CUDA Optimization with NVIDIA Tools

Julien Demouth, NVIDIA

What Will You Learn?

- An iterative method to optimize your GPU code
- A way to conduct that method with Nvidia Tools

What Does the Application Do ?

- It does not matter !!!
- We care about memory accesses, instructions, latency, …

- Companion code:

  https://github.com/jdemouth/nsight-gtc2013

C:\Windows\system32\cmd.exe

```
##############################################################################
**                        B I C G S T A B   S O L V E R                     **
##############################################################################

** DEVICE      : Tesla K20c (ECC: OFF)                                      **

##############################################################################

** SYSTEM      : res/matrix.inp                                              **

##############################################################################

** INIT. RESID.:  [   1.212971e-001   0.000000e+000   0.000000e+000   1.243311e-001 ]      **

##############################################################################

** ITERATION  0:  [   5.009870e-002   2.509095e-003   2.442529e-003   1.381766e-003 ]      **
** ITERATION  1:  [   6.322283e-002   3.624363e-003   3.439345e-003   1.459566e-003 ]      **
** ITERATION  2:  [   1.944435e-002   3.175072e-004   3.108480e-004   4.101967e-004 ]      **
** ITERATION  3:  [   1.179491e-002   9.633129e-005   9.554327e-005   2.494020e-004 ]      **
** ITERATION  4:  [   1.517741e-002   1.351845e-004   1.323939e-004   3.272856e-004 ]      **
** ITERATION  5:  [   2.840113e-002   2.370584e-004   2.333890e-004   6.397188e-004 ]      **
** ITERATION  6:  [   8.465301e-003   9.483242e-005   9.256901e-005   1.726512e-004 ]      **
** ITERATION  7:  [   2.497275e-003   2.221739e-005   2.213925e-005   6.087546e-005 ]      **
** ITERATION  8:  [   3.931372e-003   3.762042e-005   3.804746e-005   9.449076e-005 ]      **
** ITERATION  9:  [   1.004664e-003   7.813901e-006   7.722009e-006   2.470211e-005 ]      **
** ITERATION 10:  [   1.348178e-003   1.450667e-005   1.451499e-005   3.084324e-005 ]      **
** ITERATION 11:  [   3.147213e-004   3.016084e-006   2.968251e-006   7.588855e-006 ]      **
** ITERATION 12:  [   2.560259e-004   2.530426e-006   2.474577e-006   6.138979e-006 ]      **
** ITERATION 13:  [   1.941811e-004   2.010254e-006   1.992610e-006   4.670605e-006 ]      **
** ITERATION 14:  [   1.344858e-004   1.313841e-006   1.286352e-006   3.325935e-006 ]      **
** ITERATION 15:  [   2.946048e-004   3.318294e-006   3.216173e-006   7.020198e-006 ]      **
** ITERATION 16:  [   1.254350e-004   1.372731e-006   1.317983e-006   3.036414e-006 ]      **

##############################################################################

** FINAL RESID.:  [   2.529559e-005   2.054403e-007   1.903810e-007   6.569666e-007 ]      **

##############################################################################

** ELAPSED TIME:   104.723ms                                                 **

##############################################################################

Press any key to continue . . .
```

Our Method

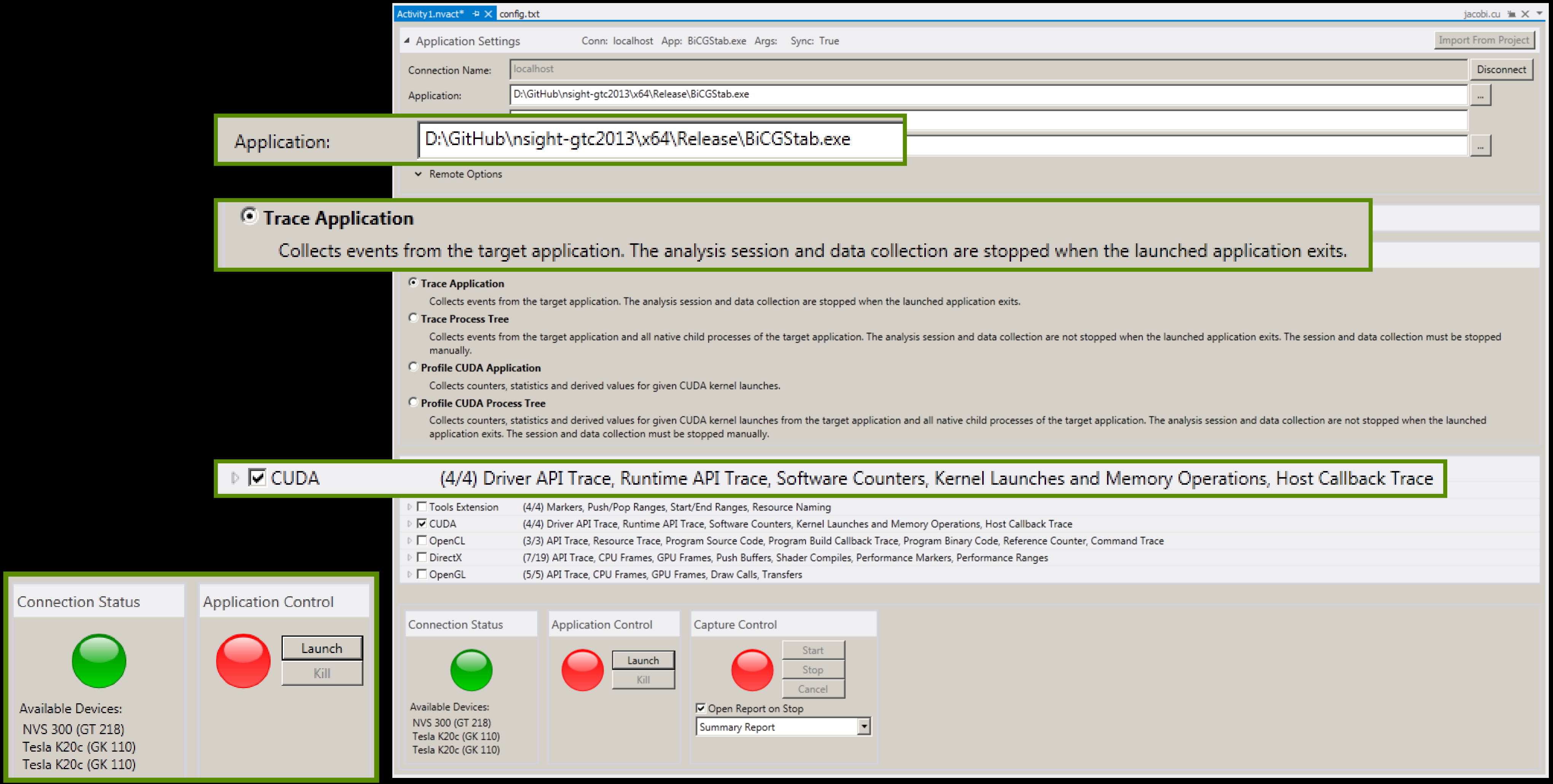
- Trace the application
- Identify the hot spot and profile it
- Identify the performance limiter
  - Memory Bandwidth
  - Instruction Throughput
  - Latency
- Optimize the code
- Iterate

Our Environment

- We use
  - Nvidia Tesla K20c (GK110, SM 3.5), ECC OFF,
  - Microsoft Windows 7 x64,
  - Microsoft Visual Studio 2012,
  - CUDA 5.5,
  - Nvidia Nsight 3.1.

ITERATION 1

Trace the Application (Nsight VSE)



CUDA Launch Summary (Nsight VSE)

BiCGStab131007_000...apture_000.nvreport Activity1.nvact* config.txt

CUDA Launch Summary

Filter

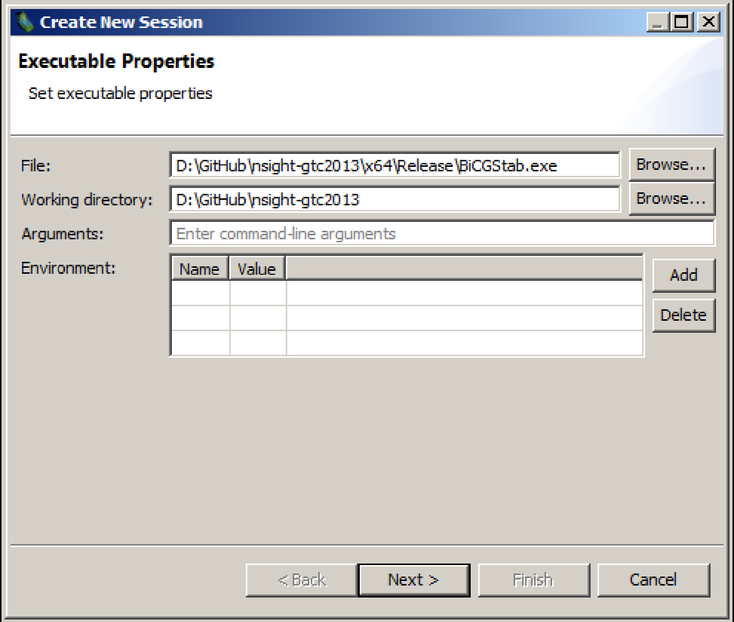
Drag a column header and drop it here to group by that column

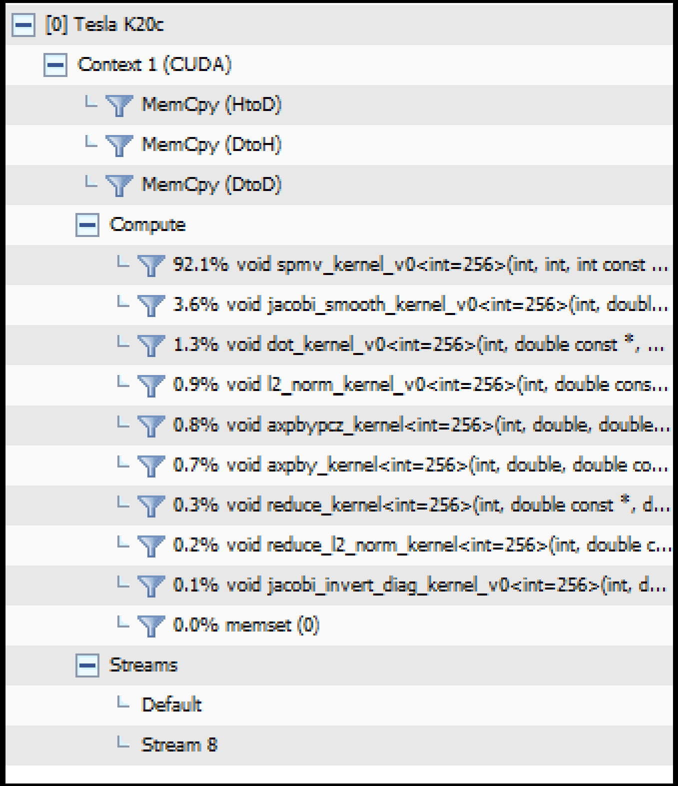
	Function Name	Module ID	Function ID	Count	Device %	Device Time (μs)
1	spmv_kernel_v0<int=256>	43	4	71	9.70	81,191.472
2	jacobi_smooth_kernel_v0<int=256>	46	4	35	0.37	3,110.917
3	dot_kernel_v0<int=256>	45	1	70	0.13	1,088.736
4	l2_norm_kernel_v0<int=256>	44	1	36	0.10	824.494
5	axpbypcz_kernel<int=256>	45	5	34	0.09	719.845
6	axpby_kernel<int=256>	45	4	37	0.07	597.463
7	reduce_kernel<int=256>	45	3	70	0.03	256.633
8	reduce_l2_norm_kernel<int=256>	44	2	36	0.02	179.251
9	jacobi_invert_diag_kernel_v0<int=256>	46	1	1	0.01	108.554

- spmv_kernel_v0 is a hot spot, let's start here!!!

Kernel	Time	Speedup
Original version	104.72ms	

Trace the Application (NVVP)



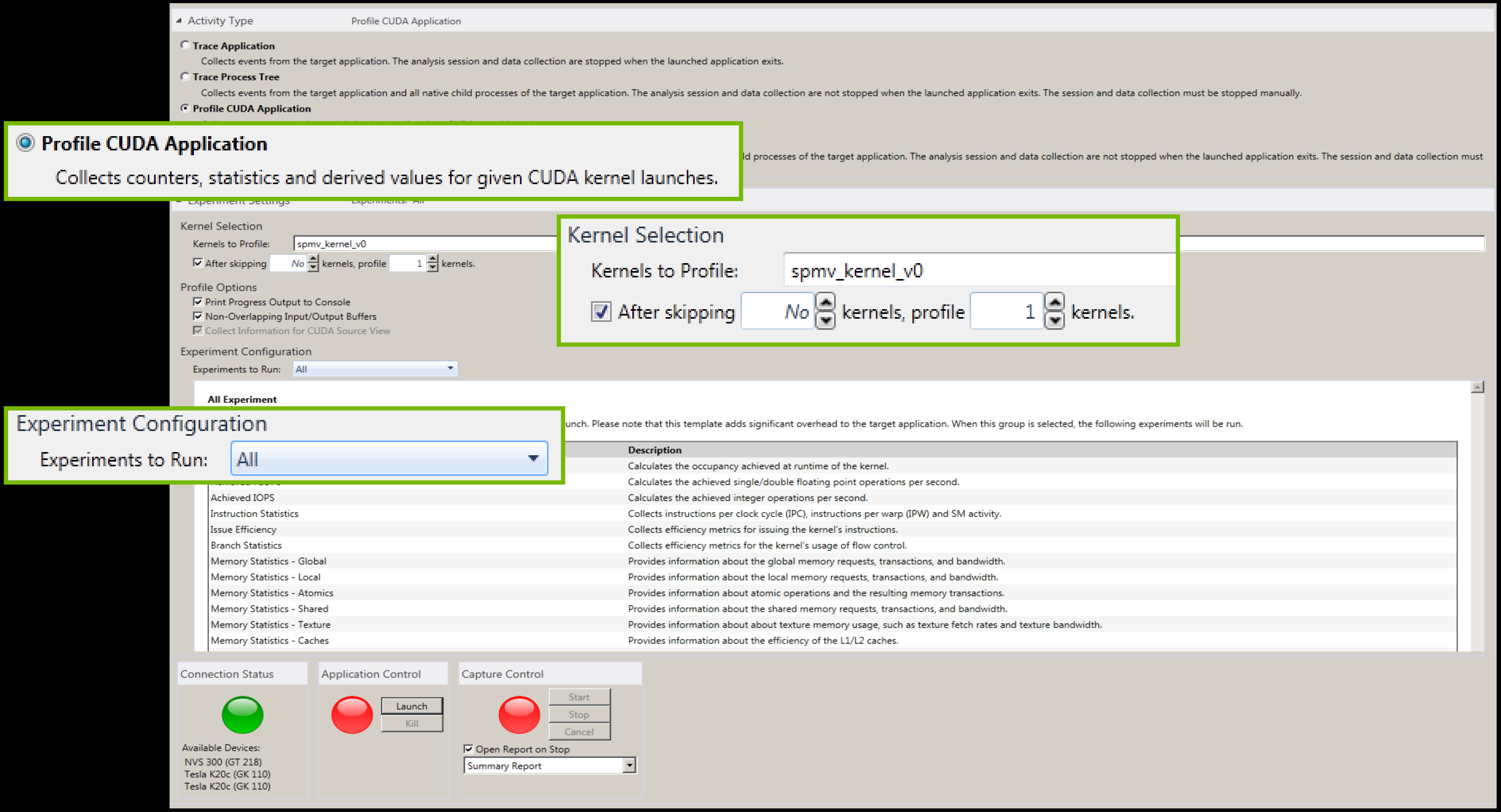


Trace the Application (nvprof)

