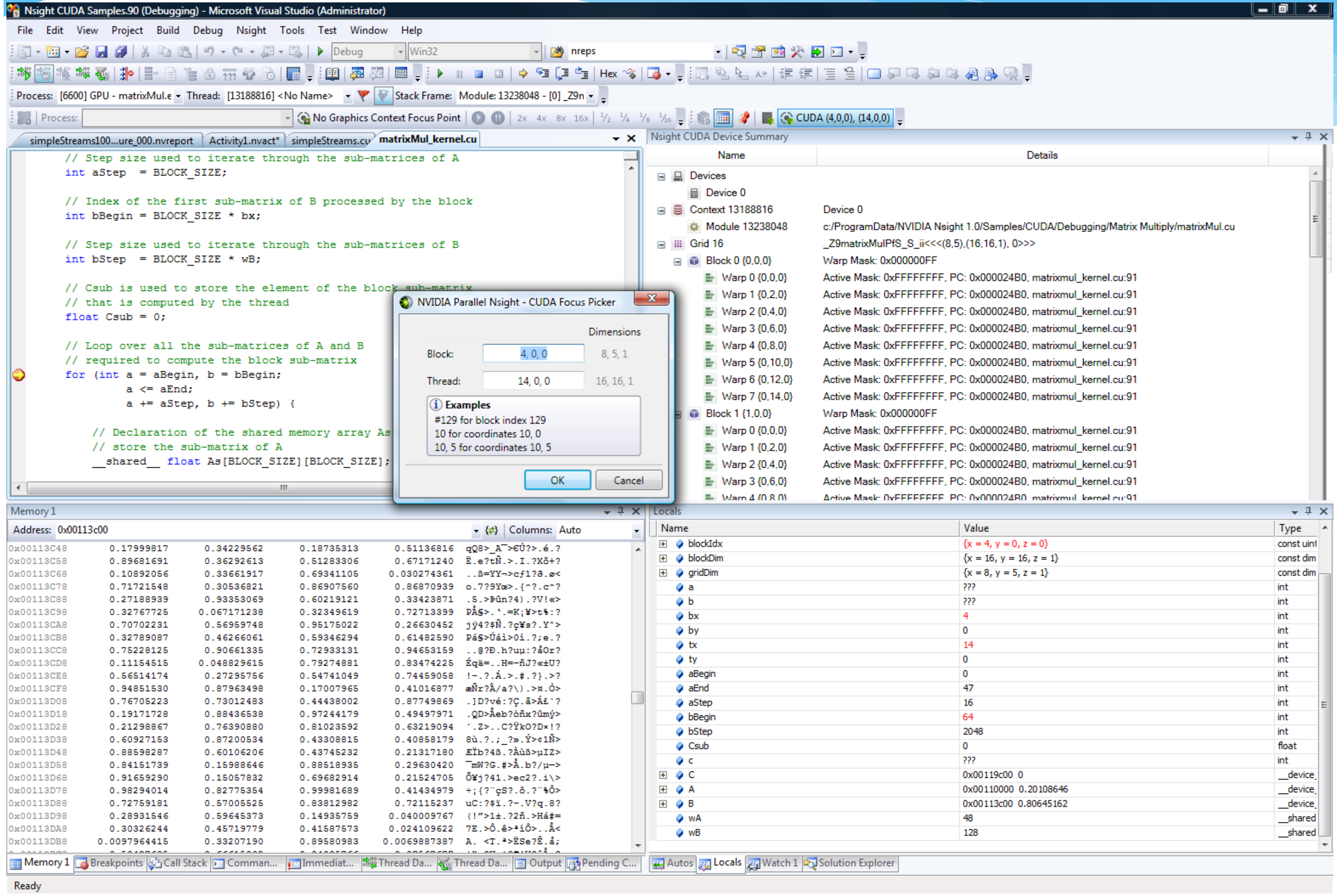
Low Compute Utilization [4.412 s / 10.653 s = 41.4%]
The multiprocessors of one or more GPUs are mostly idle. More...

Low Memcpy/Compute Overlap [0 ns / 1.888 s = 0%]
The percentage of time when memcpy is being performed in parallel with compute is low. More...

Low Memcpy Throughput [1.56 GB/s avg, for memcpys accounting for 99.9% of all memcpy time]
The memory copies are not fully using the available host to device bandwidth. More...

