OPENACC COURSE
PERFORMANCE OPTIMIZATION

29.10.2021 | J. KRAUS, M. HRYWNIAK (NVIDIA)

OUTLINE

- Performance Analysis Tools
- Memory coalescing
- Loop optimizations

GENERAL NOTES

Important flags for NVHPC Compiler

Building with lightweight debug information:

```
-gpu=lineinfo
```

Check compiler output: `-Minfo=accel`

NVHPC RUNTIME MEASUREMENTS

For quick sanity checks

- Applications compiled with NVHPC compiler: Analyze via environment variables
- Maybe simplest/quickest check

Set `NVCOMPILER_ACC_TIME=1` for lightweight profiler on time of data movements and kernels

NVHPC RUNTIME MEASUREMENTS

```
$ export NVCOMPILER_ACC_TIME=1
$ srun -n1 ./spmv
Runtime 0.007972 s.

Accelerator Kernel Timing data
/p/home/jusers/hrywniak1/jusuf/openacc-4/C/task0/spmv.c
  main  NVIDIA  devicenum=0
    time(us): 307,478
    37: data region reached 2 times
        37: data copyin transfers: 168
             device time(us): total=222,883 max=1,527 min=143 avg=1,326
        57: data copyout transfers: 4
             device time(us): total=5,145 max=1,312 min=1,277 avg=1,286
    41: compute region reached 10 times
        41: kernel launched 10 times
            grid: [63443]  block: [128]
             device time(us): total=79,450 max=7,948 min=7,940 avg=7,945
            elapsed time(us): total=79,667 max=8,011 min=7,957 avg=7,966
```

NSIGHT PROFILER SUITE

Nsight Systems and Nsight Compute

- Comes with HPC SDK, also standalone
- Profiles application, including CUDA Kernels and API calls
- Supports OpenACC
- *Systems* for whole application, *Compute* for kernel tuning
- Generates performance reports, timelines; measures events and metrics

- https://developer.nvidia.com/tools-overview

NSIGHT SYSTEMS COMMAND-LINE PROFILER

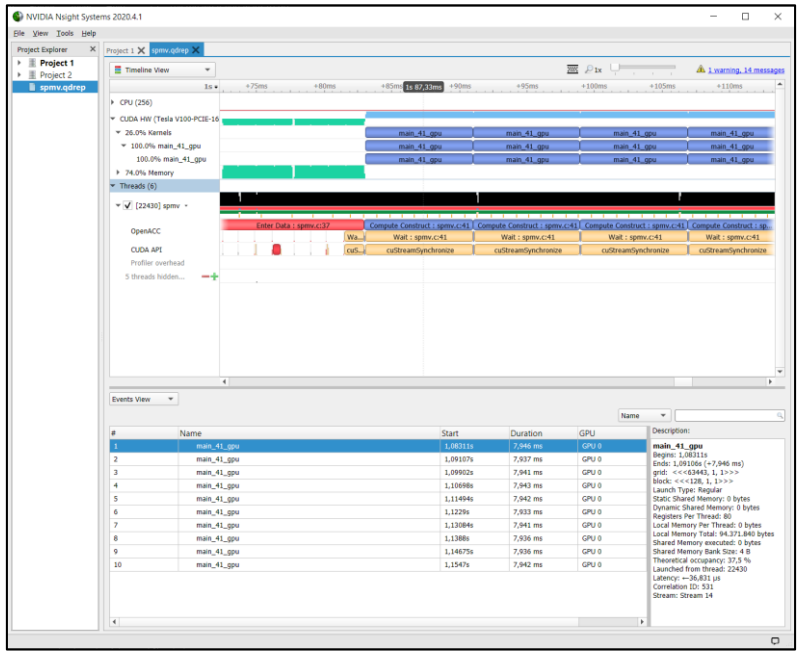
```
$ srun -n1 nsys profile -t cuda,openacc \
-f true -o spmv --stats=true ./spmv
```