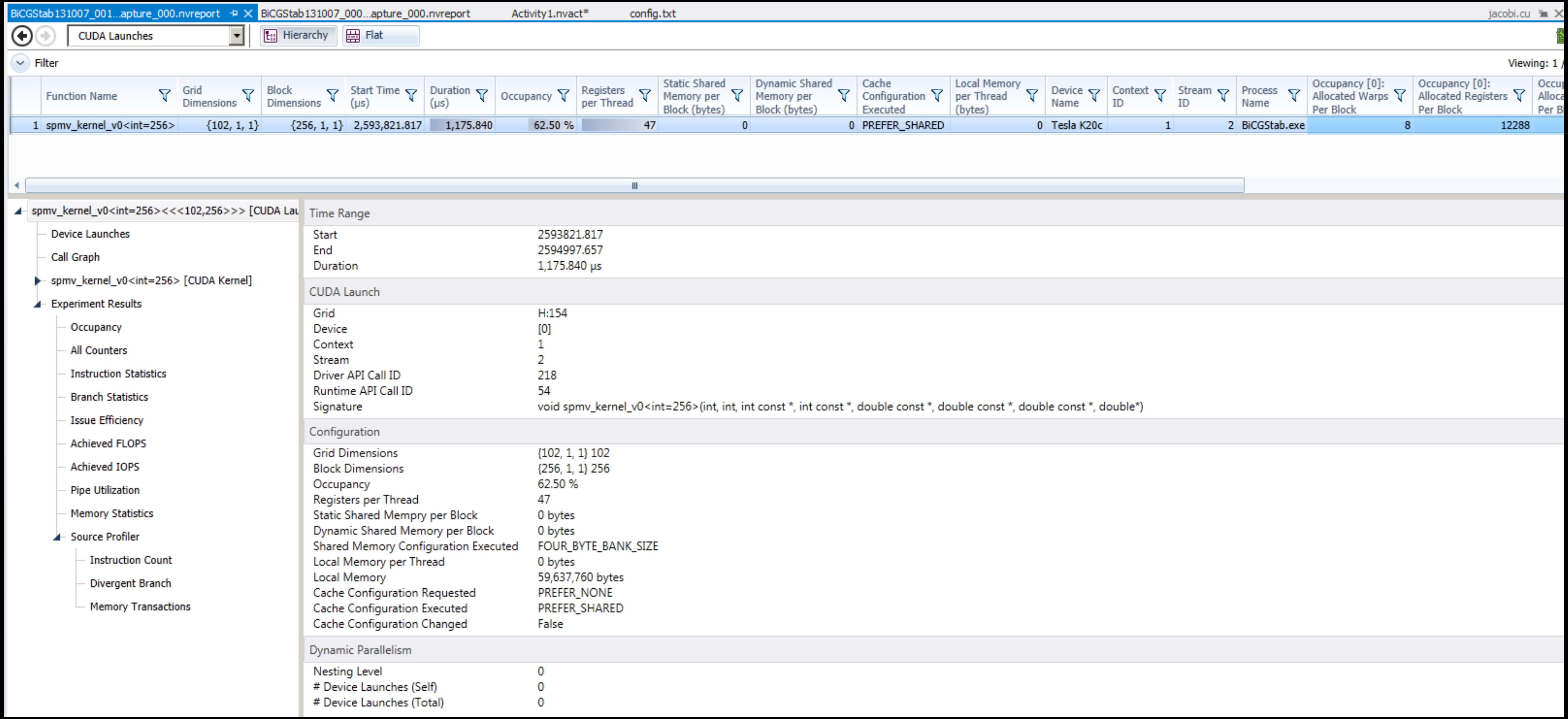
```
D:\GitHub\nsight-gtc2013 [master +8 ~2 -0 !]>
D:\GitHub\nsight-gtc2013 [master +8 ~2 -0 !]> nvprof .\x64\Release\BiCGStab.exe
```

```
==8104== Profiling application: .\x64\Release\BiCGStab.exe
==8104== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 85.03%  81.366ms        71  1.1460ms  1.0754ms  1.1917ms  void spmv_kernel_v0<int=256>(int, int, int const *, int const *, double const *,
double const *, double const *, double*)
  7.49%  7.1668ms         4  1.7917ms  34.208us  6.7015ms  [CUDA memcpy HtoD]
  3.24%  3.0994ms        35  88.554us  86.304us  97.376us  void jacobi_smooth_kernel_v0<int=256>(int, double, double const *, double const *
, double const *, double*)
  1.13%  1.0819ms        70  15.455us  11.072us  17.504us  void dot_kernel_v0<int=256>(int, double const *, double const *, double*)
  0.86%  823.17us        36  22.865us  21.568us  25.024us  void l2_norm_kernel_v0<int=256>(int, double const *, double*)
  0.74%  709.98us        34  20.881us  20.288us  21.600us  void axpbypcz_kernel<int=256>(int, double, double const *, double, double const *
, double, double const *, double*)
  0.63%  606.75us        37  16.398us  15.072us  17.184us  void axpby_kernel<int=256>(int, double, double const *, double, double const *, d
ouble*)
  0.27%  262.88us        70  3.7550us  3.1040us  4.4480us  void reduce_kernel<int=256>(int, double const *, double*)
  0.26%  247.26us       106  2.3320us  2.1120us  3.7760us  [CUDA memcpy DtoH]
  0.18%  174.08us        36  4.8350us  4.2880us  6.1440us  void reduce_l2_norm_kernel<int=256>(int, double const *, double*)
  0.12%  112.00us         1  112.00us  112.00us  112.00us  void jacobi_invert_diag_kernel_v0<int=256>(int, double const *, double*)
  0.04%  34.880us        36     968ns     672ns  10.112us  [CUDA memset]
  0.00%  1.4400us         2     720ns     704ns     736ns  [CUDA memcpy DtoD]
D:\GitHub\nsight-gtc2013 [master +8 ~2 -0 !]>
```

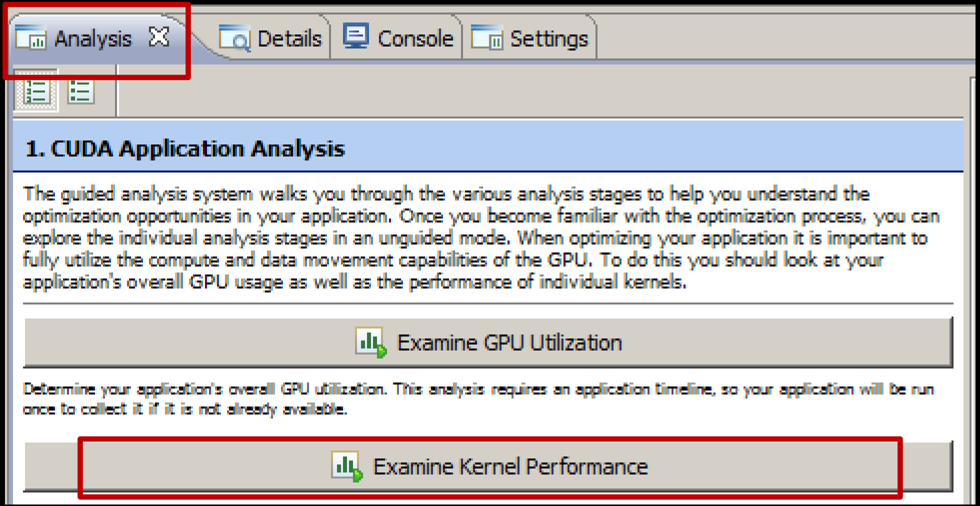
Profile the Most Expensive Kernel (Nsight VSE)

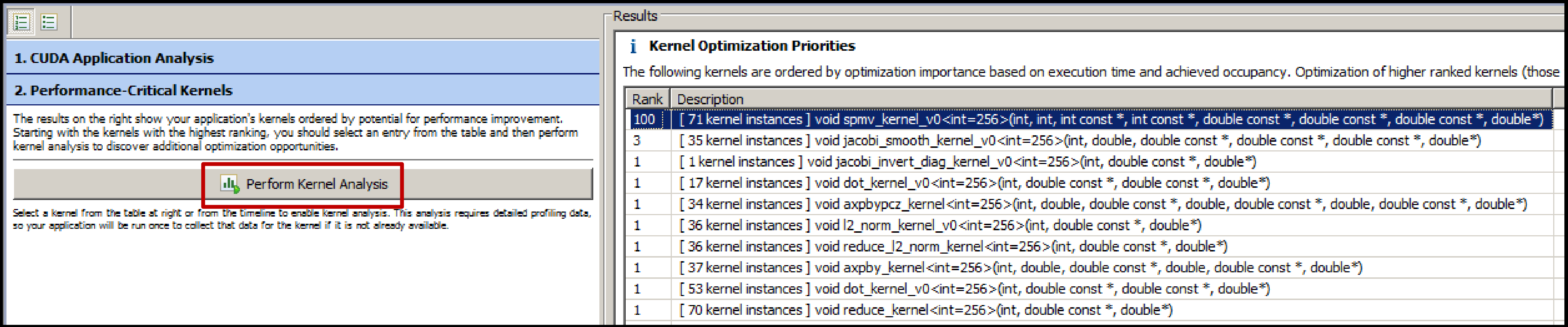


CUDA Launches (Nsight VSE)



Profile the Most Expensive Kernel (NVVP)





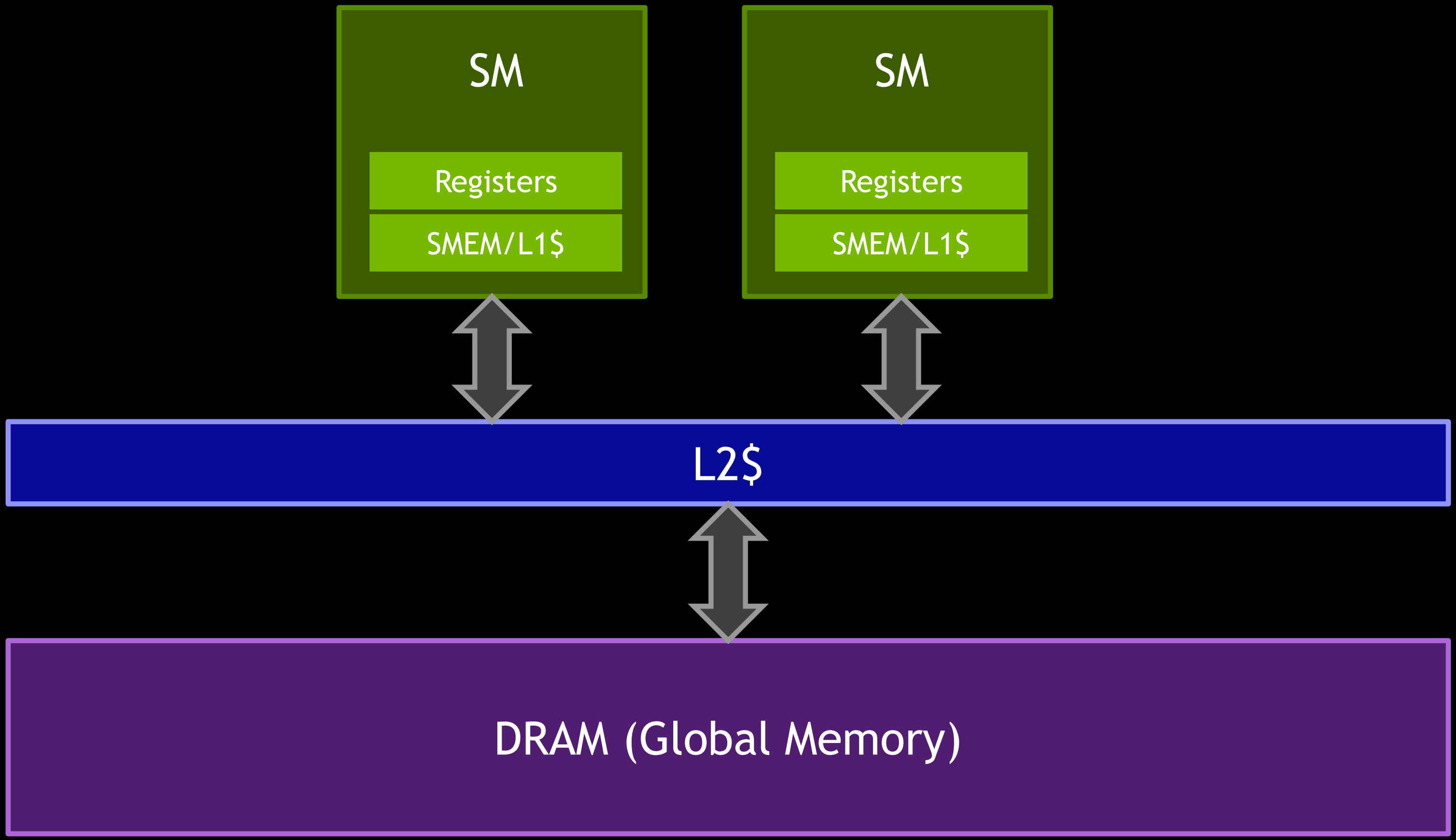
Profile the Most Expensive Kernel (nvprof)

```
> nvprof --kernels "::spmv_kernel_v0:" --metrics issue_slot_utilization .\x64\Release\BiCGStab.exe

> nvprof --query-metrics

> nvprof --kernels "::spmv_kernel_v0:" --events active_cycles .\x64\Release\BiCGStab.exe

> nvprof --query-events
```

Identify the Main Limiter

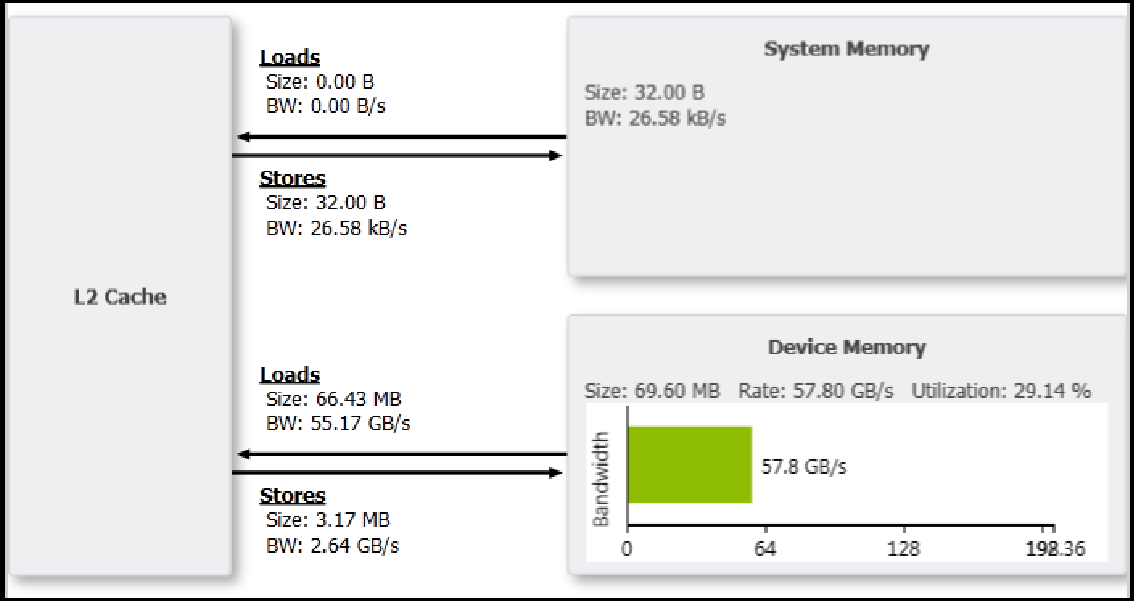
- Is it limited by the memory bandwidth ?
- Is it limited by the instruction throughput ?
- Is it limited by latency ?

Memory Bandwidth



Memory Bandwidth

- Utilization of DRAM Bandwidth: 29.14%



- We are not limited by the memory bandwidth (< 70-80%)

Memory Bandwidth (nvprof)

- Utilization of DRAM Bandwidth: 31.86%

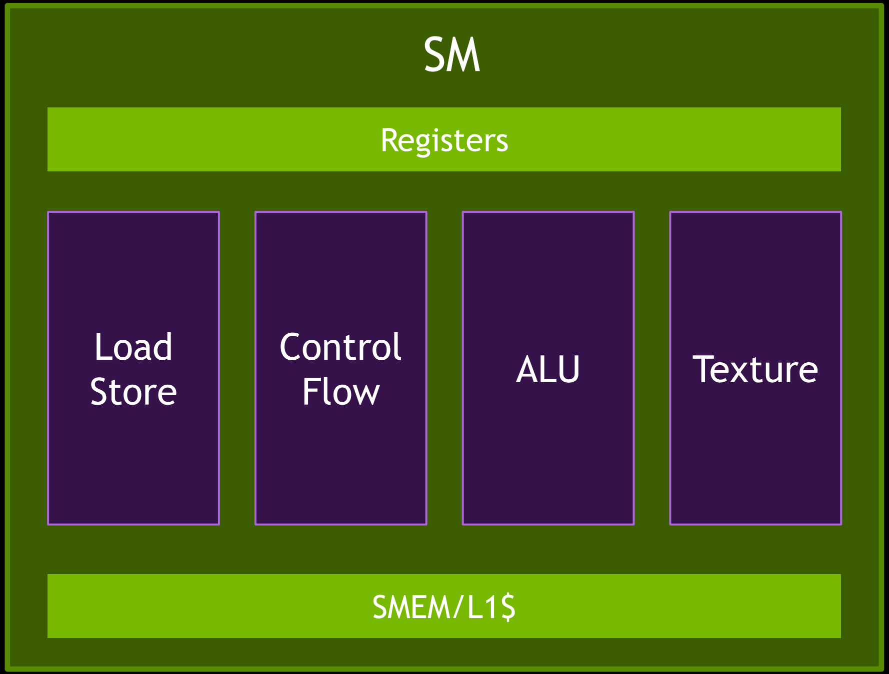
```
> nvprof --kernels "::spmv_kernel_v0:" --metrics "dram_read_throughput,dram_write_throughput" .\x64\Release\BiCGStab.exe
```

	Read	Write	Total
Bandwidth (GB/s)	60.77	2.38	63.15
Utilization (%)	29.22	1.14	30.36

Peak BW (K20c): 208GB/s

- We are not limited by the memory bandwidth

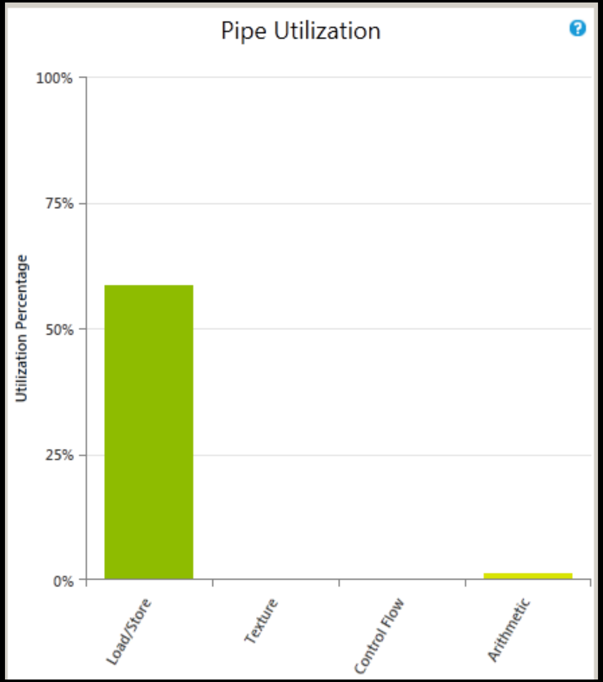
Instruction Throughput



- Instructions go to the pipes
- Issue 1 or 2 instructions every cycle
- We cannot if a pipe is saturated

Instruction Throughput

- All pipes are underutilized: <70-75%



- We are not limited by instruction throughput

Instruction Throughput

- All pipes have Low/Mid utilization

```
|> nvprof --kernels "::spmv_kernel_v0:" --metrics "ldst_fu_utilization,cf_fu_utilization" .\x64\Release\BiCGStab.exe
 > nvprof --kernels "::spmv_kernel_v0:" --metrics "alu_fu_utilization,tex_fu_utilization" .\x64\Release\BiCGStab.exe
```

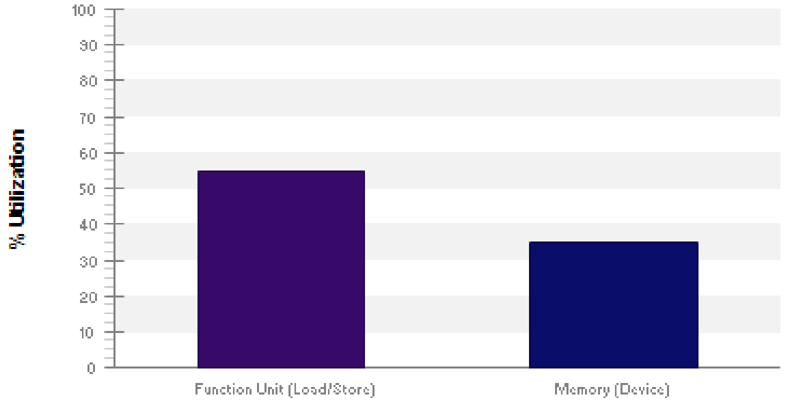
	Load/Store	Control Flow	ALU	Texture
Utilization	Mid	Low	Low	Idle

- We are not limited by instruction throughput

Guided Analysis (Nvvp)

Kernel Performance Is Bound By Instruction And Memory Latency

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "Tesla K20c". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.