Low Memcpy Overlap [0 ns / 188.41 ms = 0%]
The percentage of time when two memory copies are being performed in parallel is low. More...

NVIDIA Parallel Nsight



Nsight CUDA Samples.90 (Debugging) - Microsoft Visual Studio (Administrator)

Process: [6600] GPU - matrixMul.e Thread: [13188816] <No Name> Stack Frame: Module: 13238048 - [0] _Z9n

No Graphics Context Focus Point   CUDA (4,0,0), (14,0,0)

matrixMul_kernel.cu

```
    // Step size used to iterate through the sub-matrices of A
    int aStep  = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;

    // Step size used to iterate through the sub-matrices of B
    int bStep  = BLOCK_SIZE * wB;

    // Csub is used to store the element of the block sub-matrix
    // that is computed by the thread
    float Csub = 0;

    // Loop over all the sub-matrices of A and B
    // required to compute the block sub-matrix
    for (int a = aBegin, b = bBegin;
             a <= aEnd;
             a += aStep, b += bStep) {

        // Declaration of the shared memory array As
        // store the sub-matrix of A
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
```

NVIDIA Parallel Nsight - CUDA Focus Picker

		Dimensions
Block:	4, 0, 0	8, 5, 1
Thread:	14, 0, 0	16, 16, 1

Examples
#129 for block index 129
10 for coordinates 10, 0
10, 5 for coordinates 10, 5

Nsight CUDA Device Summary

Name	Details
Devices	
Device 0	
Context 13188816	Device 0
Module 13238048	c:/ProgramData/NVIDIA Nsight 1.0/Samples/CUDA/Debugging/Matrix Multiply/matrixMul.cu
Grid 16	_Z9matrixMulPfS_S_ii<<<(8,5),(16,16,1), 0>>>
Block 0 {0,0,0}	Warp Mask: 0x000000FF
Warp 0 {0,0,0}	Active Mask: 0xFFFFFFFF, PC: 0x000024B0, matrixmul_kernel.cu:91
Warp 1 {0,2,0}	Active Mask: 0xFFFFFFFF, PC: 0x000024B0, matrixmul_kernel.cu:91
Warp 2 {0,4,0}	Active Mask: 0xFFFFFFFF, PC: 0x000024B0, matrixmul_kernel.cu:91
Warp 3 {0,6,0}	Active Mask: 0xFFFFFFFF, PC: 0x000024B0, matrixmul_kernel.cu:91
Warp 4 {0,8,0}	Active Mask: 0xFFFFFFFF, PC: 0x000024B0, matrixmul_kernel.cu:91
Warp 5 {0,10,0}	Active Mask: 0xFFFFFFFF, PC: 0x000024B0, matrixmul_kernel.cu:91
Warp 6 {0,12,0}	Active Mask: 0xFFFFFFFF, PC: 0x000024B0, matrixmul_kernel.cu:91
Warp 7 {0,14,0}	Active Mask: 0xFFFFFFFF, PC: 0x000024B0, matrixmul_kernel.cu:91
Block 1 {1,0,0}	Warp Mask: 0x000000FF
Warp 0 {0,0,0}	Active Mask: 0xFFFFFFFF, PC: 0x000024B0, matrixmul_kernel.cu:91
Warp 1 {0,2,0}	Active Mask: 0xFFFFFFFF, PC: 0x000024B0, matrixmul_kernel.cu:91
Warp 2 {0,4,0}	Active Mask: 0xFFFFFFFF, PC: 0x000024B0, matrixmul_kernel.cu:91
Warp 3 {0,6,0}	Active Mask: 0xFFFFFFFF, PC: 0x000024B0, matrixmul_kernel.cu:91