- Always records a report (*.qdrep)
- Reports customizable
- Forgot *--stats*?
  nsys stats can post-process any qdrep

```
Time(%)  Total Time (ns)  Num Calls   Average    Minimum   Maximum          Name
 -------  ---------------  ---------  ----------  --------  --------  --------------------
    60.4         82200447         12   6850037.3   1272194   7950555  cuStreamSynchronize
    25.3         34383053        172    199901.5      1560   6964642  cuEventSynchronize
    10.0         13578282          1  13578282.0  13578282  13578282  cuMemHostAlloc
     2.7          3721103          6    620183.8    143751   1490944  cuMemAlloc_v2
     0.7           954802        168      5683.3      4600     27610  cuMemcpyHtoDAsync_v2
     0.4           533741          1    533741.0    533741    533741  cuMemAllocHost_v2
     0.3           364570        174      2095.2      1820      4311  cuEventRecord
     0.1           119510          1    119510.0    119510    119510  cuModuleLoadDataEx
     0.1           114440         10     11444.0      8460     32760  cuLaunchKernel
     0.0            28350          4      7087.5      4810     11910  cuMemcpyDtoHAsync_v2
     0.0            16230          1     16230.0     16230     16230  cuStreamCreate
     0.0             5380          4      1345.0       450      2530  cuEventCreate

CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances   Average   Minimum  Maximum     Name
 -------  ---------------  ---------  ---------  -------  -------  -----------
   100.0         79415454         10  7941545.4  7932394  7948329  main_41_gpu

CUDA Memory Operation Statistics (by time):

 Time(%)  Total Time (ns)  Operations   Average   Minimum  Maximum      Operation
 -------  ---------------  ----------  ---------  -------  -------  ------------------
    97.8        220817077         168  1314387.4   138847  1522486  [CUDA memcpy HtoD]
     2.2          4926174           4  1231543.5  1111096  1271928  [CUDA memcpy DtoH]

CUDA Memory Operation Statistics (by size in KiB):

    Total     Operations   Average    Minimum    Maximum       Operation
-----------  ----------  ---------  ---------  ---------  ------------------
  63442.195           4  15860.549  14290.383  16383.938  [CUDA memcpy DtoH]
2719562.816         168  16187.874   1684.441  16384.000  [CUDA memcpy HtoD]
```