Latency

- First two things to check:
  - Occupancy
  - Memory accesses (coalesced/uncoalesced accesses)
- Other things to check (if needed):
  - Control flow efficiency (branching, idle threads)
  - Divergence
  - Bank conflicts in shared memory

Latency (Occupancy)

- Occupancy: 55.98% Achieved / 62.50% Theoretical

Occupancy Per SM

Active Blocks		5	16
Active Warps	35.83	40	64
Active Threads		1280	2048
Occupancy	55.98 %	62.50 %	100.00 %

```
> nvprof --kernels "::spmv_kernel_v0:" --metrics "achieved_occupancy" .\x64\Release\BiCGStab.exe
```

- It’s not too high but not too low: Hard to say

Latency (Occupancy)

- Guided Analysis (Nvvp):

GPU Utilization May Be Limited By Register Usage

Theoretical occupancy is less than 100% but is large enough that increasing occupancy may not improve performance. You can attempt the following optimization to increase the number of warps on each SM but it may not lead to increased performance.

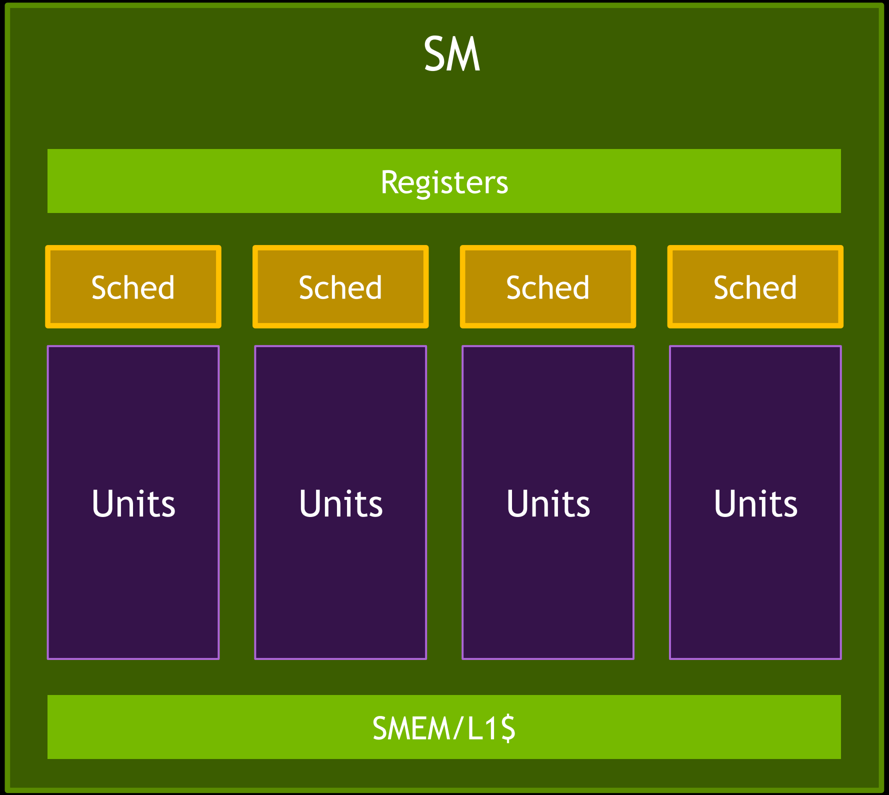
The kernel uses 47 registers for each thread (12032 registers for each block). This register usage is likely preventing the kernel from fully utilizing the GPU. Device "Tesla K20c" provides up to 65536 registers for each block. Because the kernel uses 12032 registers for each block each SM is limited to simultaneously executing 5 blocks (40 warps). Chart "Varying Register Count" below shows how changing register usage will change the number of blocks that can execute on each SM.

Optimization: Use the -maxrregcount flag or the __launch_bounds__ qualifier to decrease the number of registers used by each thread. This will increase the number of blocks that can execute on each SM.

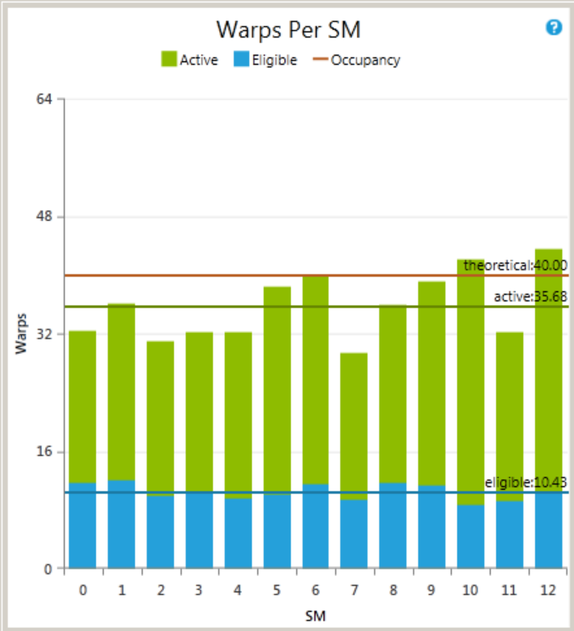
More...

- *"Theoretical occupancy is less than 100% but is large enough that increasing occupancy may not improve performance"*

Latency (Occupancy)



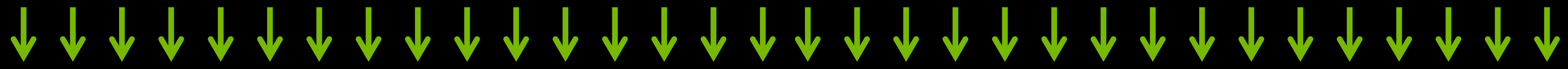
- Eligible Warps per Active Cycle: 10.43



- Occupancy is not an issue (> 4)

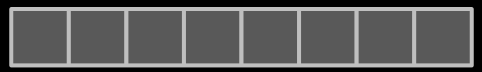
Memory Transactions

- Warps of threads (32 threads)



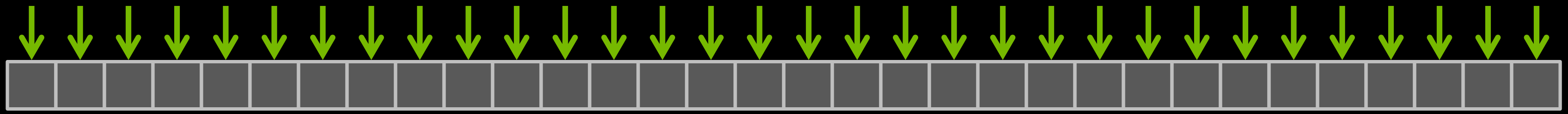
- L1 transaction: 128B – Alignment: 128B (0, 128, 256, …)

- L2 transaction: 32B – Alignment: 32B (0, 32, 64, 96, …)

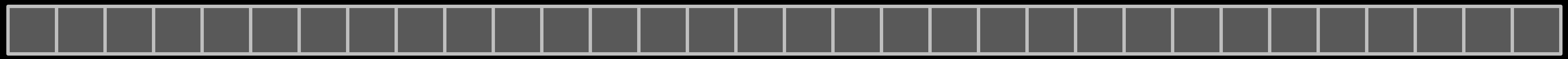


Memory Transactions (fp32)

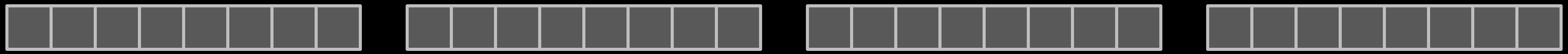
- Ideal case: 32 aligned and consecutive fp32 numbers



- 1x L1 transaction: 128B needed / 128B transferred

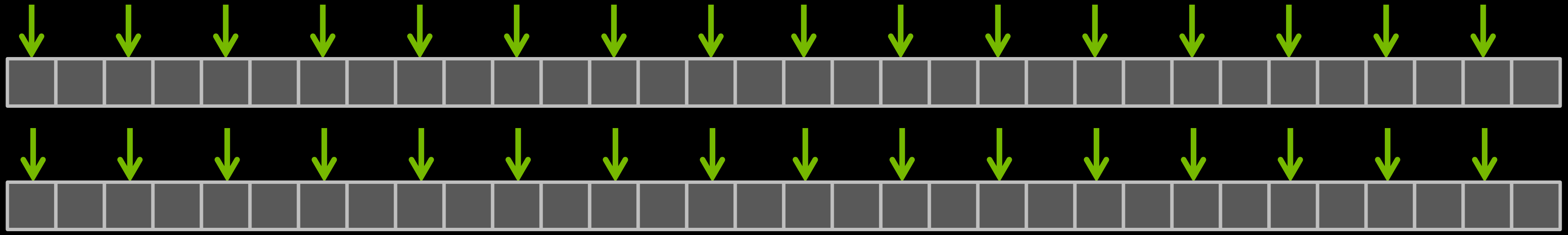


- 4x L2 transactions: 128B needed / 128B transferred



Memory Transactions (fp64)

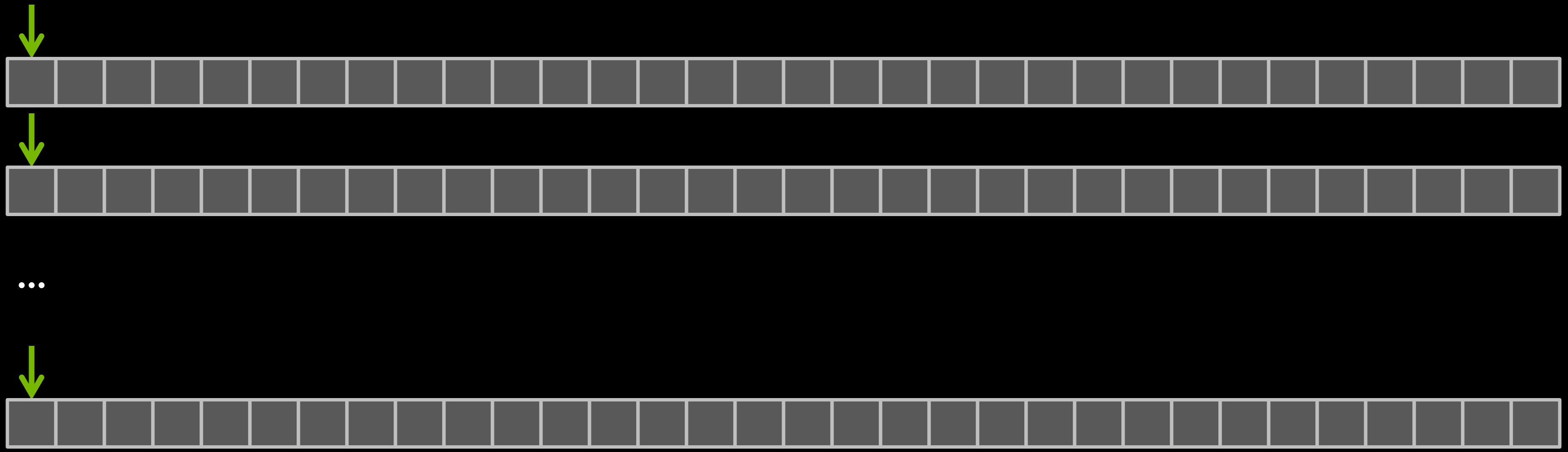
- Ideal case: 32 aligned and consecutive fp64 numbers



- 2x L1 transactions: 256B needed / 256B transferred
- 8x L2 transactions: 256B needed / 256B transferred

Memory Transactions (fp64)

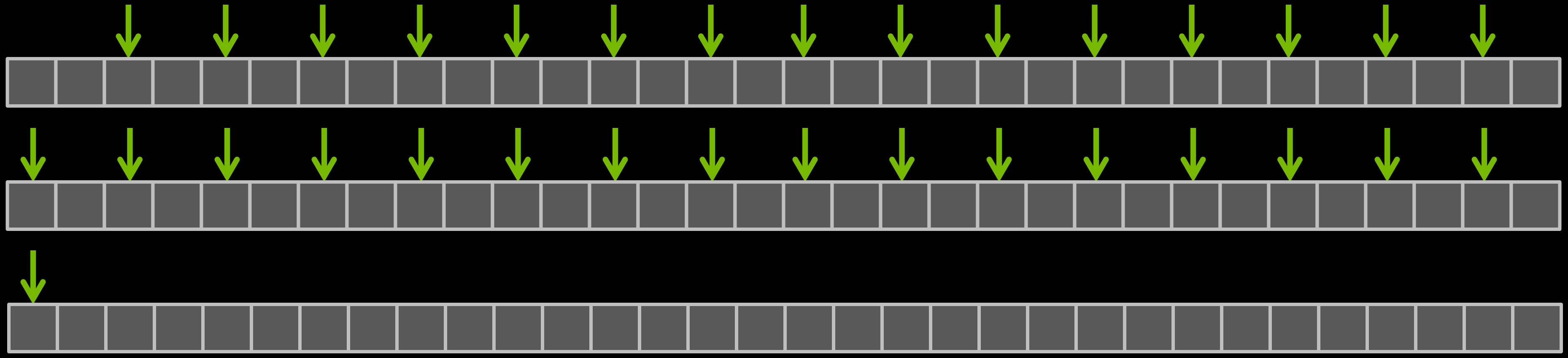
- Worst case: 32 fp64 with a stride of 128B (16x fp64)



- 32x L1 transactions: 256B needed / 32x128B transferred
- 32x L2 transactions: 256B needed / 32x32B transferred

Memory Transactions (fp64)

- Misaligned: 32 fp64



- 3x L1 transactions: 256B needed / 384B transferred
- 9x L2 transactions: 256B needed / 288B transferred

Memory Transactions

- Broadcast: 1 fp64

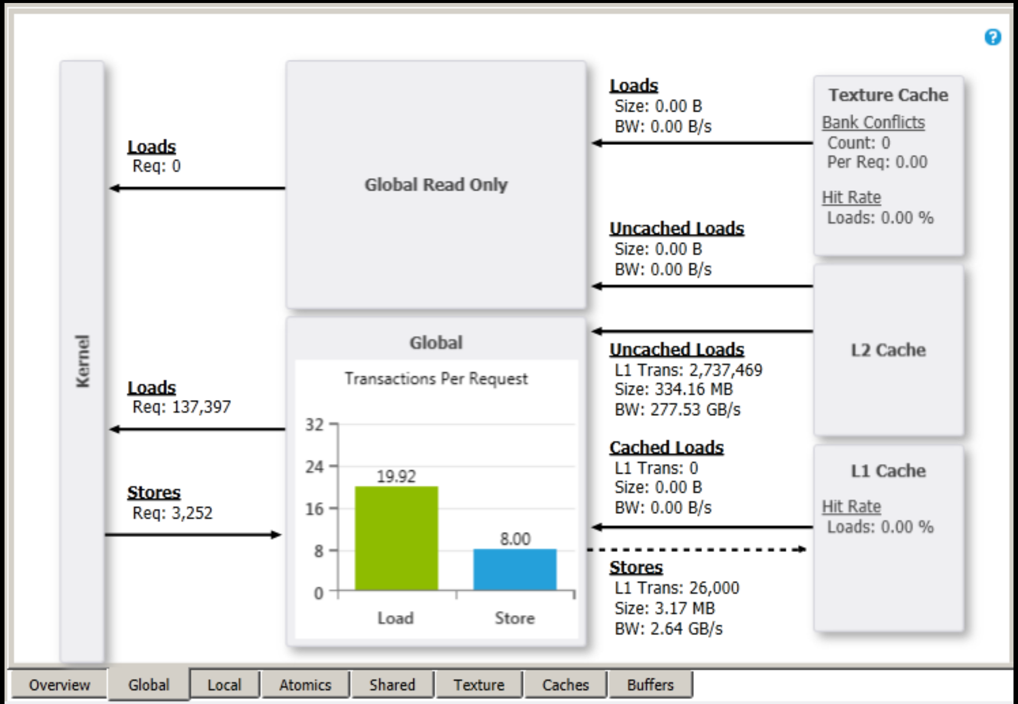
- 1 L1 transaction: 8B needed / 128B transferred
- 1 L2 transaction: 8B needed / 32B transferred

Replays

- A Memory Request: LD/ST instruction
- The 1st transaction is *issued*
- Other transactions induce *replays*
- Note: For each fp64 request, we have at least 1 replay

Latency (Memory Accesses)

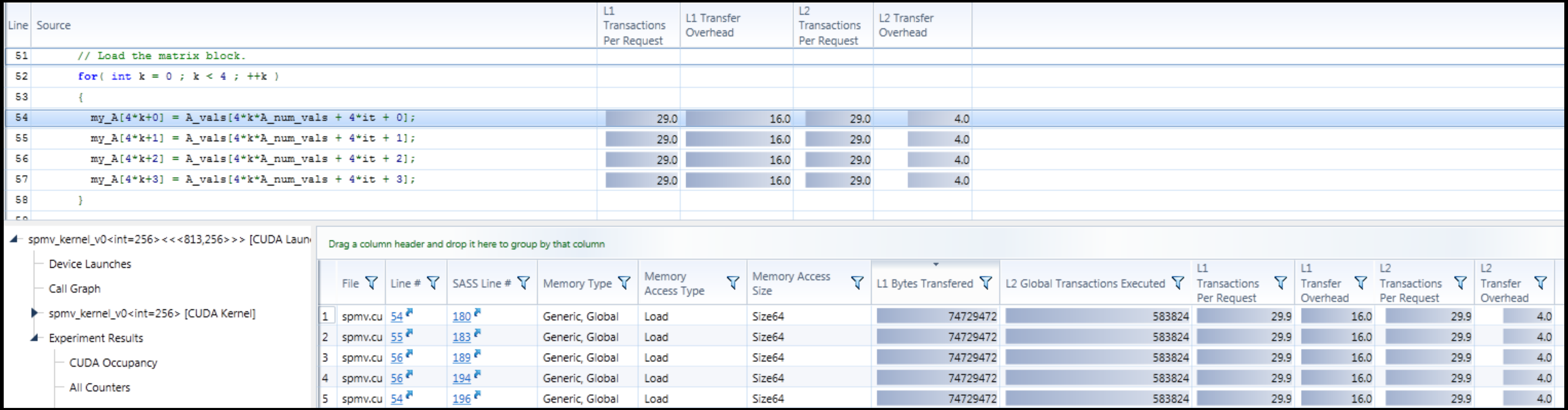
- Transactions per Request: 19.92 loads / 8 stores



- We have too many uncoalesced accesses!!!