Memory 1

Address: 0x00113c00   Columns: Auto

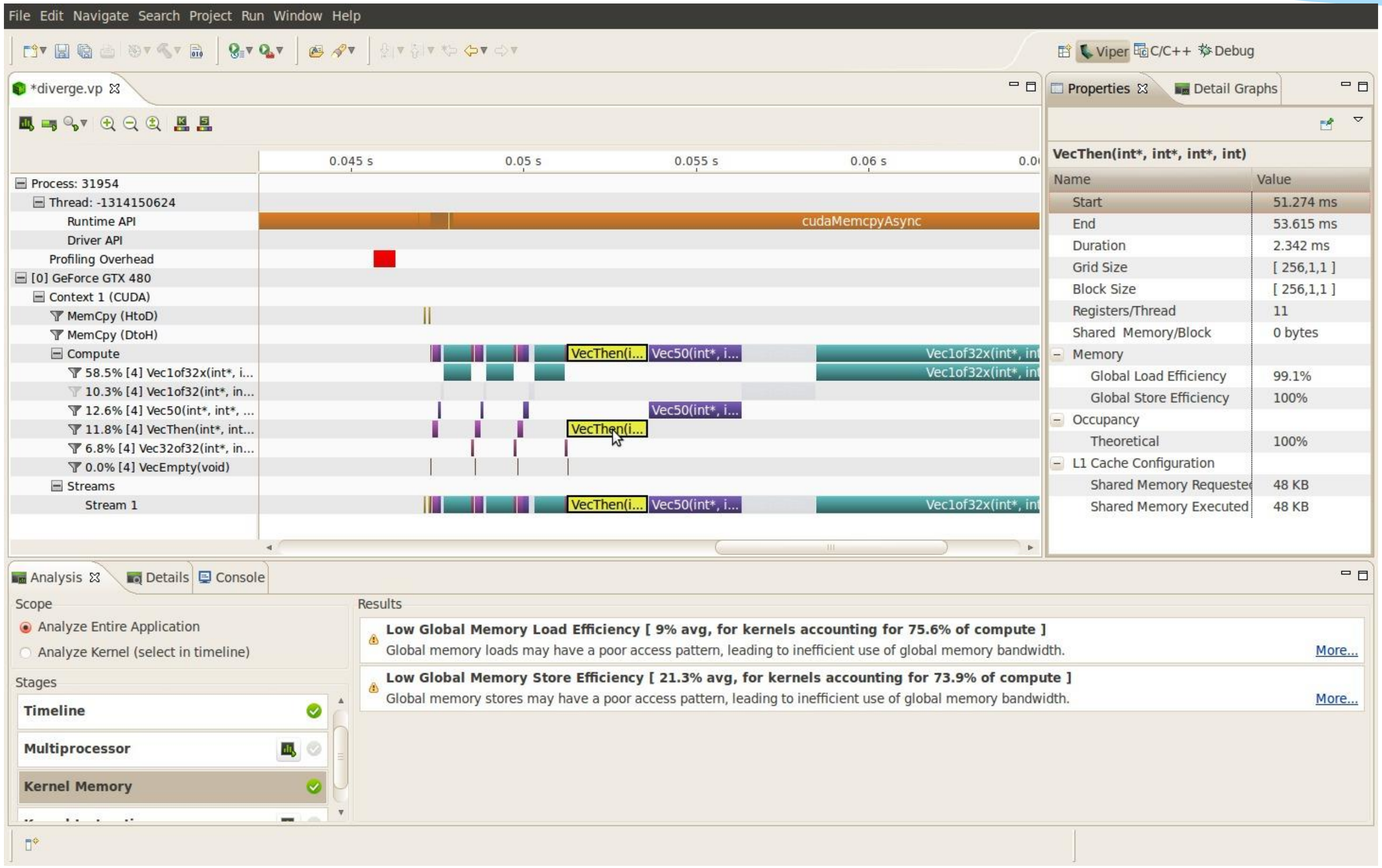
```
0x00113C48      0.17999817     0.34229562     0.18735313     0.51136816  qQ8>_A¯>€Ù?>.é.?
0x00113C58      0.89681691     0.36292613     0.51283306     0.67171240  Ë.e?tÑ.>.I.?Xõ+?
0x00113C68      0.10892056     0.33661917     0.69341105    0.030274361  ..ß=YY¬>cf1?ð.ø<
0x00113C78      0.71721548     0.30536821     0.86907560     0.86870939  o.7?9Yœ>.{^?.c^?
0x00113C88      0.27188939     0.93353069     0.60219121     0.33423871  .5.>Þûn?4).?V!«>
0x00113C98      0.32767725    0.067171238     0.32349619     0.72713399  PÅ§>.'.=K¡¥>t%:?
0x00113CA8      0.70702231     0.56959748     0.95175022     0.26630452  jÿ4?$Ñ.?ç¥s?.Y^>
0x00113CB8      0.32789087     0.46266061     0.59346294     0.61482590  Pá§>Úái>0í.?;e.?
0x00113CC8      0.75228125     0.90661335     0.72933131     0.94653159  ..@?Ð.h?uµ:?åOr?
0x00113CD8      0.11154515    0.048829615     0.79274881     0.83474225  Éqä=..H=-ñJ?«±U?
0x00113CE8      0.56514174     0.27295756     0.54741049     0.74459058  !-.?.Á.>.#.?}.>?
0x00113CF8      0.94851530     0.87963498     0.17007965     0.41016877  æÑr?Â/a?\).>¤.Ò>
0x00113D08      0.76705223     0.73012483     0.44438002     0.87749869  .]D?vé:?Ç.ã>Á£`?
0x00113D18      0.19171728     0.88436538     0.97244179     0.49497971  .QD>Åeb?òñx?ûmý>
0x00113D28      0.21298867     0.76390880     0.81023592     0.63219094  ´.Z>..C?ŸkO?D×!?
0x00113D38      0.60927153     0.87200534     0.43308815     0.40858179  8ù.?.;_?».Ý>¢1Ñ>
0x00113D48      0.88598287     0.60106206     0.43745232     0.21317180  ÆÏb?4ß.?Àùß>µIZ>
0x00113D58      0.84151739     0.15988646     0.88518935     0.29630420  ¯mW?G.#>Å.b?/µ->
0x00113D68      0.91659290     0.15057832     0.69682914     0.21524705  Õ¥j?41.>ec2?.i\>
0x00113D78      0.98294014     0.82775354     0.99981689     0.41434979  ÷¡{?¨çS?.ô.?¨%Ô>
0x00113D88      0.72759181     0.57005525     0.83812982     0.72115237  uC:?$ï.?-.V?q.8?
0x00113D98      0.28931546     0.59645373     0.14935759    0.040009767  (!″>1±.?2ñ.>Há#=
0x00113DA8      0.30326244     0.45719779     0.41587573    0.024109622  7E.>Ô.ê>ªíÔ>..Å<
0x00113DB8    0.0097964415     0.33207190     0.89580983   0.0069887387  A. <T.ª>ËSe?Ê.å;
```