NSIGHT SYSTEMS GUI

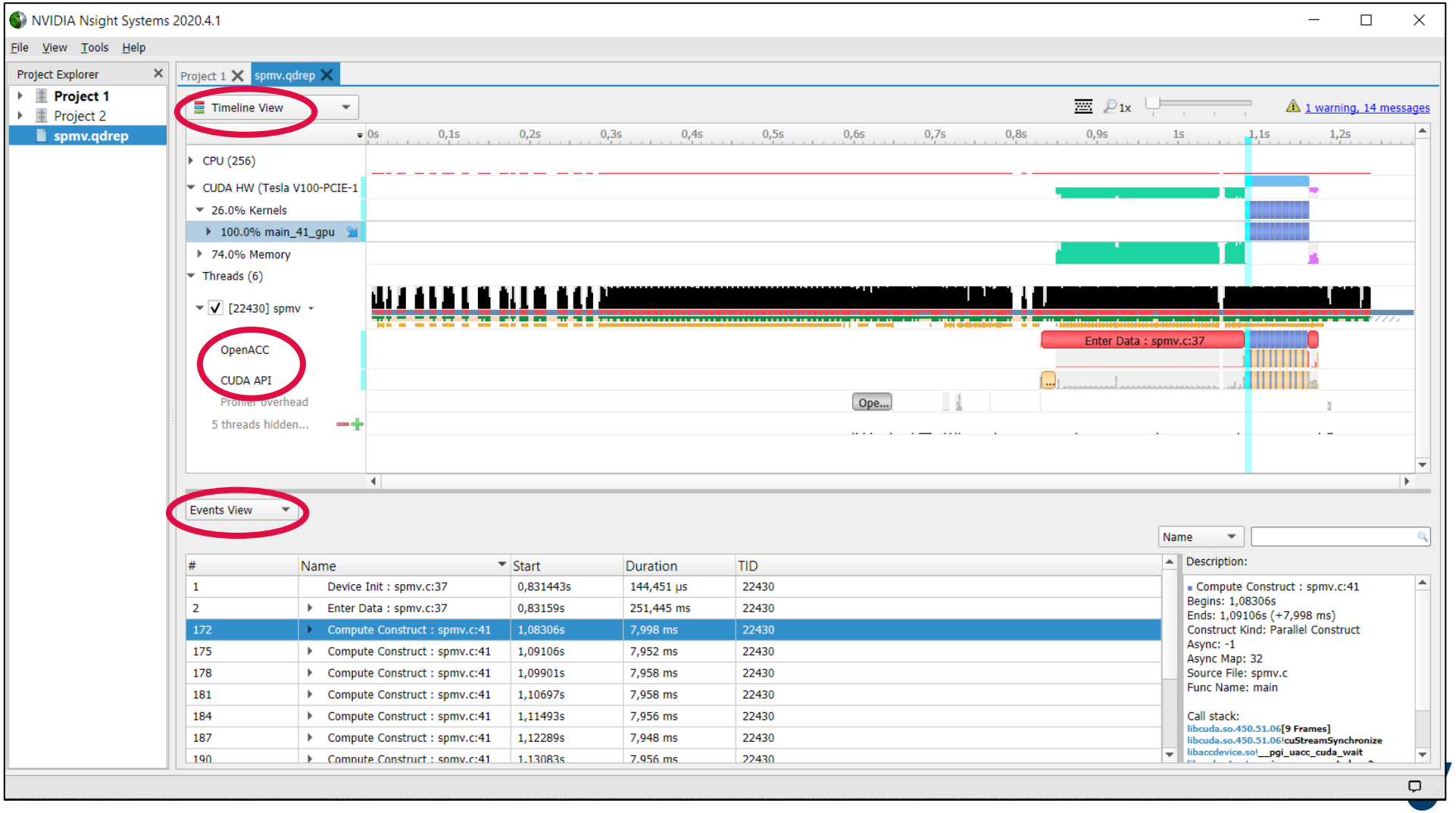
- Graphical, interactive profiler
- Comes with HPC SDK, also standalone
- Nice visualizations, quick insight
- Timeline traces for OpenACC, OpenMP, CUDA, MPI, etc.



- https://docs.nvidia.com/nsight-systems/UserGuide/index.html

NSIGHT SYSTEMS GUI

Timeline, Traces and Events View



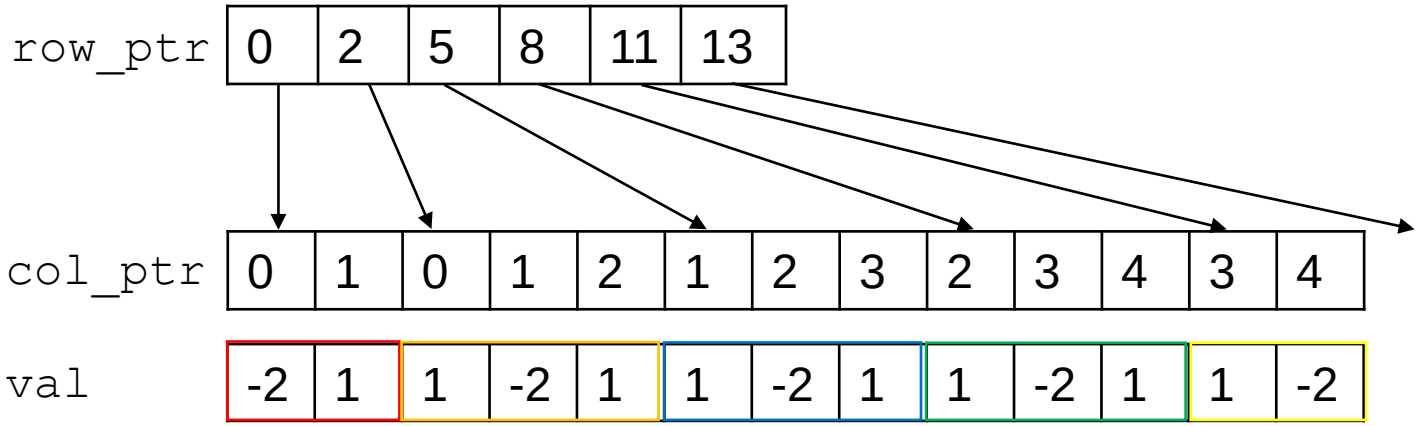
#	Name	Start	Duration	TID
1	Device Init : spmv.c:37	0,831443s	144,451 µs	22430
2	Enter Data : spmv.c:37	0,83159s	251,445 ms	22430
172	Compute Construct : spmv.c:41	1,08306s	7,998 ms	22430
175	Compute Construct : spmv.c:41	1,09106s	7,952 ms	22430
178	Compute Construct : spmv.c:41	1,09901s	7,958 ms	22430
181	Compute Construct : spmv.c:41	1,10697s	7,958 ms	22430
184	Compute Construct : spmv.c:41	1,11493s	7,956 ms	22430
187	Compute Construct : spmv.c:41	1,12289s	7,948 ms	22430
190	Compute Construct : spmv.c:41	1,13083s	7,956 ms	22430