Where Do Those Accesses Happen? (Nsight VSE)

- CUDA Source Profiler (Nsight VSE):
  - Where are uncoalesced requests (need to compile with –lineinfo)

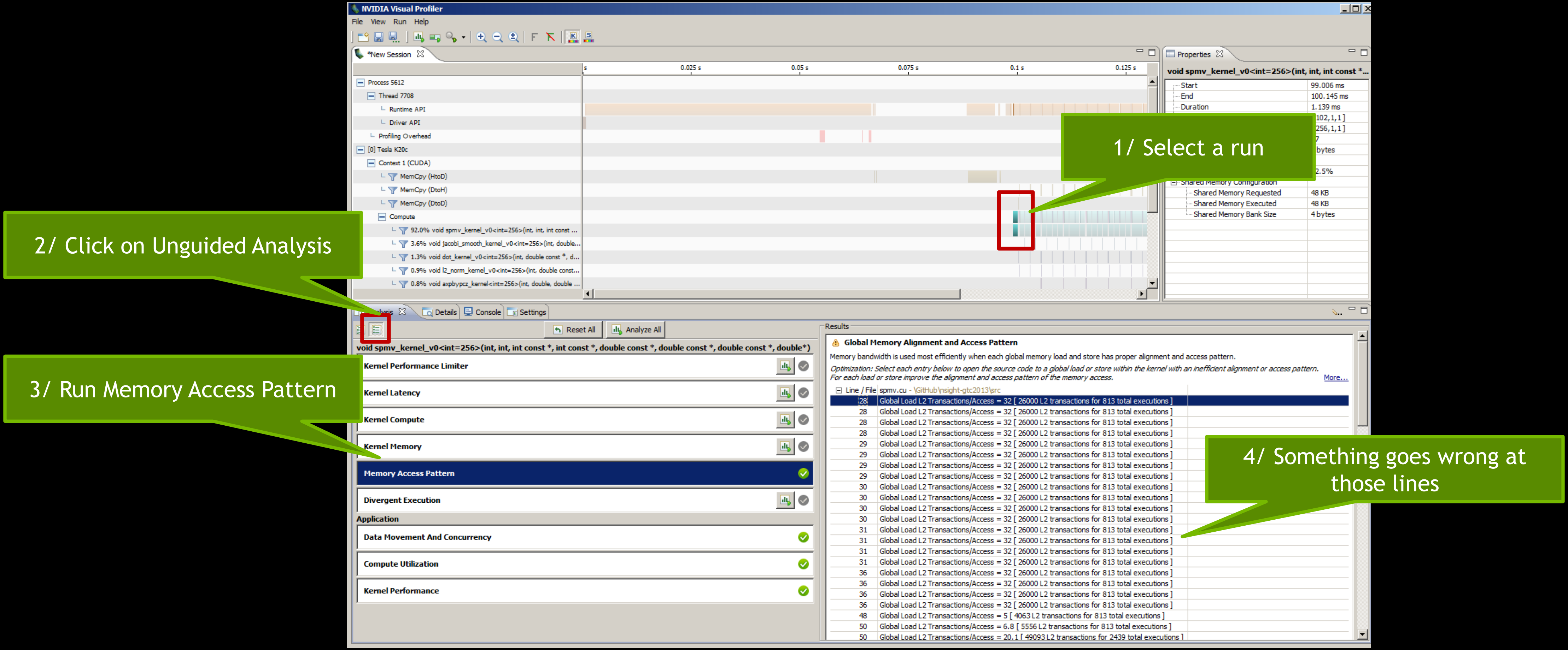


Line	Source	L1 Transactions Per Request	L1 Transfer Overhead	L2 Transactions Per Request	L2 Transfer Overhead
51	`// Load the matrix block.`				
52	`for( int k = 0 ; k < 4 ; ++k )`				
53	`{`				
54	`my_A[4*k+0] = A_vals[4*k*A_num_vals + 4*it + 0];`	29.0	16.0	29.0	4.0
55	`my_A[4*k+1] = A_vals[4*k*A_num_vals + 4*it + 1];`	29.0	16.0	29.0	4.0
56	`my_A[4*k+2] = A_vals[4*k*A_num_vals + 4*it + 2];`	29.0	16.0	29.0	4.0
57	`my_A[4*k+3] = A_vals[4*k*A_num_vals + 4*it + 3];`	29.0	16.0	29.0	4.0
58	`}`				

	File	Line #	SASS Line #	Memory Type	Memory Access Type	Memory Access Size	L1 Bytes Transfered	L2 Global Transactions Executed	L1 Transactions Per Request	L1 Transfer Overhead	L2 Transactions Per Request	L2 Transfer Overhead
1	spmv.cu	54	180	Generic, Global	Load	Size64	74729472	583824	29.9	16.0	29.9	4.0
2	spmv.cu	55	183	Generic, Global	Load	Size64	74729472	583824	29.9	16.0	29.9	4.0
3	spmv.cu	56	189	Generic, Global	Load	Size64	74729472	583824	29.9	16.0	29.9	4.0
4	spmv.cu	56	194	Generic, Global	Load	Size64	74729472	583824	29.9	16.0	29.9	4.0
5	spmv.cu	54	196	Generic, Global	Load	Size64	74729472	583824	29.9	16.0	29.9	4.0

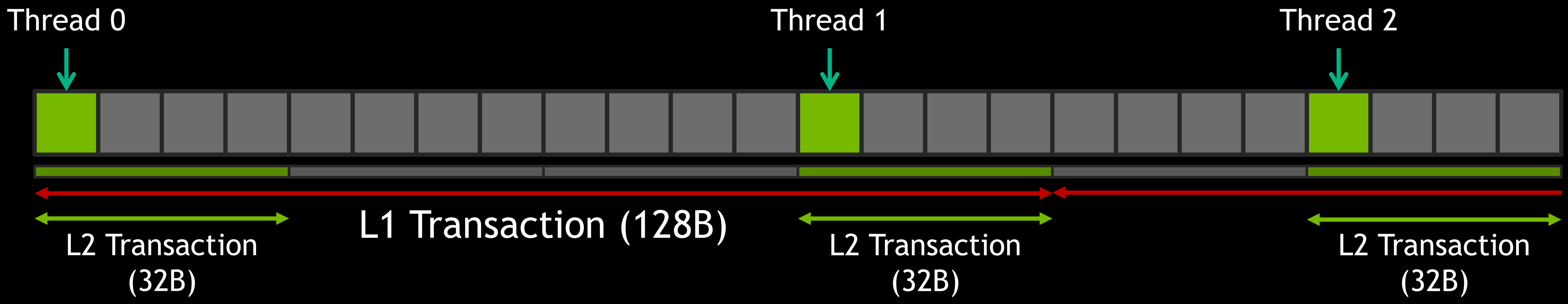
- Tip: Sort “L2 Global Transactions Executed”

Where Do Those Accesses Happen? (Nvvp)



Access Pattern

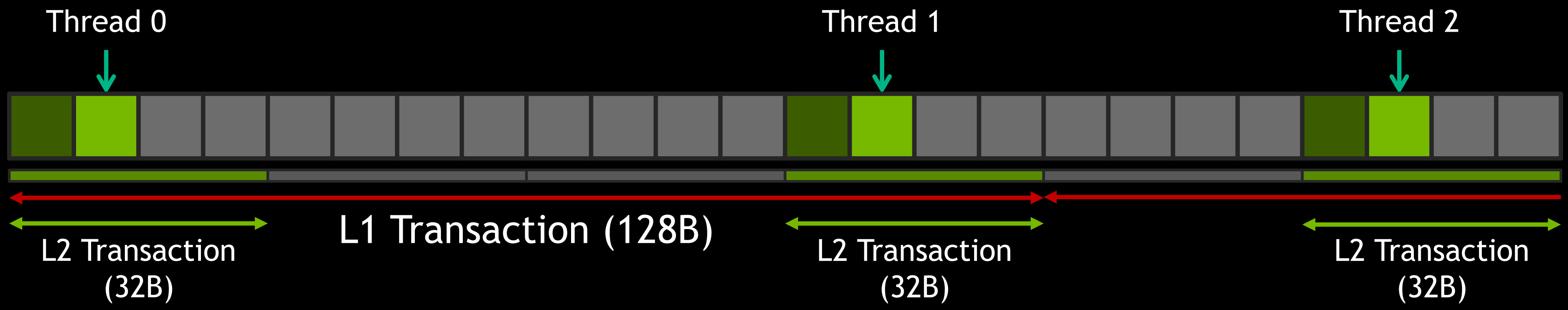
- Double precision numbers: 64-bit



- Per Warp:
  - Up to 32 L1 Transactions / Ideal case: 2 Transactions
  - Up to 32 L2 Transactions / Ideal case: 8 Transactions

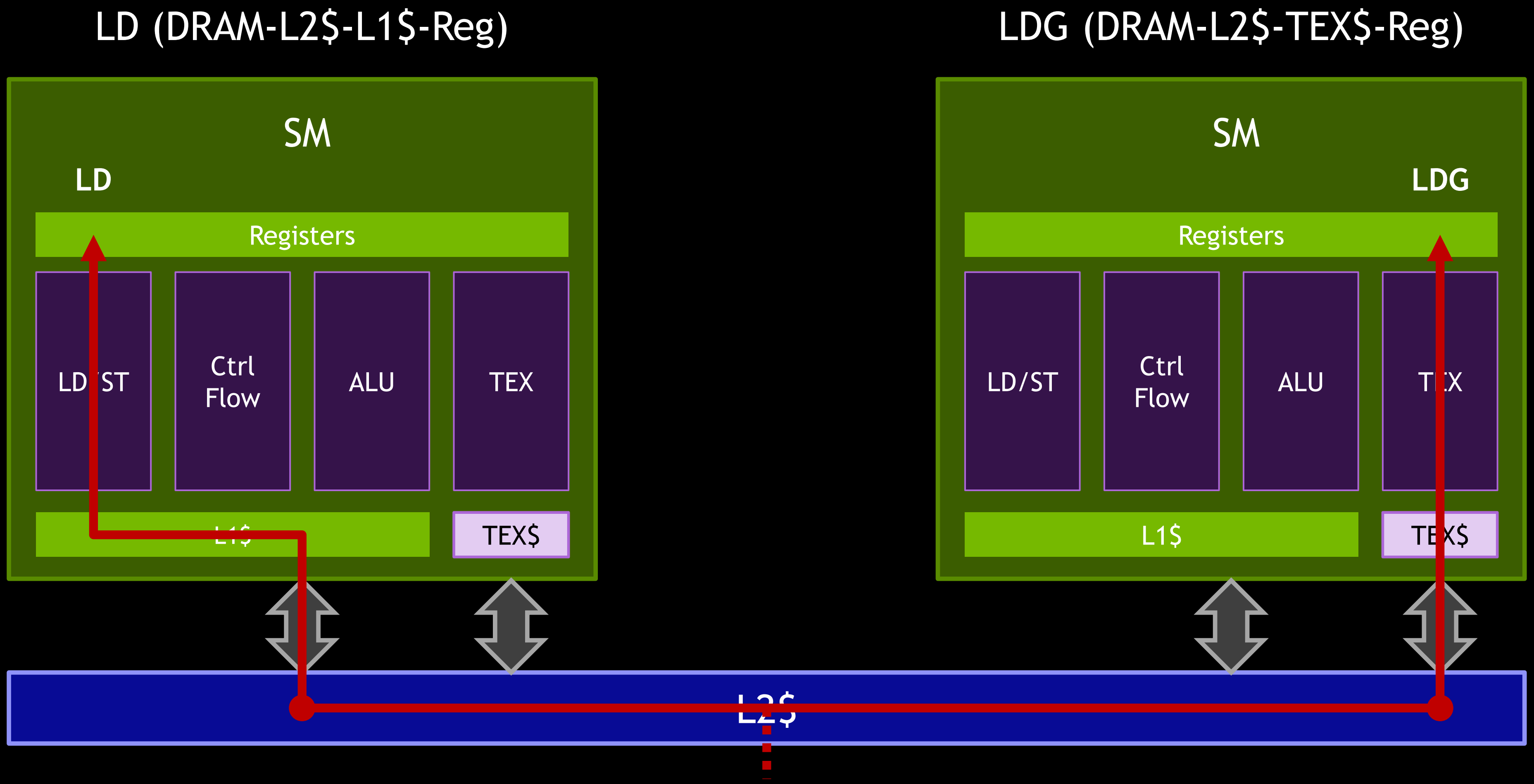
Access Pattern

- Next iteration:



- Idea: Use the Read-only cache (LDG load)
  – On Fermi: Use a texture or Use 48KB for L1

First Modification: Use __ldg



First Modification: Use __ldg

- We change the source code:

```
// Load the matrix block.
for( int k = 0 ; k < 4 ; ++k )
{
  my_A[4*k+0] = A_vals[4*k*A_num_vals + 4*it + 0];
  my_A[4*k+1] = A_vals[4*k*A_num_vals + 4*it + 1];
  my_A[4*k+2] = A_vals[4*k*A_num_vals + 4*it + 2];
  my_A[4*k+3] = A_vals[4*k*A_num_vals + 4*it + 3];
}
```

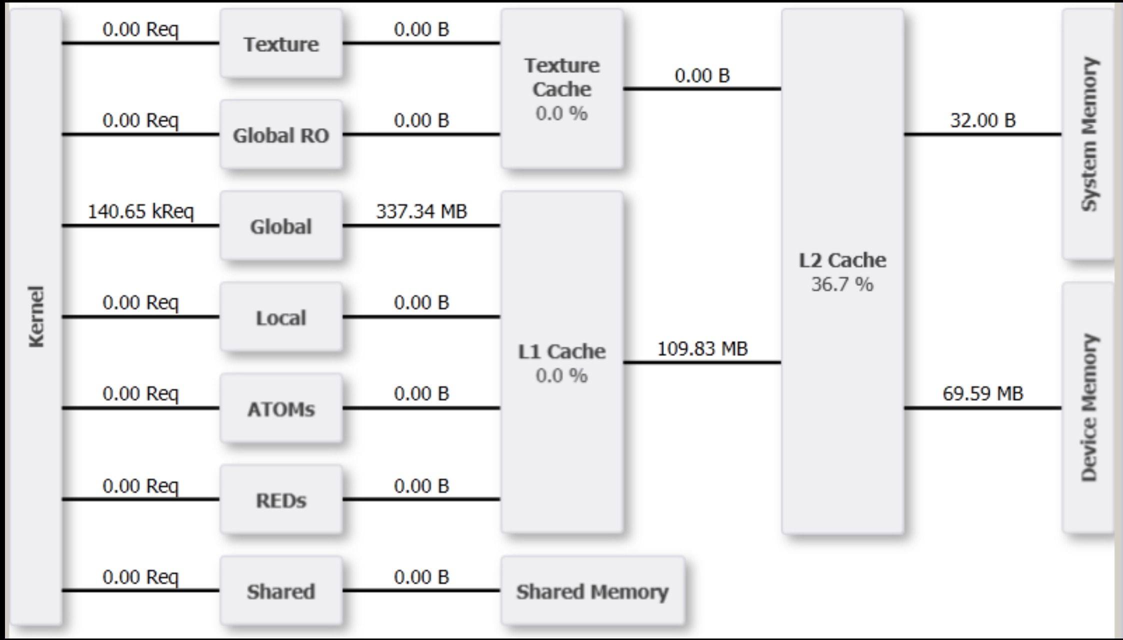
```
// Load the matrix block.
for( int k = 0 ; k < 4 ; ++k )
{
  my_A[4*k+0] = __ldg(&A_vals[4*k*A_num_vals + 4*it + 0]);
  my_A[4*k+1] = __ldg(&A_vals[4*k*A_num_vals + 4*it + 1]);
  my_A[4*k+2] = __ldg(&A_vals[4*k*A_num_vals + 4*it + 2]);
  my_A[4*k+3] = __ldg(&A_vals[4*k*A_num_vals + 4*it + 3]);
}
```