Locals

Name	Value	Type
blockIdx	{x = 4, y = 0, z = 0}	const uint
blockDim	{x = 16, y = 16, z = 1}	const dim
gridDim	{x = 8, y = 5, z = 1}	const dim
a	???	int
b	???	int
bx	4	int
by	0	int
tx	14	int
ty	0	int
aBegin	0	int
aEnd	47	int
aStep	16	int
bBegin	64	int
bStep	2048	int
Csub	0	float
c	???	int
C	0x00119c00 0	__device_
A	0x00110000 0.20108646	__device_
B	0x00113c00 0.80645162	__device_
wA	48	__shared
wB	128	__shared

Ready

NSight Eclipse Edition



VecThen(int*, int*, int*, int)

Name	Value
Start	51.274 ms
End	53.615 ms
Duration	2.342 ms
Grid Size	[256,1,1]
Block Size	[256,1,1]
Registers/Thread	11
Shared Memory/Block	0 bytes
Memory	
Global Load Efficiency	99.1%
Global Store Efficiency	100%
Occupancy	
Theoretical	100%
L1 Cache Configuration	
Shared Memory Requeste	48 KB
Shared Memory Executed	48 KB

Scope

Analyze Entire Application [✓]

Analyze Kernel (select in timeline)

Stages

- Timeline ✓
- Multiprocessor
- Kernel Memory ✓

Results

Low Global Memory Load Efficiency [9% avg, for kernels accounting for 75.6% of compute]
Global memory loads may have a poor access pattern, leading to inefficient use of global memory bandwidth. More...

Low Global Memory Store Efficiency [21.3% avg, for kernels accounting for 73.9% of compute]
Global memory stores may have a poor access pattern, leading to inefficient use of global memory bandwidth. More...