CSR SPARSE MATRIX STORAGE

Compressed Sparse Row

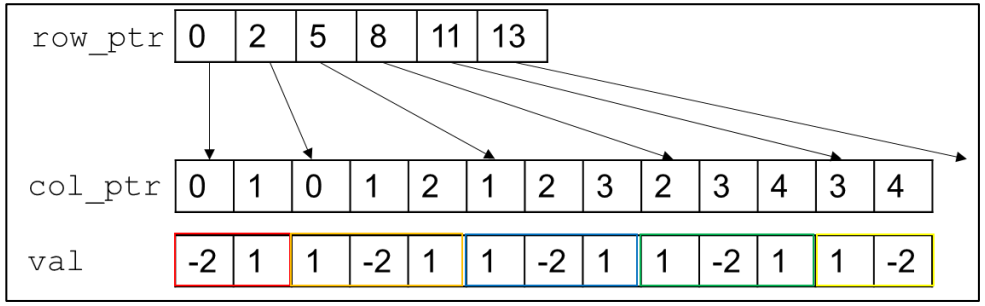
	0	1	2	3	4
0	-2	1	0	0	0
1	1	-2	1	0	0
2	0	1	-2	1	0
3	0	0	1	-2	1
4	0	0	0	1	-2

	0	1	2	3	4
0	-2	1			
1	1	-2	1		
2		1	-2	1	
3			1	-2	1
4				1	-2



SPARSE MATRIX VECTOR PRODUCT (SPMV)