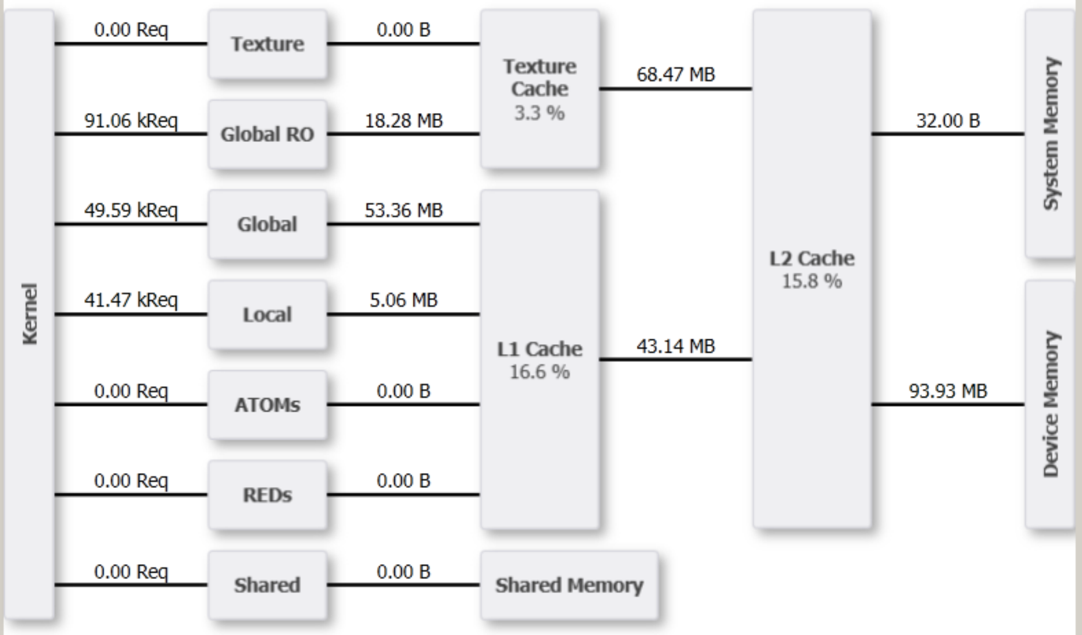
- It is slower: 625.8ms

Kernel	Time	Speedup
Original version	104.72ms	
LDG to load A	125.67ms	0.83x

First Modification: Use __ldg

- No benefit from the read-only cache: Hit rate at 3.3%

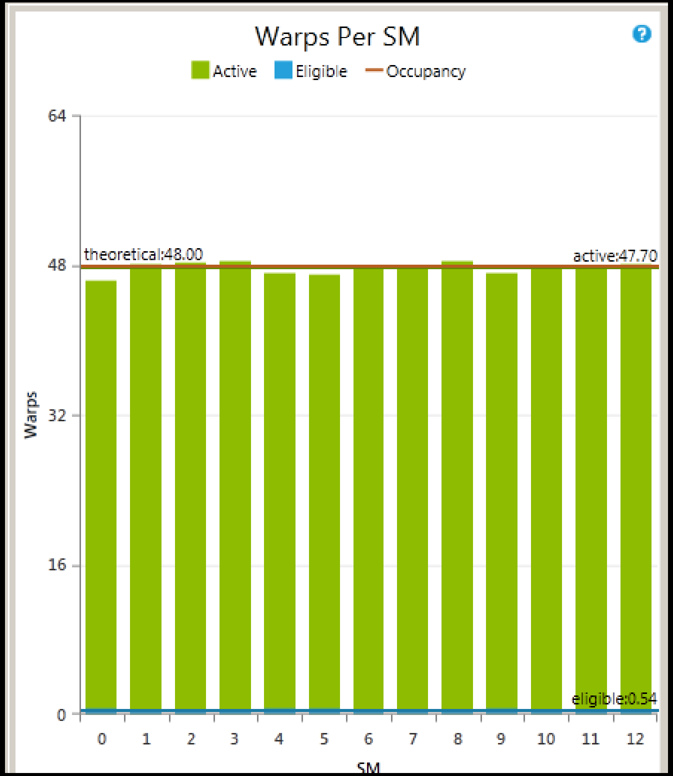




- Worse hit rate in L2$: 15.8% compared to 36.7%

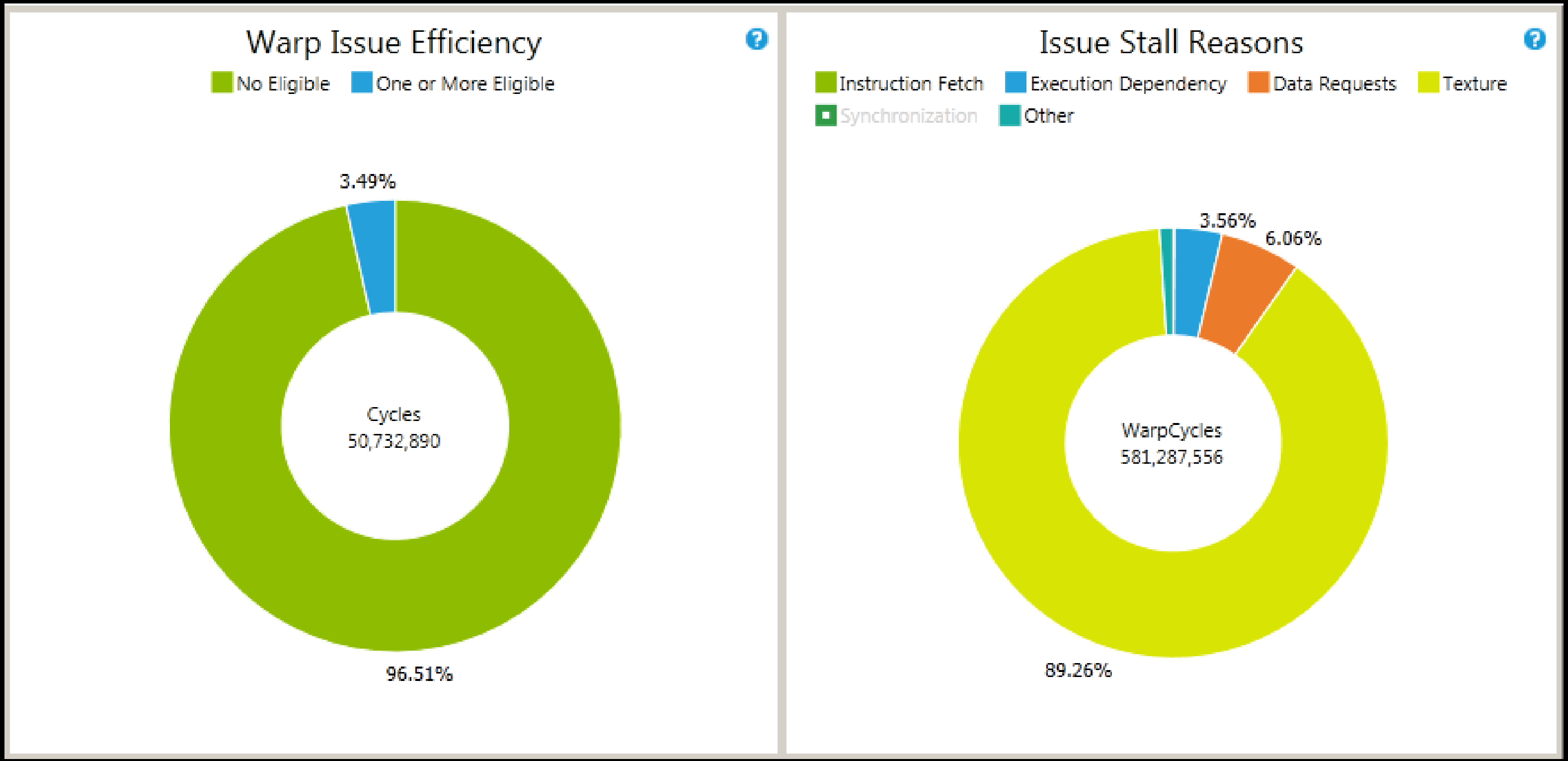
First Modification: Use __ldg

- Eligible Warps per Active Cycle has dropped to 0.54



First Modification: Use __ldg

- Warps cannot issue because they have to wait



First Modification: Use __ldg

- The loads compete for the cache too much
  - Low hit rate: 3.3%
- Texture requests introduce too much latency (in that case)

- Things to check in those cases:
  - Texture Hit Rate: Low means no reuse
  - Issue Efficiency and Stall Reasons

- It was actually expected: GPU caches are not CPU caches!!!

First Modification: Use __ldg

- Other accesses may benefit from LDGs
- Memory blocks accessed several times by several threads
- How can we detect it?
  - Source code analysis
  - There is no way to detect it from Nsight

First Modification: Use __ldg

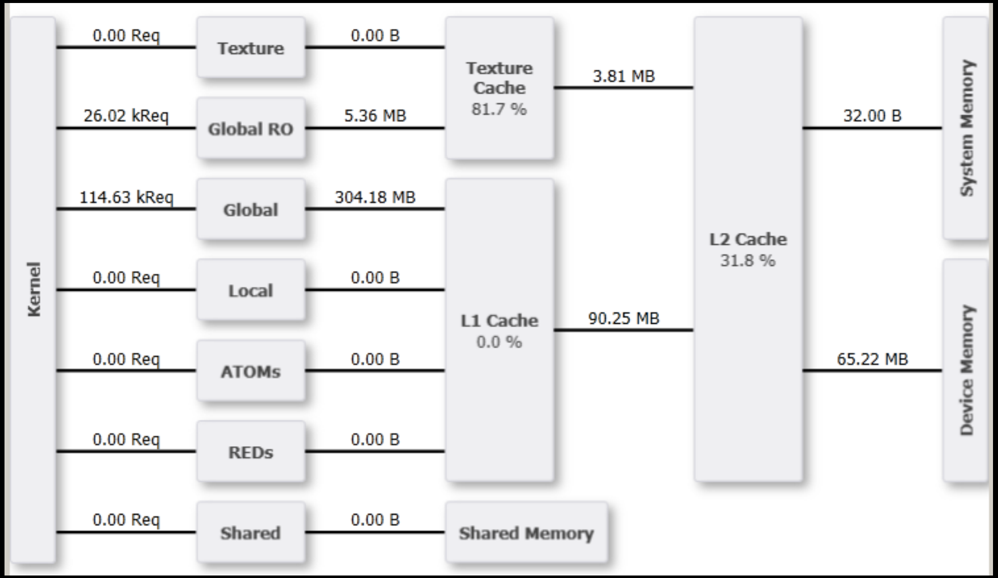
- We change the source code
  - In y = Ax, we use __ldg when loading x

- It’s faster: 98.30ms

Kernel	Time	Speedup
Original version	104.72ms	
LDG to load A	125.67ms	0.83x
LDG to load X	98.30ms	1.07x

First Modification: Use __ldg

- Good hit rate in Texture Cache: 82%



- Slightly less data transferred from L2 (94MB vs 110MB)

ITERATION 2

CUDA Launch Summary

	Function Name	Module ID	Function ID	Count	Device %	Device Time (μs)	Min (μs)	Avg (μs)	Max (μs)
1	spmv_kernel_v2<int=256>	46	2	71	28.62	77,976.969	1,036.168	1,098.267	1,168.022
2	jacobi_smooth_kernel_v0<int=256>	43	4	35	1.14	3,119.735	86.792	89.135	103.723
3	dot_kernel_v0<int=256>	44	1	70	0.41	1,104.842	11.041	15.783	18.434
4	l2_norm_kernel_v0<int=256>	45	1	36	0.30	819.765	21.667	22.771	24.227
5	axpbypcz_kernel<int=256>	44	5	34	0.26	721.191	20.674	21.212	21.858
6	axpby_kernel<int=256>	44	4	37	0.23	619.803	16.065	16.751	17.473
7	reduce_kernel<int=256>	44	3	70	0.10	261.758	3.136	3.739	4.289
8	reduce_l2_norm_kernel<int=256>	45	2	36	0.07	180.501	4.737	5.014	6.337
9	jacobi_invert_diag_kernel_v0<int=256>	43	1	1	0.04	109.932	109.932	109.932	109.932

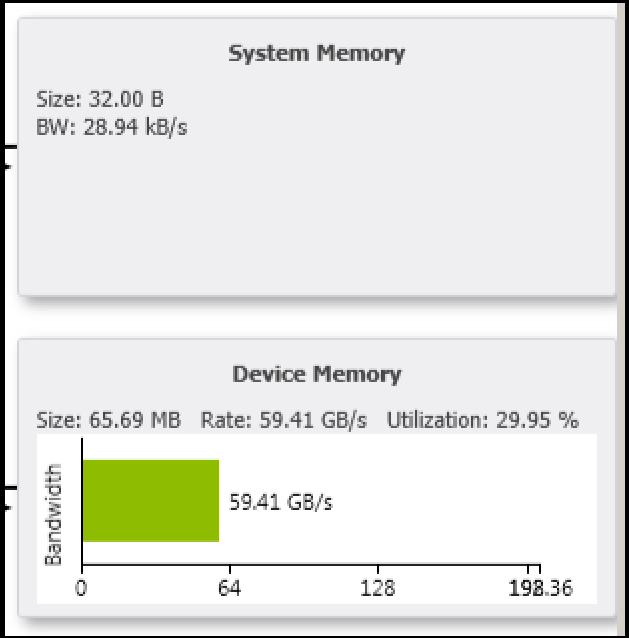
- spmv_kernel_v2 is still a hot spot, so we profile it

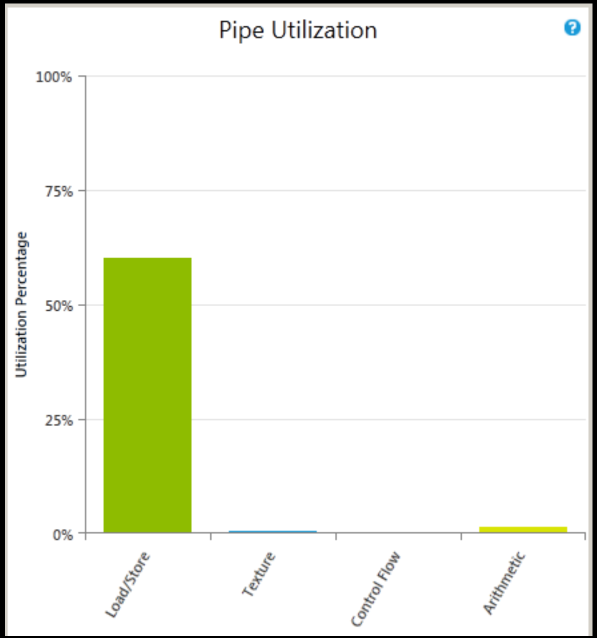
Identify the Main Limiter

- Is it limited by the memory bandwidth ?
- Is it limited by the instruction throughput ?
- Is it limited by latency ?

Identify the Main Limiter

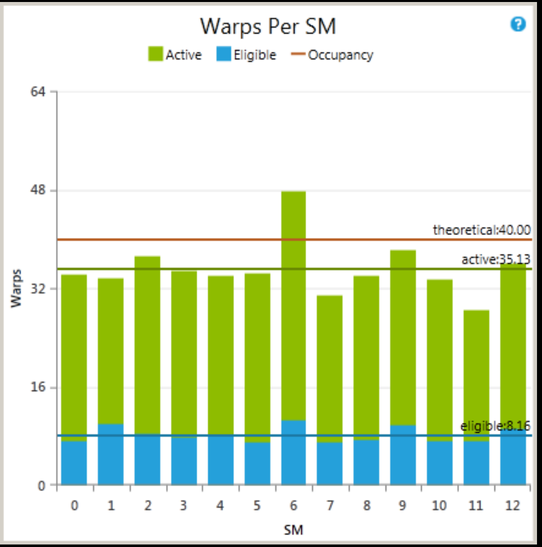
- We are still limited by latency
  - Low DRAM utilization: 29.95%
  - Pipe utilization is Low/Mid: <70-75%





Identify the Main Limiter

- We are not limited by the Occupancy
  - We have > 4 Eligible Warps per Active Cycle (8.16)

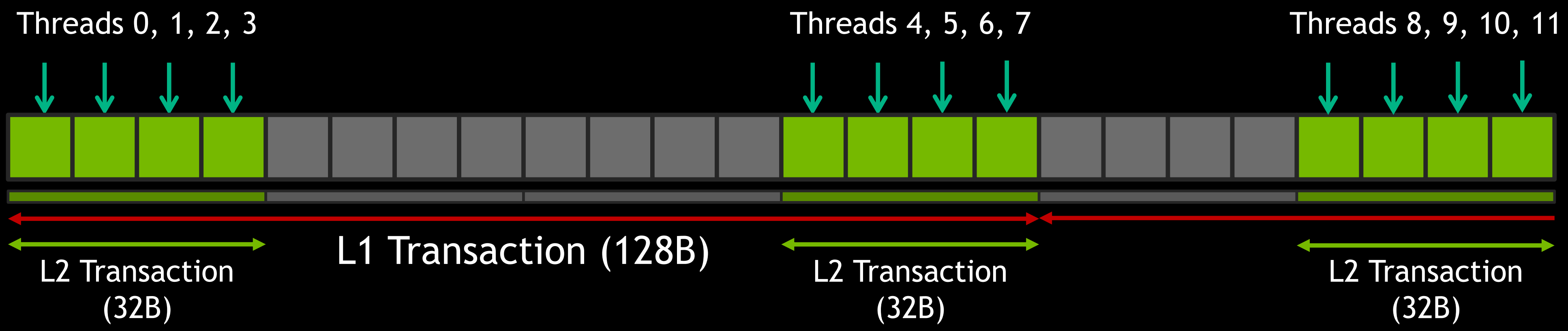


Name	Total	Per Warp	Per Second
Total - SM to L1/Tex/L2			
Requests	140,649.00	172.36	130,273,000.00
Transactions	2,491,833.00	3,053.72	2,308,005,000.00
Size	309.54 MB	388.45 kB	279.99 GB/s
Replay Overhead	40.79 %		