Profiling. CUDA+MPI

```
# Open MPI
if [ ! -z ${OMPI_COMM_WORLD_RANK} ] ; then
rank=${OMPI_COMM_WORLD_RANK}
fi
# MVAPICH
if [ ! -z ${MV2_COMM_WORLD_RANK} ] ; then
rank=${MV2_COMM_WORLD_RANK}
fi
# INTEL
if [ ! -z ${PMI_RANK} ] ; then
rank=${PMI_RANK}
fi
# Set the nvprof command and arguments.
NVPROF="nvprof --output-profile outfile.$rank $nvprof_args"
exec $NVPROF $*
```

Profiling. CUDA+MPI

```
mpirun -np 4 ./nvprof-script.sh --print-api-trace ./a.out
```

```
CUDA 5.5
mpirun -np 2 nvprof --output-profile output.%p ./a.out
```

```
nvprof -i out.14895
======== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 50.26%  3.3920us         2  1.6960us  1.5040us  1.8880us  [CUDA memcpy HtoD]
 29.87%  2.0160us         1  2.0160us  2.0160us  2.0160us  [CUDA memcpy DtoH]
 19.87%  1.3410us         1  1.3410us  1.3410us  1.3410us  AddVectors(...
```

Analysis of profiling results

- Absolute values tells you nothing
- Take into account ratio of counters and how value of a counter changes.
  - gld_incoherent, gst_incoherent moved to zero.

Instructions or memory

- Optimal value instruction:byte for Tesla C2050:
  - ~3.6 : 1, float, ECC on
  - ~4.5 : 1, float, ECC off
- counters
  - 32*instructions_issued (+1 for warp)
  - 128B*(global_store_transaction+ l1_global_load_miss) (+1 one cache line of L1)
- CUDA 4.0+ defines limiters automatically

Instruction analysis

- performance counters (per warp)
  - instructions executed: number of executed instructions
  - instructions issued: including serelization
- Bigger difference – bigger problems. Cache misses.
- Optimal values for the device
  - see Programming Guide or Visual Profiler

Serialization

- Warp divergence
  - Counters: divergent_branch, branch
  - Percent of divergent branches
- Shared bank conflicts
  - Conters:
    - l1_shared_bank_conflict,
    - shared_load, shared_store
- Conflicts are obstacle if both conditions are true:
  - l1_shared_bank_conflict >> (shared_load+ shared_store)
  - l1_shared_bank_conflict >> instructions_issued
- CUDA 4.0+ defines limiters automatically

Register spilling

- Compiler could move values from registers to local memory (spilling)
  - Fermi has 63register per thread max.
  - User could define maximum number of registers
  - Local memory works as global but cached in L1
    - L1 cache miss – request to global memory
  - Compiler flag –ptxas-options=–v shows number of local memory, number of registers for kernels
- Could influence on performance
  - Additional memory traffic
  - Additional instructions
  - Could not de a problem