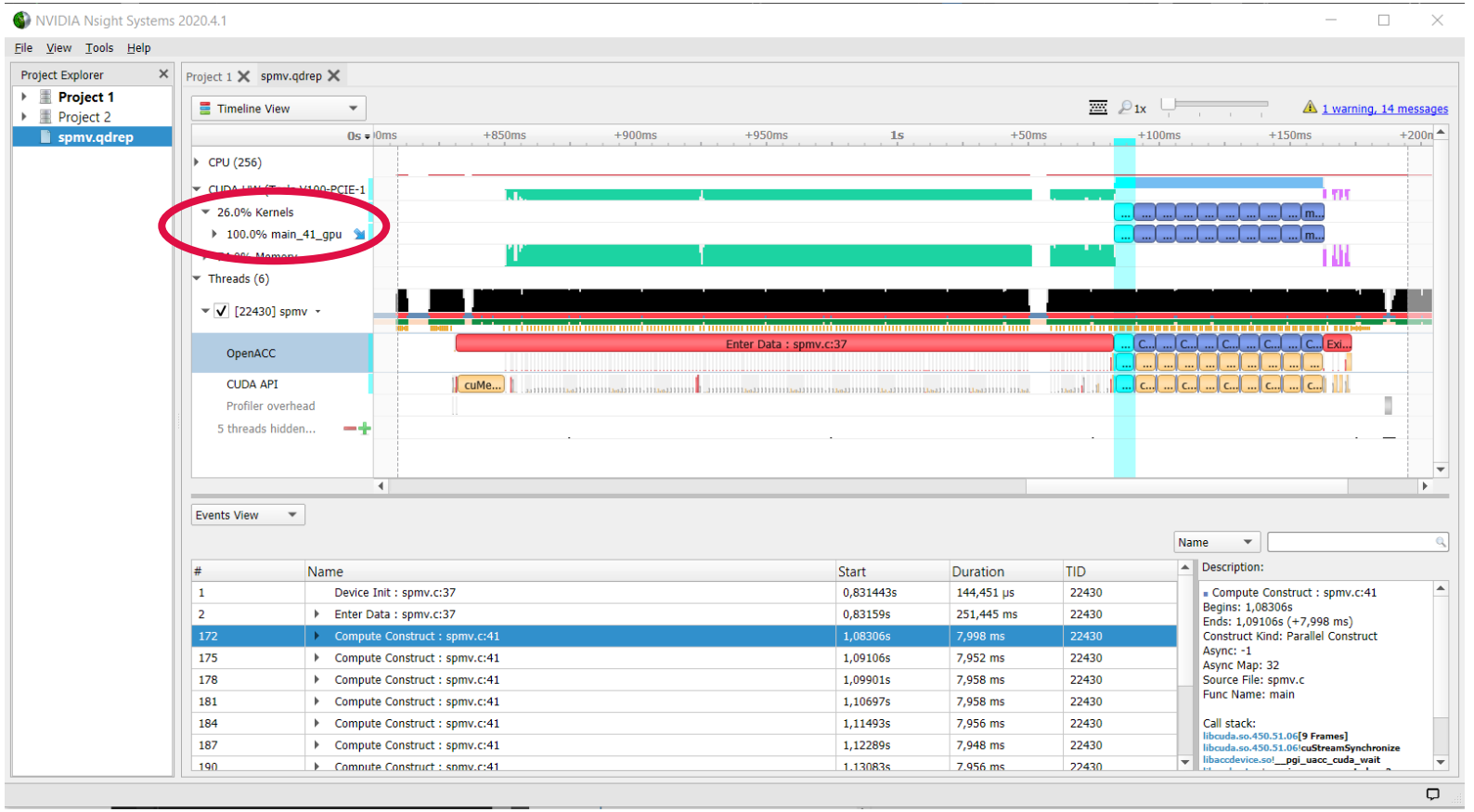
```
42:#pragma acc parallel loop
43:for (int row=0; row<num_rows; ++row)
44:{
45:   y[row] = 0.0;
46:   const int row_start = row_ptr[row];
47:   const int row_end = row_ptr[row+1];
48:   for (int col_idx=row_start; col_idx<row_end; ++col_idx)
49:   {
50:     y[row] += val[col_idx] * x[ col_ptr[col_idx] ];
51:   }
52:
53:}
```