- Too many uncoalesced accesses: 40.79% of Replay Overhead

Second Strategy: Change Memory Accesses

- 4 consecutive threads load 4 consecutive elements



- Per Warp:
  - Up to 8 L1 Transactions / Ideal case: 2 Transactions
  - Up to 8 L2 Transactions / Ideal case: 8 Transactions

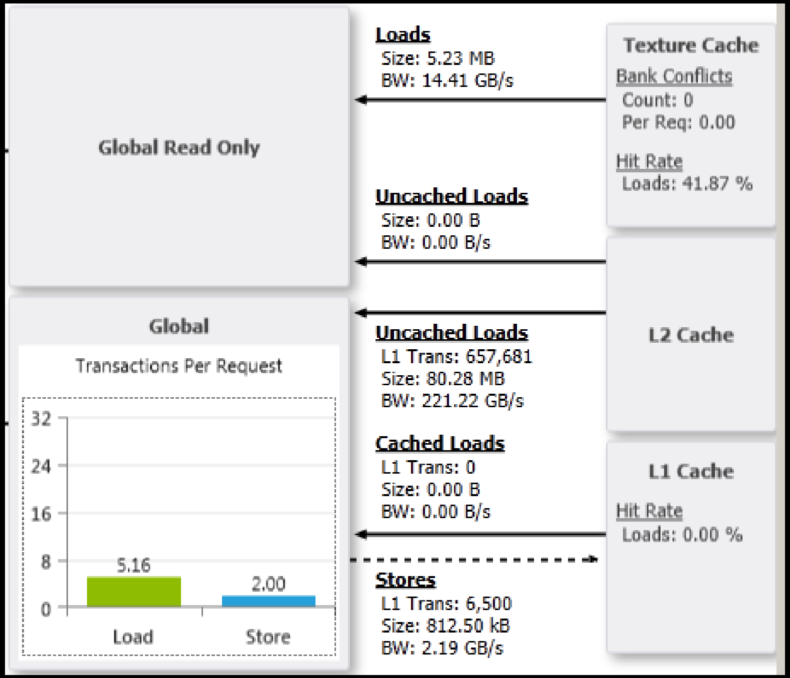
Second Strategy: Change Memory Accesses

- It's much faster: 45.61ms

Kernel	Time	Speedup
Original version	104.72ms	
LDG to load A	125.67ms	0.83x
LDG to load X	98.30ms	1.07x
Coalescing with 4 Threads	45.61ms	2.30x

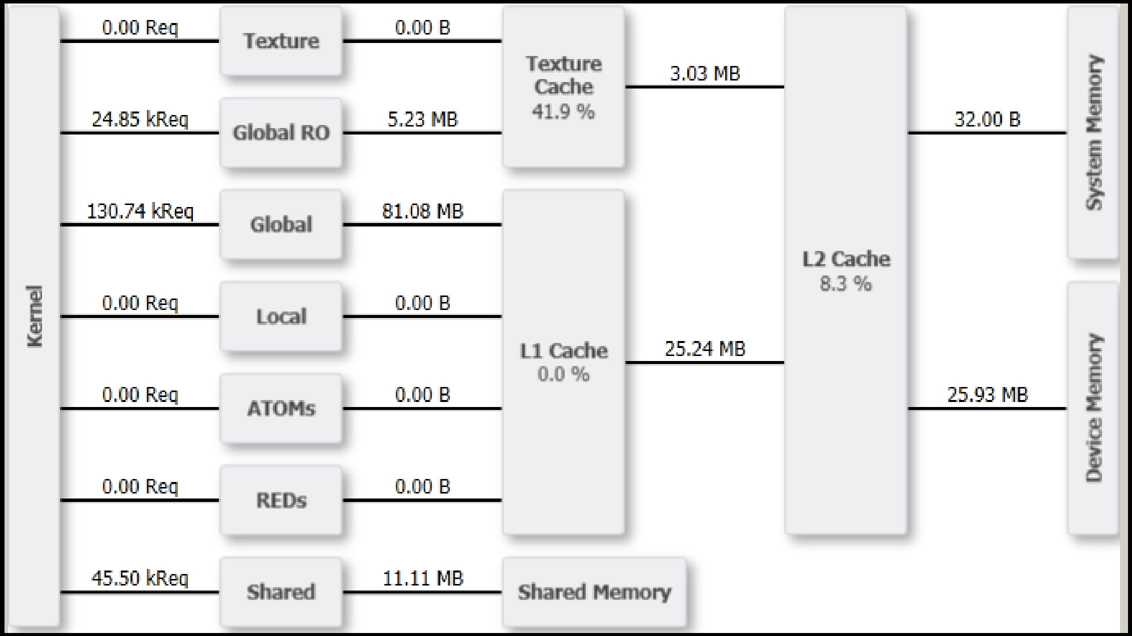
Second Strategy: Change Memory Accesses

- We have much fewer Transactions per Request: 5.16 (LD)



Second Strategy: Change Memory Accesses

- Much less traffic from L2: 28.27MB (it was 109.83MB)



- Much less DRAM traffic: 25.93MB (it was 69.59MB)

ITERATION 3

CUDA Launch Summary

	Function Name	Module ID	Function ID	Count	Device %	Device Time (µs)	Min (µs)	Avg (µs)	Max (µs)
1	spmv_kernel_v3<int=256>	46	1	71	11.12	24,940.624	348.100	351.276	354.884
2	jacobi_smooth_kernel_v0<int=256>	44	4	35	1.42	3,178.500	86.729	90.814	102.666
3	dot_kernel_v0<int=256>	45	1	70	0.49	1,090.577	10.977	15.580	18.434
4	l2_norm_kernel_v0<int=256>	43	1	36	0.36	806.421	21.378	22.401	23.171
5	axpbypcz_kernel<int=256>	45	5	34	0.32	718.986	20.642	21.147	21.795
6	axpby_kernel<int=256>	45	4	37	0.27	610.141	15.746	16.490	17.122
7	reduce_kernel<int=256>	45	3	70	0.11	255.583	3.104	3.651	4.033
8	reduce_l2_norm_kernel<int=256>	43	2	36	0.08	190.223	5.057	5.284	6.241
9	jacobi_invert_diag_kernel_v0<int=256>	44	1	1	0.05	109.931	109.931	109.931	109.931

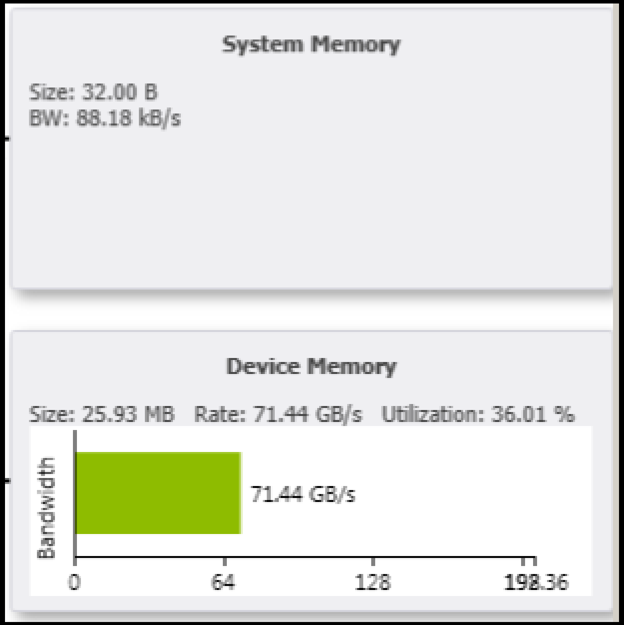
- spmv_kernel_v3 is still a hot spot, so we profile it

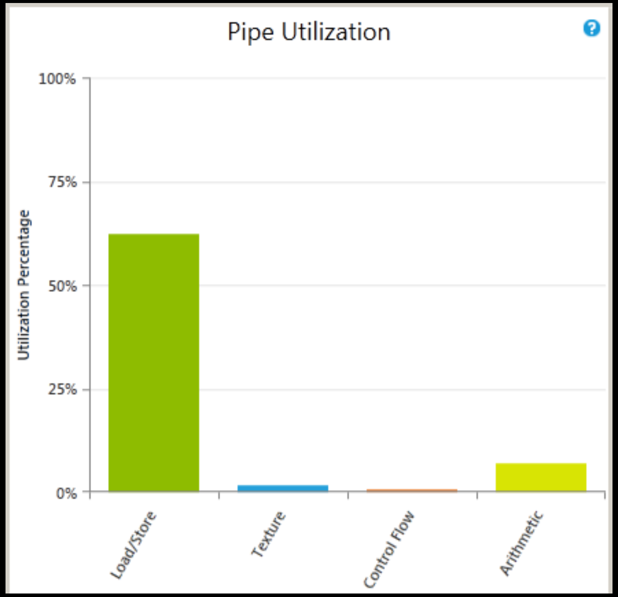
Identify the Main Limiter

- Is it limited by the memory bandwidth ?
- Is it limited by the instruction throughput ?
- Is it limited by latency ?

Identify the Main Limiter

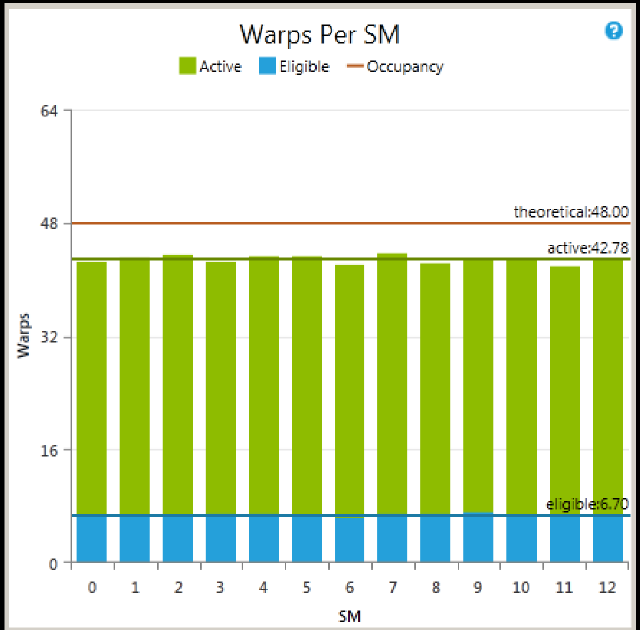
- We are still limited by latency
  - Low DRAM utilization: 36.01%
  - Pipe Utilization is still Low/Mid





Latency

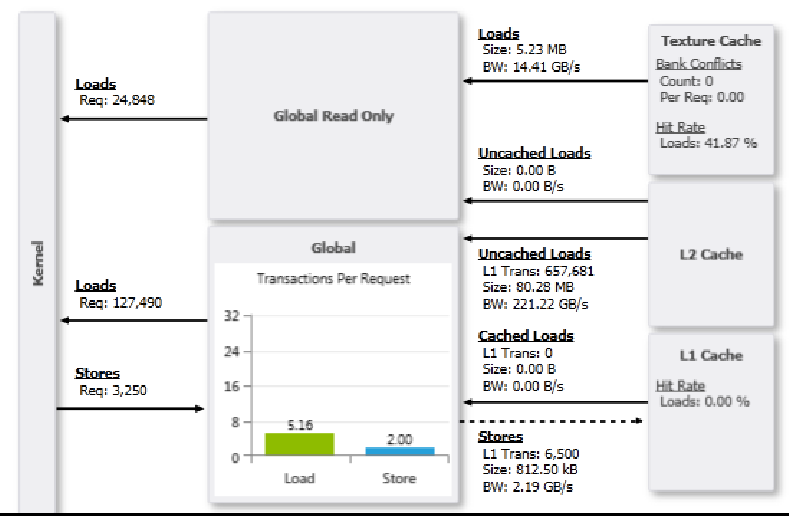
- Eligible Warps per Active Cycle: 6.70 on average



- We are not limited by occupancy

Latency

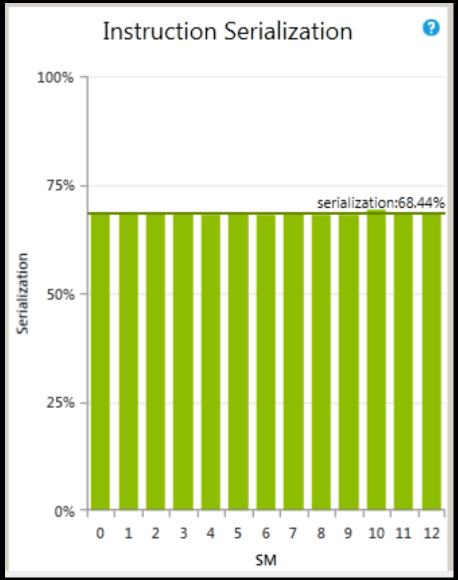
- Memory Accesses:
  - Load: 5.16 Transactions per Request
  - Store: 2 Transactions per Request

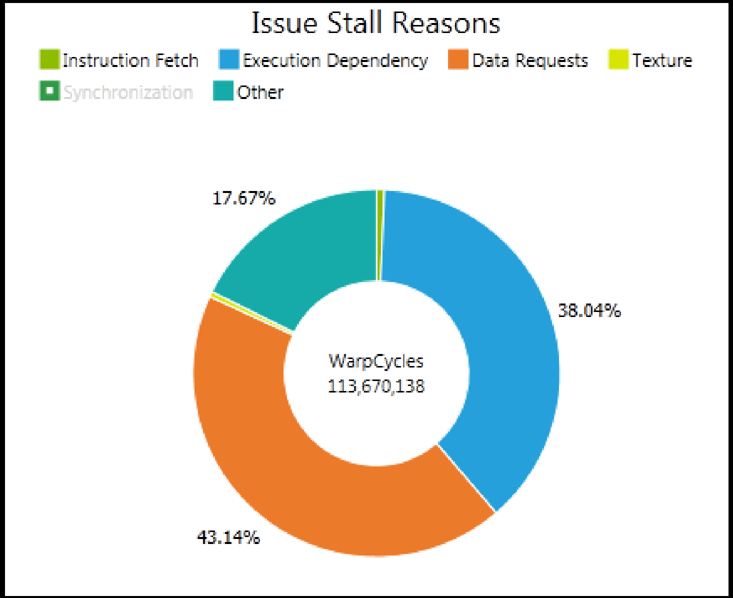


- We still have too many uncoalesced accesses

Latency

- We still have too many uncoalesced accesses
  - Nearly 68.44% of Instruction Serialization (Replays)
  - Stall Reasons: 43.14% due to Data Requests





Latency

- Serialization: (Inst. Issued – Inst. Executed) / Inst. Issued

```
> nvprof --kernels "::spmv_kernel_v3:" --metrics "inst_replay_overhead" .\x64\Release\BiCGStab.exe
```

- Inst. Replay Overhead: Avg. Number of replays per Inst.

- Inst. Issued = 1 + Avg. Number of Replays

Inst. Replay Overhead	Inst. Replay Overhead / (1 + Inst. Replay Overhead)
2.17	68.45%

Latency

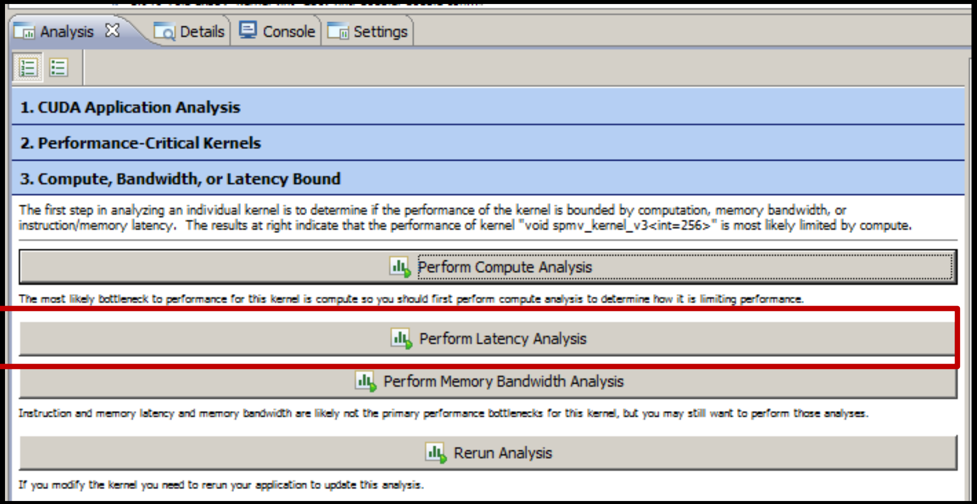
- Issue Stall Reasons

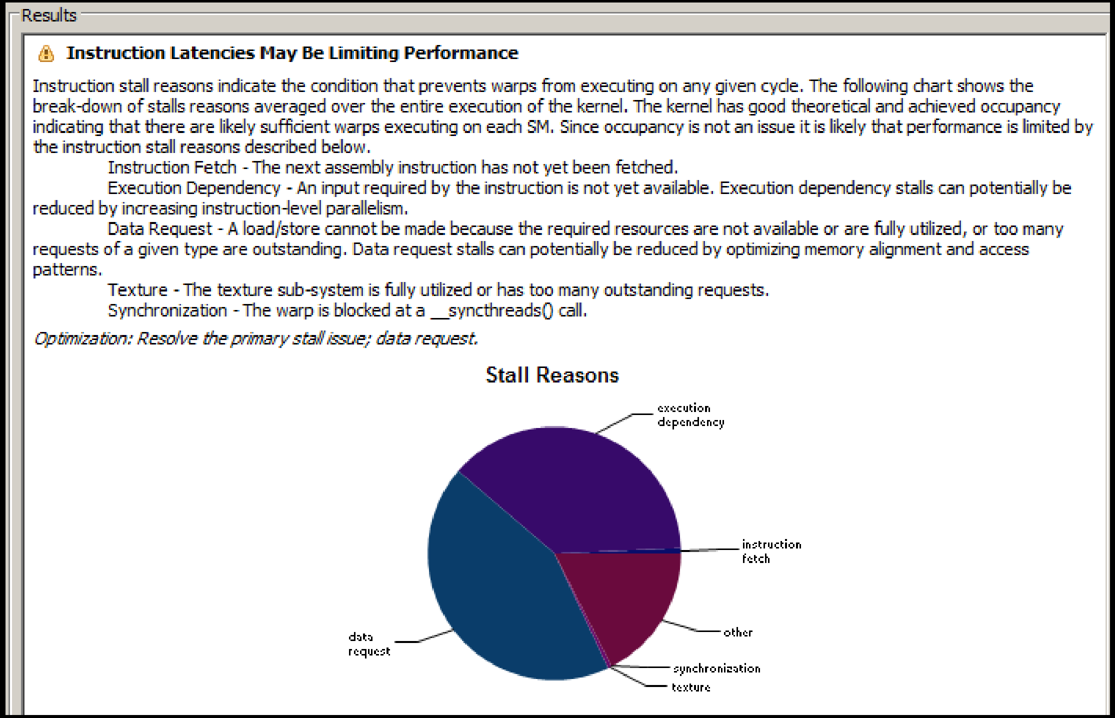
```
> nvprof --kernels "::spmv_kernel_v3:" --metrics "stall_inst_fetch,stall_exec_dependency,stall_data_request" .\x64\Release\BiCGStab.exe
```

```
> nvprof --kernels "::spmv_kernel_v3:" --metrics "stall_texture,stall_sync,stall_other" .\x64\Release\BiCGStab.exe
```