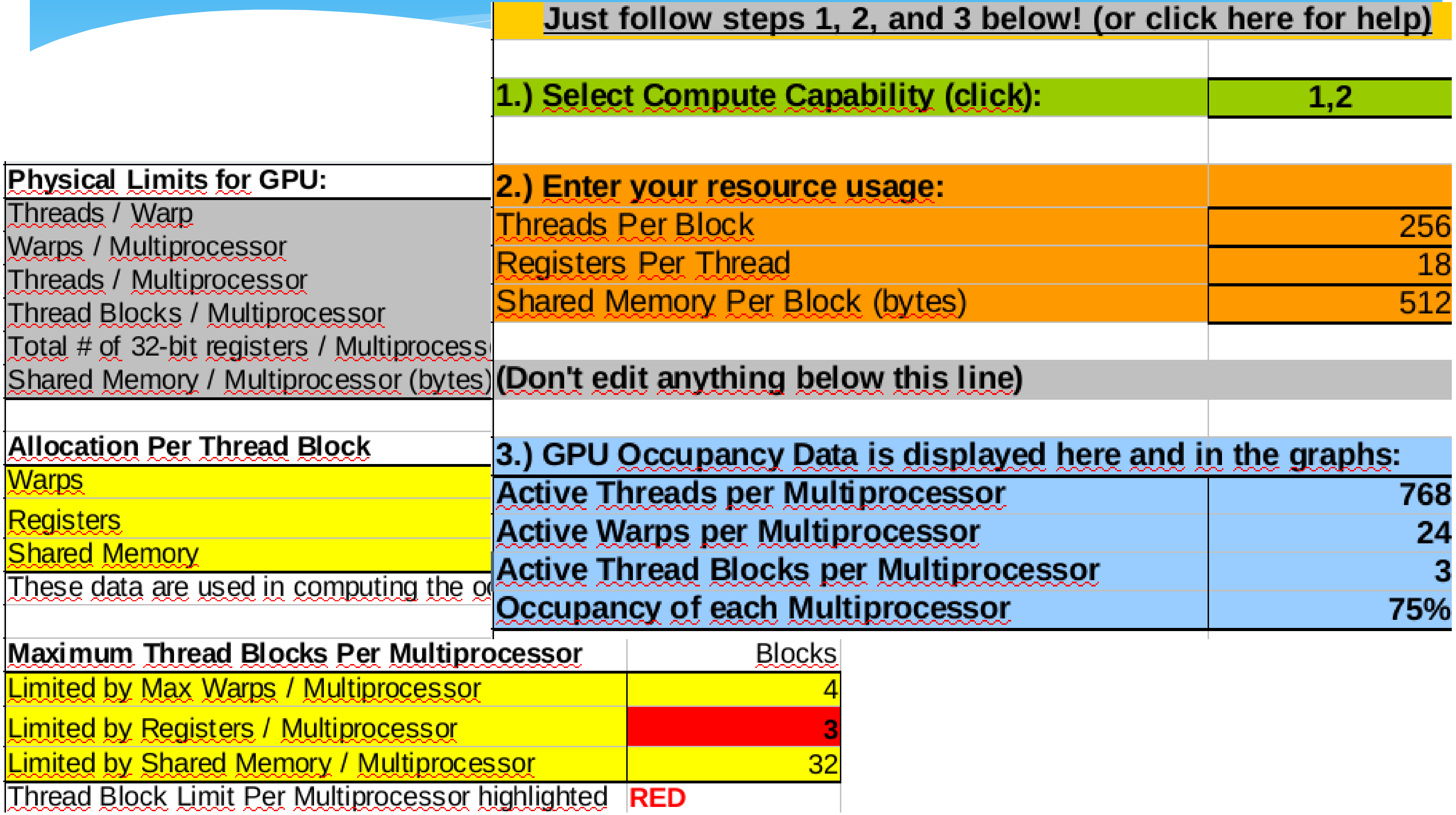
Register spilling

- counters: l1_local_load_hit, l1_local_load_miss
  - Influence on # of instructions
  - Compare with total number of instructions
- Influence on memory band width
  - Compare 2*l1_local_load_miss with global memory operation (read + write)

Occupancy

- Occupancy - the ratio of the number of active warps per multiprocessor to the maximum number of warps that can be active on the multiprocessor at once

CUDA Occupancy calculator



Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	**1,2**

2.) Enter your resource usage:	
Threads Per Block	256
Registers Per Thread	18
Shared Memory Per Block (bytes)	512

(Don't edit anything below this line)