Stall Reasons	
Instruction Fetch	0.58%
Execution Dependency	32.51%
Data Request	37.85%
Texture	0.41%
Sync	0.00%
Other	15.13%

Latency


Analysis
Details
Console
Settings
1. CUDA Application Analysis
2. Performance-Critical Kernels
3. Compute, Bandwidth, or Latency Bound
The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "void spmv_kernel_v3<int=256>" is most likely limited by compute.
Perform Compute Analysis
The most likely bottleneck to performance for this kernel is compute so you should first perform compute analysis to determine how it is limiting performance.
Perform Latency Analysis
Perform Memory Bandwidth Analysis
Instruction and memory latency and memory bandwidth are likely not the primary performance bottlenecks for this kernel, but you may still want to perform those analyses.
Rerun Analysis
If you modify the kernel you need to rerun your application to update this analysis.


Results
Instruction Latencies May Be Limiting Performance
Instruction stall reasons indicate the condition that prevents warps from executing on any given cycle. The following chart shows the break-down of stalls reasons averaged over the entire execution of the kernel. The kernel has good theoretical and achieved occupancy indicating that there are likely sufficient warps executing on each SM. Since occupancy is not an issue it is likely that performance is limited by the instruction stall reasons described below.
Instruction Fetch - The next assembly instruction has not yet been fetched.
Execution Dependency - An input required by the instruction is not yet available. Execution dependency stalls can potentially be reduced by increasing instruction-level parallelism.
Data Request - A load/store cannot be made because the required resources are not available or are fully utilized, or too many requests of a given type are outstanding. Data request stalls can potentially be reduced by optimizing memory alignment and access patterns.
Texture - The texture sub-system is fully utilized or has too many outstanding requests.
Synchronization - The warp is blocked at a __syncthreads() call.
Optimization: Resolve the primary stall issue; data request.
Stall Reasons
execution dependency
instruction fetch
other
synchronization
texture
data request

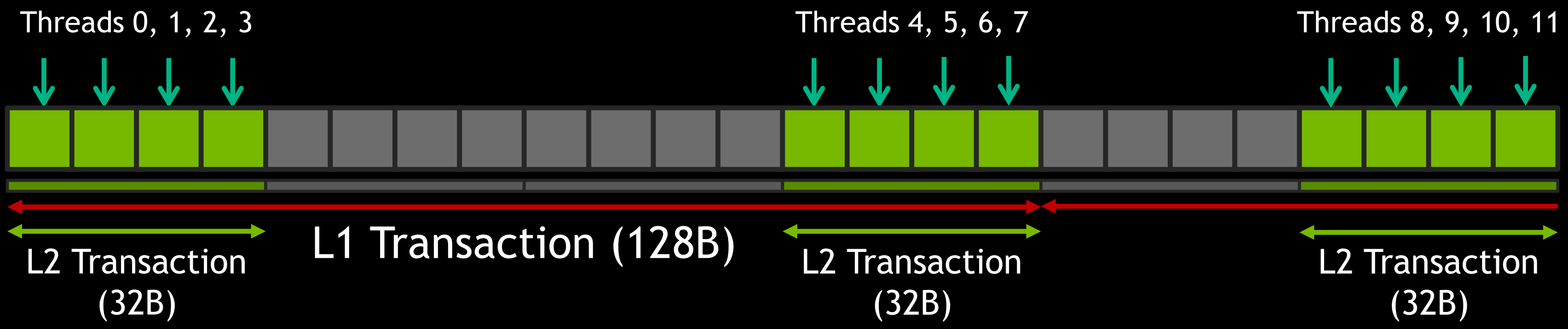
Where Do Those Accesses Happen?

- Same lines of code as before

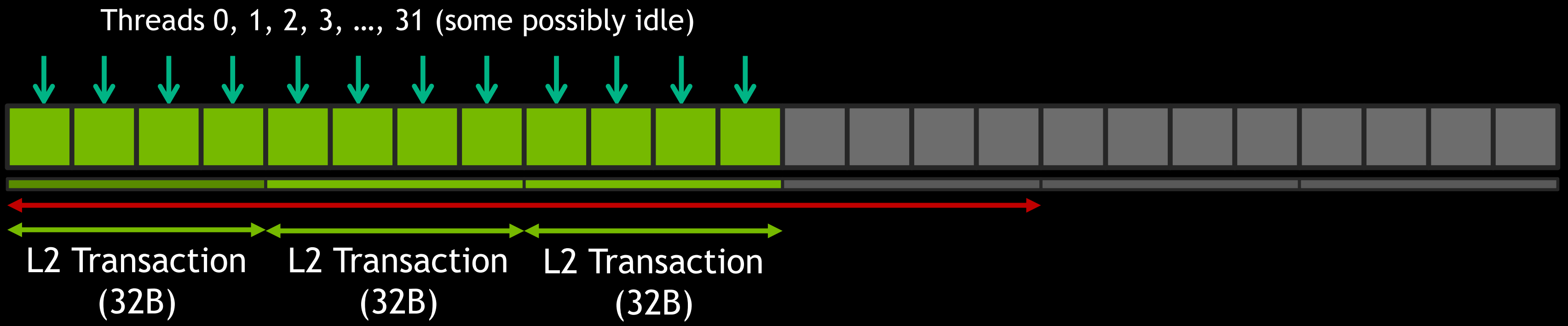
Line	Source						
276	`// Each thread iterates over its row.`						
277	`for( int it = A_rows[row], end = A_rows[row+1] ; it < end ; ++it )`	151606	4366176	85.0	6500	83.8	Gen
278	`{`						
279	`const int col = A_cols[it];`	140284	3764736	83.9			Gen
280							
281	`// Load the matrix block.`						
282	`for( int k = 0 ; k < 4 ; ++k )`						
283	`my_A[k] = A_vals[4*k*A_num_vals + 4*it + lane_id_mod_4];`	215980	5816000	84.2			Gen
284							
285	`// Load the x block.`						
286	`my_x = __ldg(&x[4*col + lane_id_mod_4]);`	64794	1744800	84.2			

What Can We Do?

- In our kernel: 4 threads per row of the matrix A



- New approach: 1 warp of threads per row of the matrix A



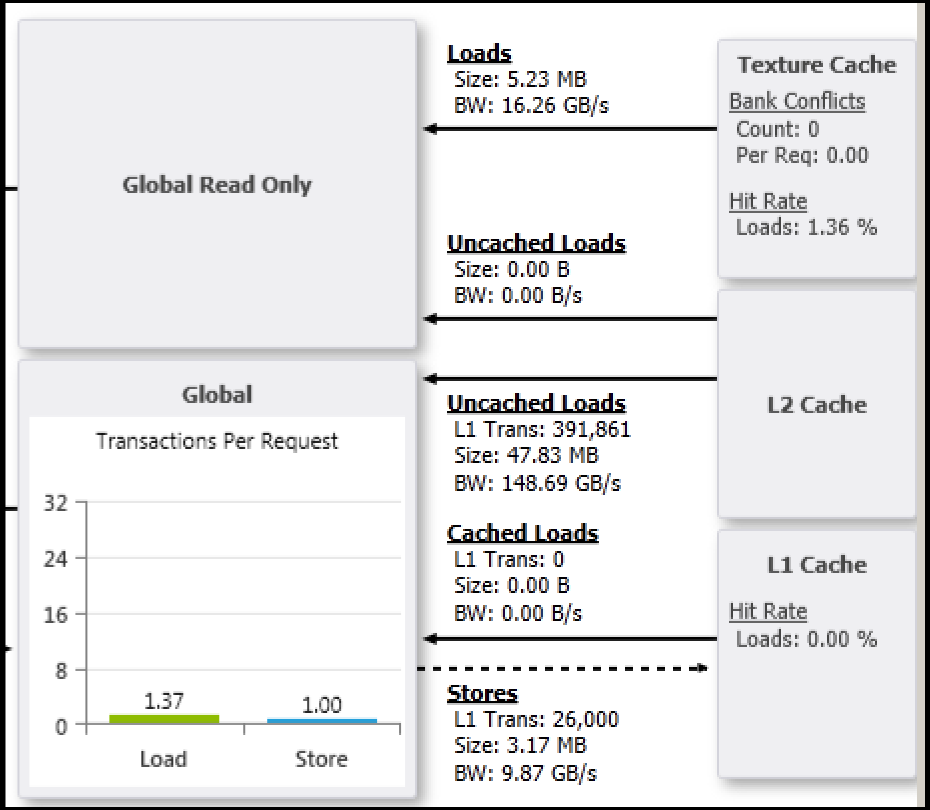
One Warp Per Row

- It's faster: 37.50ms

Kernel	Time	Speedup
Original version	104.72ms	
LDG to load A	125.67ms	0.83x
LDG to load X	98.30ms	1.07x
Coalescing with 4 Threads	45.61ms	2.30x
1 Warp per Row	37.50ms	2.79x

One Warp Per Row

- Much fewer Transactions Per Request: 1.37 (LD) / 1 (ST)



ITERATION 4

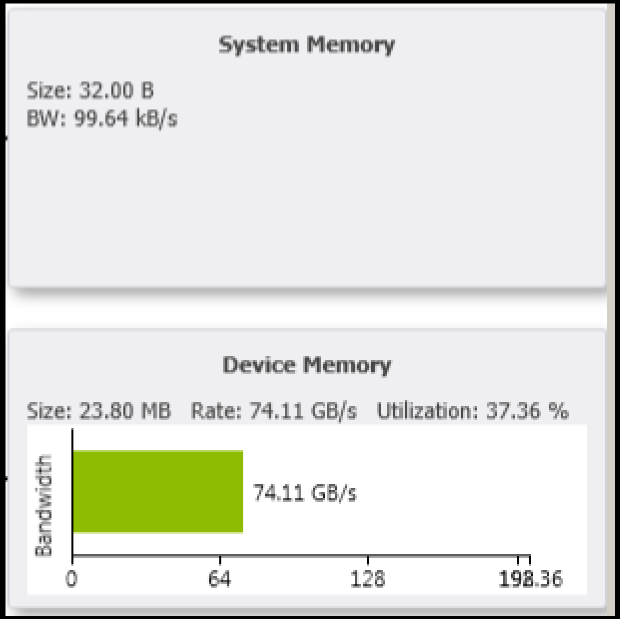
One Warp Per Row

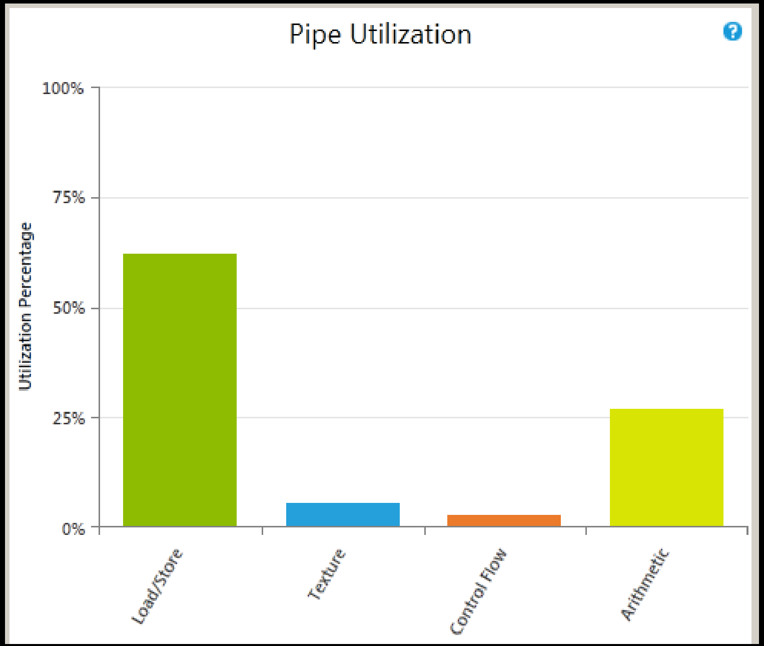
- spmv_kernel_v4 is the hot spot

	Function Name	Module ID	Function ID	Count	Device %	Device Time (µs)	Min (µs)	Avg (µs)	Max (µs)
1	spmv_kernel_v4<int=128>	45	6	71	9.91	22,168.706	310.528	312.235	314.496
2	jacobi_smooth_kernel_v0<int=256>	47	4	35	1.40	3,124.255	86.410	89.264	97.258
3	dot_kernel_v0<int=256>	46	1	70	0.49	1,092.756	11.009	15.611	17.986
4	l2_norm_kernel_v0<int=256>	44	1	36	0.37	822.549	21.826	22.849	24.643
5	axpbypcz_kernel<int=256>	46	5	34	0.32	719.882	20.578	21.173	21.666
6	axpby_kernel<int=256>	46	4	37	0.27	608.196	15.778	16.438	17.282
7	reduce_kernel<int=256>	46	3	70	0.11	256.478	3.072	3.664	4.033
8	reduce_l2_norm_kernel<int=256>	44	2	36	0.08	190.192	5.056	5.283	6.177
9	jacobi_invert_diag_kernel_v0<int=256>	47	1	1	0.05	109.612	109.612	109.612	109.612

One Warp Per Row

- DRAM utilization: 37.36%
- Pipe Utilization is Low/Mid

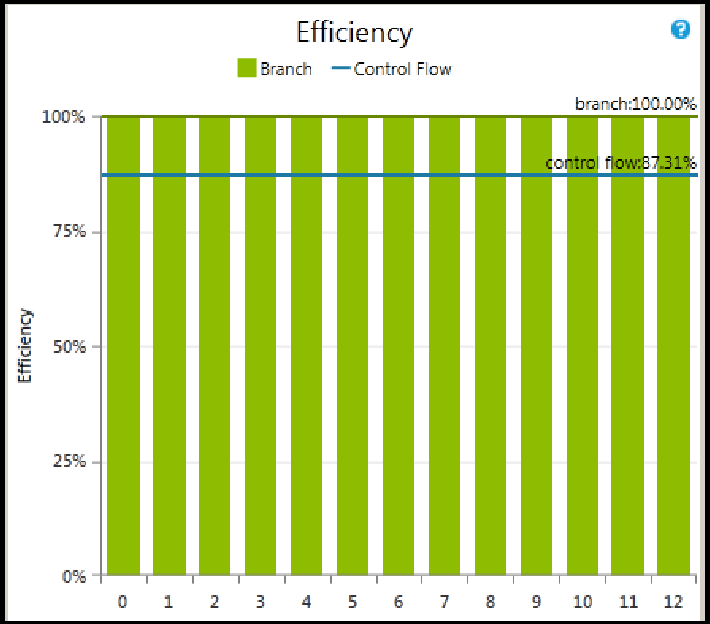




- We are still limited by latency

One Warp Per Row

- Occupancy and memory accesses are OK (not shown)
- Control Flow Efficiency: 87.31%




```
> nvprof --kernels "::spmv_kernel_v4:" --metrics "warp_execution_efficiency" .\x64\Release\BiCGStab.exe
```

Control Flow Efficiency

- All threads work: 100%

- Some threads do nothing: Less efficiency

```
if( threadIdx.x % 32 < 24 ) {
    … // do some long computation
}
```

Efficiency = 24/32 = 75%

Control Flow Efficiency

- Low efficiency in one the key loop: 69.9%

Line	Source	Instruct Execute	Thread Instructio Executed	Thread Execution Efficiency
363	for(int it = A_rows[row]+lane_id/4, end = A_rows[row+1] ; it < end ; it…	429000	12476000	90.9
364	{			
365	// Load the column.			
366	int col = A_cols[it];	208000	4652800	69.9
367				
368	// Load the matrix block and x.			
369	#pragma unroll			
370	for(int k = 0 ; k < 4 ; ++k)			
371	my_A[k] = A_vals[4*k*A_num_vals + 4*it + lane_id_mod_4];	260000	5816000	69.9
372	my_x = __ldg(&x[4*col+lane_id_mod_4]);	78000	1744800	69.9