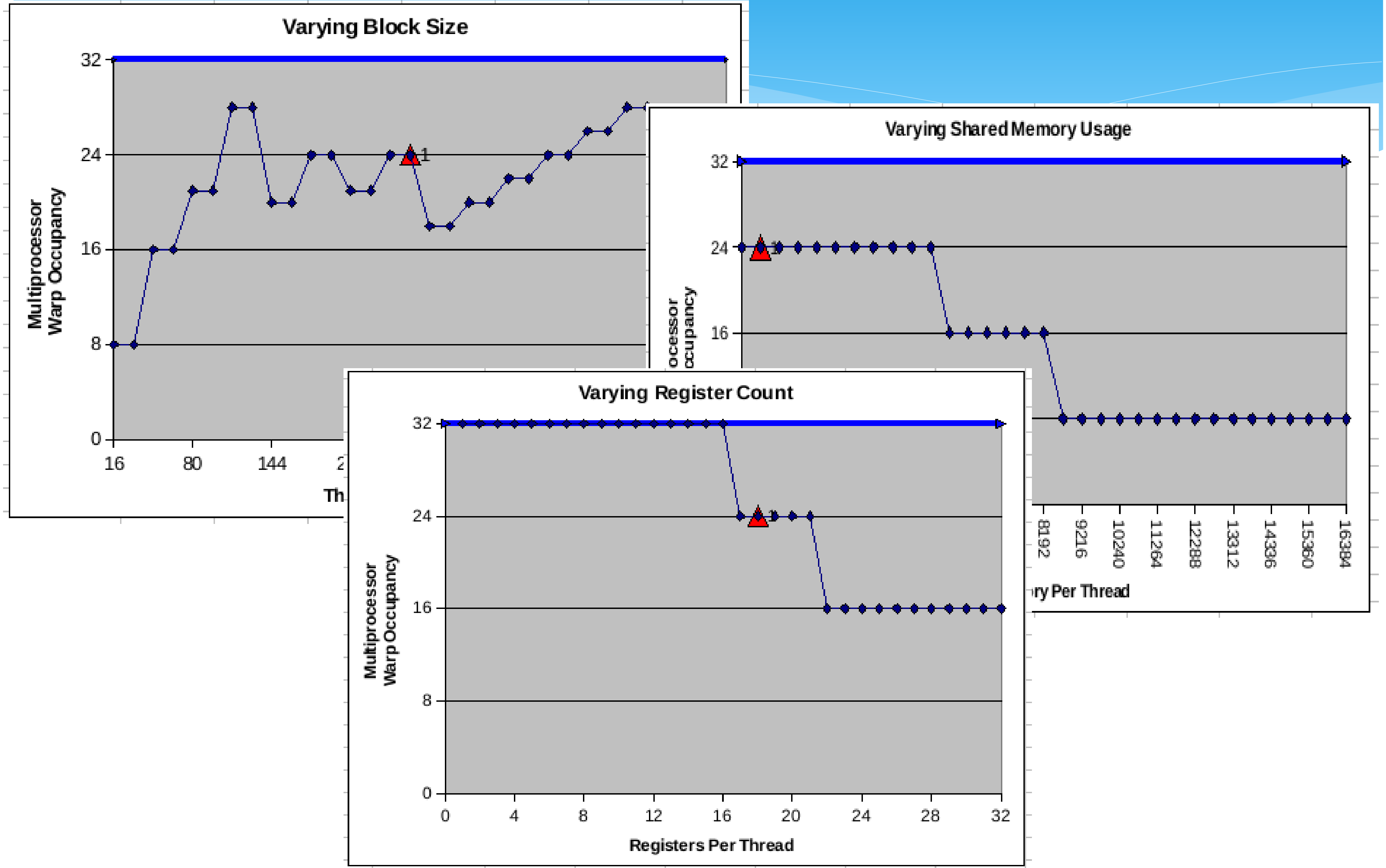
3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	**768**
Active Warps per Multiprocessor	**24**
Active Thread Blocks per Multiprocessor	**3**
Occupancy of each Multiprocessor	**75%**

Physical Limits for GPU:
Threads / Warp
Warps / Multiprocessor
Threads / Multiprocessor
Thread Blocks / Multiprocessor
Total # of 32-bit registers / Multiprocess
Shared Memory / Multiprocessor (bytes)

Allocation Per Thread Block
Warps
Registers
Shared Memory
These data are used in computing the oc

Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Multiprocessor	4
Limited by Registers / Multiprocessor	**3**
Limited by Shared Memory / Multiprocessor	32
Thread Block Limit Per Multiprocessor highlighted	**RED**

CUDA Occupancy calculator



CUDA occupancy API

- Appeared in CUDA 6.5
- <CUDA_Toolkit_Path>/include/cuda_occupancy.h
  - cudaOccupancyMaxActiveBlocksPerMultiprocessor()
  - cudaOccupancyMaxPotentialBlockSize()
  - cudaOccupancyMaxPotentialBlockSizeVariableSMem()

- Example:
  - http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-occupancy-api-simplifies-launch-configuration/

Optimization

- Instruction
  - Load of operands
  - Execute instruction
  - Store result
- Optimization
  - Use ‘fast’ instructions
  - Reduce memory access latency
  - Fully utilize memory band width

Instructions

- Arithmetic operations (cc = 1.2):
  - 4 clocks - FMUL, FADD, FMAD IADD, binary operations, ICMP, MIN, MAX
  - 16 clocks - __log, 1/sqrt, IMUL, 1/(float)x
  - 32 clocks - sqrt, __sin, __cos, __exp
  - 36 clocks FDIV
  - 20 clocks - __fdividef(x, y)
- Compiler options:
  - -ftz=true
  - -prec-div=false
  - -prec-sqrt=false

If conditions. Branching

- Existing of two execution paths in a warp can cause threads to diverge (i.e. to follow different execution paths).
- Minimize flow control instructions (within a warp).
  - Pre-calculations
  - Re-arrangement of threads

Memory access

- Global memory is accessed via 32-, 64-, or 128-byte memory transactions.
- Only the 32-, 64-, or 128-byte segments of device memory that are aligned to their size can be read/written by memory transactions.
- To maximize global memory throughput, it's important to maximize coalescing by:
  - Following the most optimal access patterns,
  - Using data types that meet the size and alignment requirement,
  - Padding data in some cases.

- Use shared, constant memory, and textures.

Memory access

- Use cudaMalloPitch for 2D arrays
- Use CUDA arrays (cudaMallocArray) for 2D , 3D arrays
- Use page-locked memory for Device-Host operations
  - cudaHostAlloc(), cudaFreeHost(), cudaHostRegister(),
- Use textures (massy access pattern)
- Use surfaces (CUDA 4.0)

L1 – caching and size (Fermi)

- possibilities:
  - L1 cache on
    - By default (option -Xptxas –dlcm=ca)
    - Memory transaction - 128 bytes
  - L1 cache off
    - option –Xptxas –dlcm=cg
    - Memory transaction - 32 bytes
- Cache size (L1/SMEM)
  - 16KB L1, 48KB SMEM or 48KB L1, 16KB SMEM
  - Set with CUDA API
- Recommendation:
  - Try all possibilities (CA, CG) x(16, 48)

Concurrent kernel execution

- Available if:
  - Device with computer compatibility greater than 2.0
  - Device prop. concurrentKernels == 1
  - Kernels are from the same context
- Max number of concurrent kernels - 16

Data transfer operations

- D2D memory bandwidth is much higher than H2D and D2H
- It could give benefits to launch kernel with low parallelism than to copy data to Host, process them and return back to Device.
- Use page-locked memory cudaMallocHost()
  - Async operations
  - Overlap to/from data transfer operations
- Overlap data transfer operation with kernel execution

Concurrent data transfer

- Possible if:
  - page-locked memory is used

  - Device computer compatibility >= 2.0
  - Property **asyncEngineCount** = 2

Overlapping data transfer with kernel execution

* Possible if:
    * Device computer compatibility) >= 1.1
    * Property asyncEngineCount > 0
* Not allowed with CUDA Arrays or 2Darrays, allocated with cudaMallocPitch()
* Variable CUDA_LAUNCH_BLOCKING set to “1” block the possibility

Compute capability

- Compute Capability 1.0+
  - Async kernel execution
- Compute Capability 1.1+ (e.g., C1060, cc1.3)
  - One copy engine added. Property **asyncEngineCount**
- Compute Capability 2.0+ (e.g., C2050)
  - Possibility of concurrent kernel execution added (property **concurrentKernels**)
  - Second copy engine added. Property **asyncEngineCount**

GPU to GPU copying

CUDA 3.2

cudaMemcpy(Host, GPU1);
cudaMemcpy(GPU2, Host);

CUDA 4.0

cudaMemcpy(GPU1, GPU2);

Requirements:

Tesla 20xx (Fermi)

64-bit application and OS

Registers spilling

- Change max limit for registers
  - Use __launch_bounds__

```
__global__ void
__launch_bounds__(maxThreadsPerBlock, minBlocksPerMS)
MyKernel(...){...}
```

- turn L1 cache off
- Increase L1 cache size upto 48KB

Registers spilling

```
arom@cuda:~/cuda/edison$ /usr/local/cuda/bin/nvcc -
gencode=arch=compute_20,code=\"sm_20,compute_20\" -m32 --compiler-
options -fno-strict-aliasing  -I. -I /usr/local/cuda/include -I
/usr/local/cudasdk/C/common/inc -I/usr/local/cudasdk/shared/inc -
DUNIX -O2 -g --ptxas-options=-v,-abi=no  -c most_cuda.cu -
maxrregcount=32
ptxas info    : Compiling entry function
'_Z8swlon_doPdS_S_S_S_S_jiiddi' for 'sm_20'
ptxas info    : Used 32 registers, 52+0 bytes lmem, 11520+0 bytes
smem, 92 bytes cmem[0], 32 bytes cmem[14], 48 bytes cmem[16]
ptxas info    : Compiling entry function
'_Z4Inv1PdS_S_S_S_S_S_jiid' for 'sm_20'
ptxas info    : Used 19 registers, 80 bytes cmem[0], 32 bytes
cmem[14], 16 bytes cmem[16]
```