One Half Warp Per Row

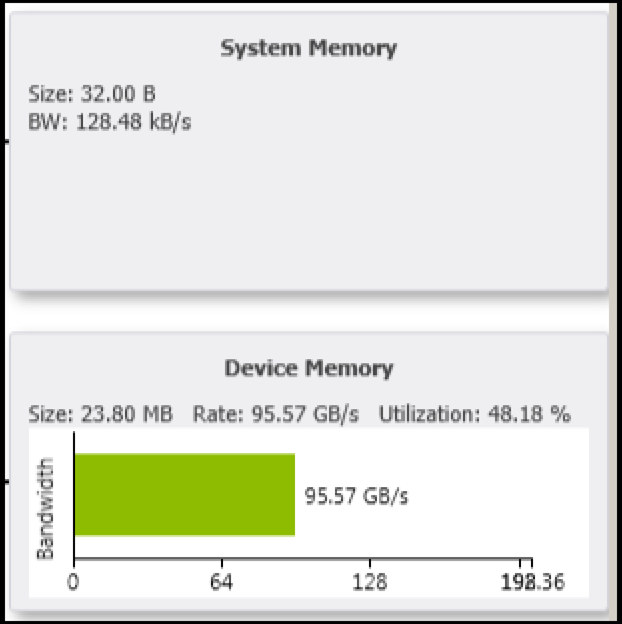
- It is faster: 35.81ms

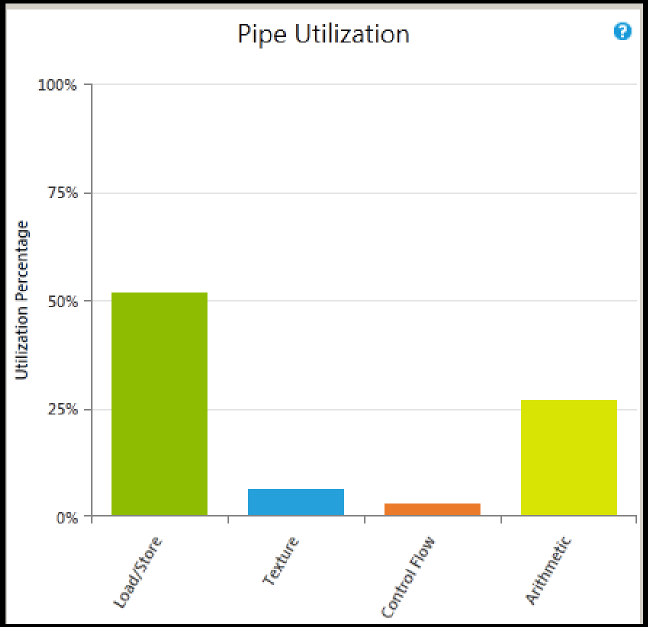
Kernel	Time	Speedup
Original version	104.72ms	
LDG to load A	125.67ms	0.73x
LDG to load X	98.30ms	1.07x
Coalescing with 4 Threads	45.61ms	2.30x
1 Warp per Row	37.50ms	2.79x
½ Warp per Row	35.81ms	2.93x

ITERATION 5

One Half Warp Per Row

- DRAM utilization: 48.18%
- Pipe Utilization is Low/Mid

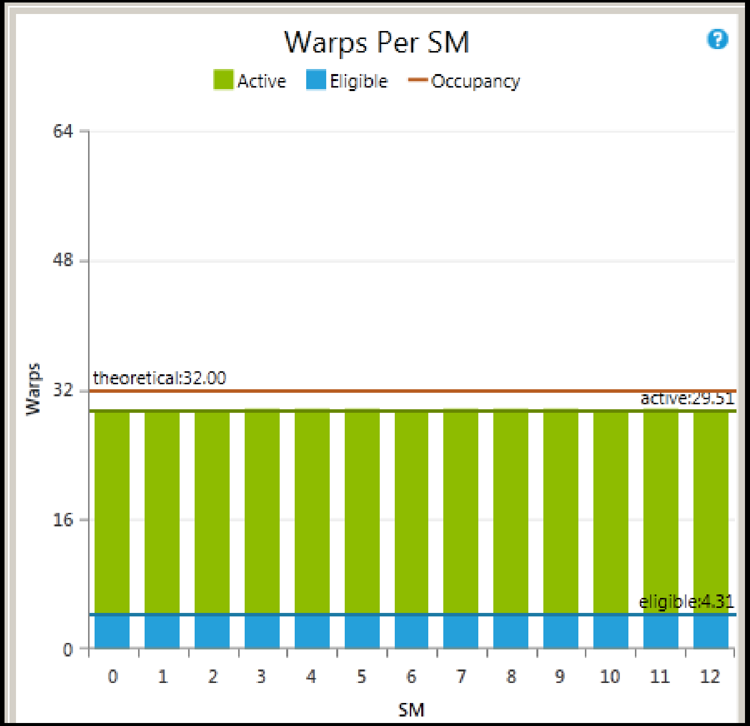




- We are still limited by latency

One Half Warp Per Row

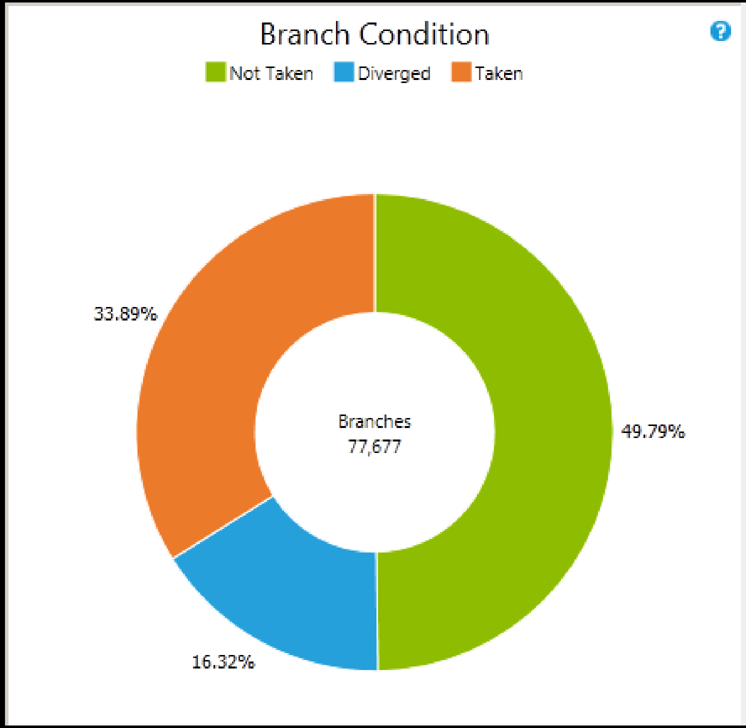
- Occupancy is not an issue



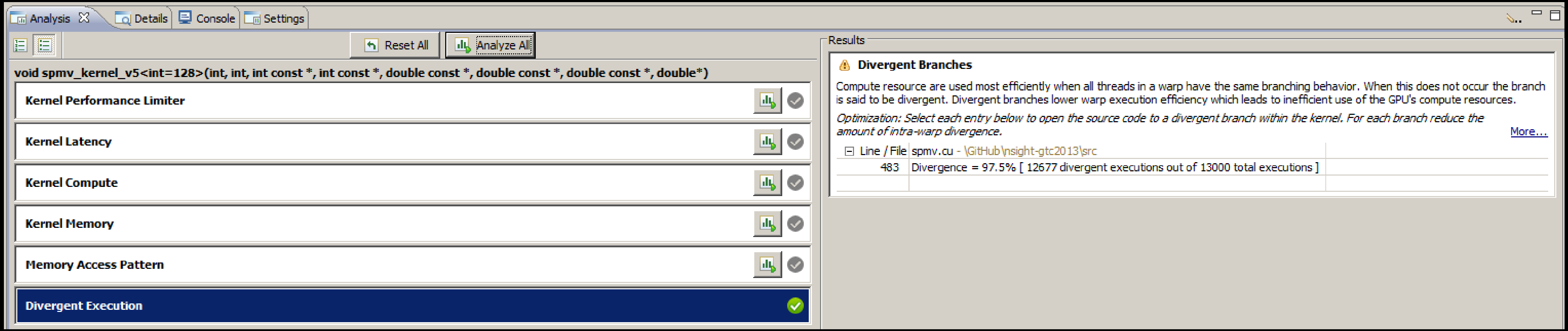
- Memory accesses are good enough

One Half Warp Per Row

- Branch divergence induce latency
- We have 16.32% of divergent branches



Branch Divergence

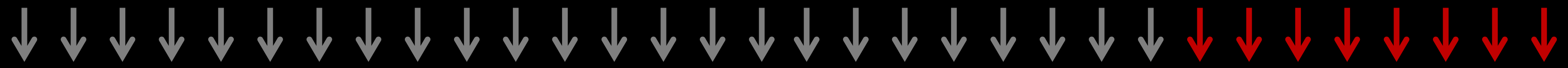


Branch Divergence

- Execution Time = Time of If branch + Time of Else branch

```
if( threadIdx.x % 32 < 24 ) {
    … // do some long computation
}
```

```
else {
    … // do some long computation
}
```



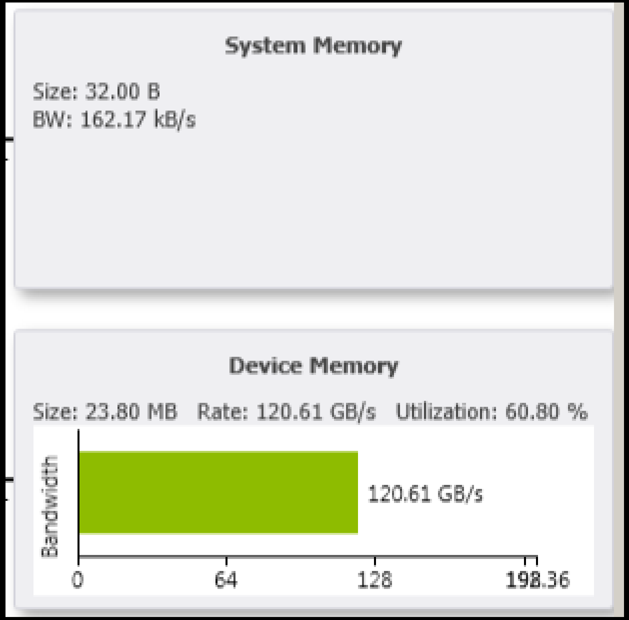
One Half Warp Per Row

- We fix branch divergence
- It is faster: 29.60ms

Kernel	Time	Speedup
Original version	104.72ms	
LDG to load A	125.67ms	0.83x
LDG to load X	98.30ms	1.07x
Coalescing with 4 Threads	45.61ms	2.30x
1 Warp per Row	37.50ms	2.79x
½ Warp per Row	35.81ms	2.93x
No divergence	29.60ms	3.54x

One Half Warp Per Row

- DRAM utilization: 60.80%



- We are still far from the peak…

So Far

- We have consecutively:
  - Improved caching using __ldg (use with care)
  - Improved coalescing
  - Improved control flow efficiency
  - Improved branching

- Our new kernel is 3.5x faster than our first implementation

- Tools helped us a lot

C:\Windows\system32\cmd.exe

```
###############################################################################
**                          B I C G S T A B   S O L V E R                      **
###############################################################################

** DEVICE      : Tesla K20c (ECC: OFF)                                        **

###############################################################################

** SYSTEM      : res/matrix.inp                                                **

###############################################################################

** INIT. RESID.:  [   1.212971e-001   0.000000e+000   0.000000e+000   1.243311e-001 ]     **

###############################################################################

** ITERATION  0:  [   5.009870e-002   2.509095e-003   2.442529e-003   1.381766e-003 ]     **
** ITERATION  1:  [   6.322283e-002   3.624363e-003   3.439345e-003   1.459566e-003 ]     **
** ITERATION  2:  [   1.944435e-002   3.175072e-004   3.108480e-004   4.101967e-004 ]     **
** ITERATION  3:  [   1.179491e-002   9.633129e-005   9.554327e-005   2.494020e-004 ]     **
** ITERATION  4:  [   1.517741e-002   1.351845e-004   1.323939e-004   3.272856e-004 ]     **
** ITERATION  5:  [   2.840113e-002   2.370584e-004   2.333890e-004   6.397188e-004 ]     **
** ITERATION  6:  [   8.465301e-003   9.483242e-005   9.256901e-005   1.726512e-004 ]     **
** ITERATION  7:  [   2.497275e-003   2.221739e-005   2.213925e-005   6.087546e-005 ]     **
** ITERATION  8:  [   3.931372e-003   3.762042e-005   3.804746e-005   9.449076e-005 ]     **
** ITERATION  9:  [   1.004664e-003   7.813901e-006   7.722009e-006   2.470211e-005 ]     **
** ITERATION 10:  [   1.348178e-003   1.450667e-005   1.451499e-005   3.084324e-005 ]     **
** ITERATION 11:  [   3.147213e-004   3.016084e-006   2.968251e-006   7.588855e-006 ]     **
** ITERATION 12:  [   2.560259e-004   2.530426e-006   2.474577e-006   6.138979e-006 ]     **
** ITERATION 13:  [   1.941811e-004   2.010254e-006   1.992610e-006   4.670605e-006 ]     **
** ITERATION 14:  [   1.344858e-004   1.313841e-006   1.286352e-006   3.325935e-006 ]     **
** ITERATION 15:  [   2.946048e-004   3.318294e-006   3.216173e-006   7.020198e-006 ]     **
** ITERATION 16:  [   1.254350e-004   1.372731e-006   1.317983e-006   3.036414e-006 ]     **

###############################################################################

** FINAL RESID.:  [   2.529559e-005   2.054403e-007   1.903810e-007   6.569666e-007 ]     **

###############################################################################

** ELAPSED TIME:    29.199ms                                                   **

###############################################################################

Press any key to continue . . . _
```

ITERATION 6

Next Kernel

- We are satisfied with the performance of spmv_kernel
- We move to the next kernel: jacobi_smooth

	Function Name	Module ID	Function ID	Count	Device %	Device Time (µs)	Min (µs)	Avg (µs)	Max (µs)
1	spmv_kernel_v6<int=128>	46	7	71	6.82	13,264.048	183.924	186.818	189.972
2	jacobi_smooth_kernel_v0<int=256>	47	4	35	1.59	3,088.802	86.441	88.251	96.746
3	dot_kernel_v0<int=256>	44	1	70	0.56	1,091.926	11.041	15.599	17.986
4	l2_norm_kernel_v0<int=256>	45	1	36	0.43	830.548	21.858	23.071	23.842
5	axpbypcz_kernel<int=256>	44	5	34	0.37	719.280	20.579	21.155	21.635
6	axpby_kernel<int=256>	44	4	37	0.31	593.790	15.105	16.048	16.833
7	reduce_kernel<int=256>	44	3	70	0.13	261.372	3.104	3.734	4.353
8	reduce_l2_norm_kernel<int=256>	45	2	36	0.09	171.217	4.416	4.756	6.208
9	jacobi_invert_diag_kernel_v0<int=256>	47	1	1	0.06	109.068	109.068	109.068	109.068

C:\Windows\system32\cmd.exe

```
##############################################################################
**                        B I C G S T A B   S O L V E R                     **
##############################################################################

** DEVICE      : Tesla K20c (ECC: OFF)                                      **

##############################################################################

** SYSTEM      : res/matrix.inp                                              **

##############################################################################

** INIT. RESID.:  [   1.212971e-001   0.000000e+000   0.000000e+000   1.243311e-001 ]     **

##############################################################################

** ITERATION  0:  [   5.009870e-002   2.509095e-003   2.442529e-003   1.381766e-003 ]     **
** ITERATION  1:  [   6.322283e-002   3.624363e-003   3.439345e-003   1.459566e-003 ]     **
** ITERATION  2:  [   1.944435e-002   3.175072e-004   3.108480e-004   4.101967e-004 ]     **
** ITERATION  3:  [   1.179491e-002   9.633129e-005   9.554327e-005   2.494020e-004 ]     **
** ITERATION  4:  [   1.517741e-002   1.351845e-004   1.323939e-004   3.272856e-004 ]     **
** ITERATION  5:  [   2.840113e-002   2.370584e-004   2.333890e-004   6.397188e-004 ]     **
** ITERATION  6:  [   8.465301e-003   9.483242e-005   9.256901e-005   1.726512e-004 ]     **
** ITERATION  7:  [   2.497275e-003   2.221739e-005   2.213925e-005   6.087546e-005 ]     **
** ITERATION  8:  [   3.931372e-003   3.762042e-005   3.804746e-005   9.449076e-005 ]     **
** ITERATION  9:  [   1.004664e-003   7.813901e-006   7.722009e-006   2.470211e-005 ]     **
** ITERATION 10:  [   1.348178e-003   1.450667e-005   1.451499e-005   3.084324e-005 ]     **
** ITERATION 11:  [   3.147213e-004   3.016084e-006   2.968251e-006   7.588855e-006 ]     **
** ITERATION 12:  [   2.560259e-004   2.530426e-006   2.474577e-006   6.138979e-006 ]     **
** ITERATION 13:  [   1.941811e-004   2.010254e-006   1.992610e-006   4.670605e-006 ]     **
** ITERATION 14:  [   1.344858e-004   1.313841e-006   1.286352e-006   3.325935e-006 ]     **
** ITERATION 15:  [   2.946048e-004   3.318294e-006   3.216173e-006   7.020198e-006 ]     **
** ITERATION 16:  [   1.254350e-004   1.372731e-006   1.317983e-006   3.036414e-006 ]     **

##############################################################################

** FINAL RESID.:  [   2.529559e-005   2.054403e-007   1.903810e-007   6.569666e-007 ]     **

##############################################################################

** ELAPSED TIME:    27.189ms                                                 **

##############################################################################

Press any key to continue . . .
```

What Have You Seen?

- An iterative method to optimize your GPU code
  - Trace your application
  - Identify the hot spot and profile it
  - Identify the performance limiter
  - Optimize the code
  - Iterate

- A way to conduct that method with Nvidia tools