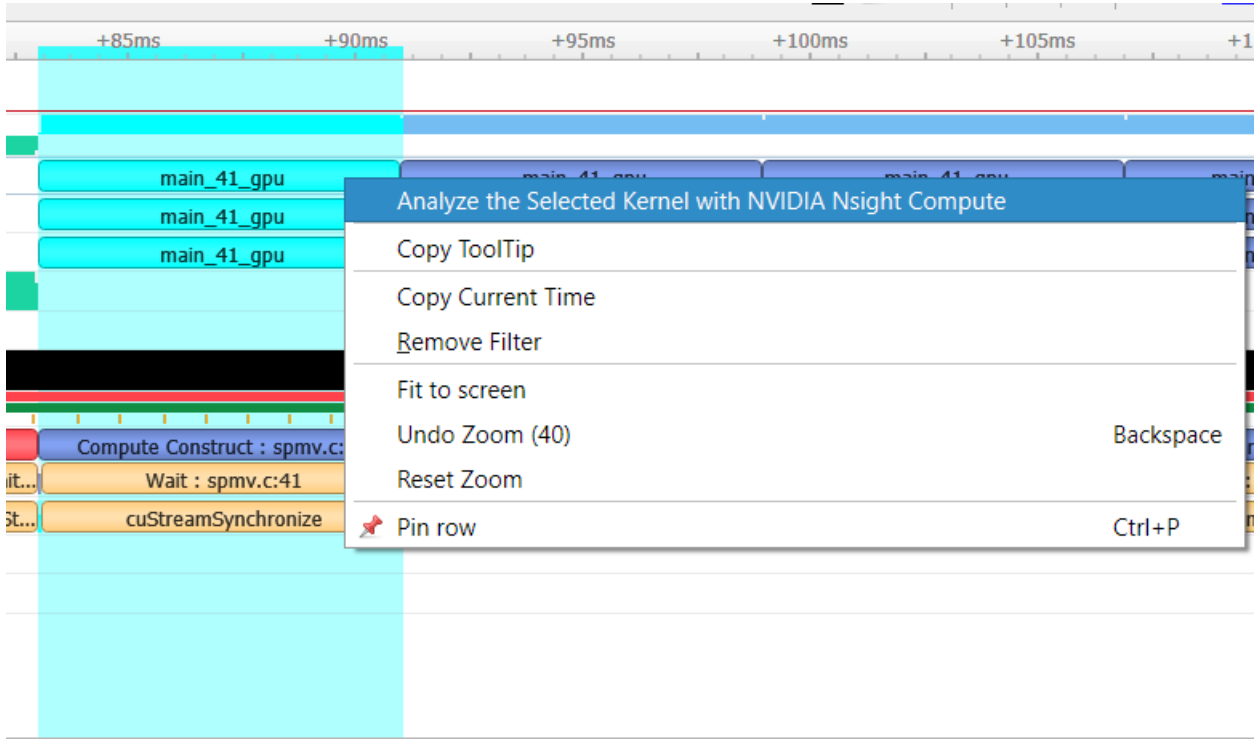
SPMV ON V100

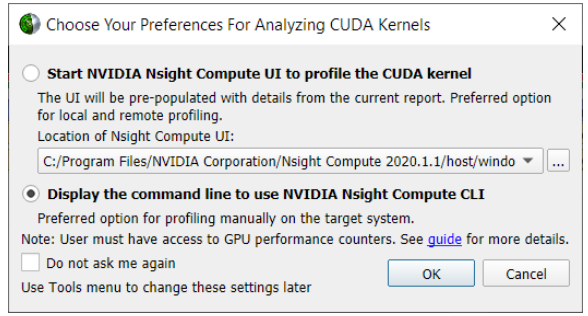
```
$ make
nvc -fast -acc -Minfo=accel spmv.c -o spmv
main:
     37, Generating copyin(col_ptr[:num_vals],row_ptr[:num_rows+1],val[:num_vals],x[:num_rows])
           [if not already present]
         Generating copy(y[:num_rows]) [if not already present]
     41, Generating Tesla code
         43, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         48, #pragma acc loop seq
     48, Complex loop carried dependence of y-> prevents parallelization
         Loop carried dependence of y-> prevents parallelization
         Loop carried backward dependence of y-> prevents vectorization
srun -n 1 ./spmv
Runtime 0.007956 s.
```

SPMV ON V100



SPMV ON V100





Will give command similar to:

```
ncu --kernel-regex main_41_gpu \
--launch-skip 0 --launch-count 1 ./spmv
```

SPMV ON V100

Running Nsight Compute

```
$ srun -n1 ncu --kernel-regex main_41_gpu --launch-skip 0 --launch-count 1 ./spmv
==PROF== Connected to process 31183 (/p/home/jusers/hrywniak1/jusuf/openacc-4/C/task0/spmv)
==PROF== Profiling "main_41_gpu": 0%....50%....100%Runtime 0.070514 s.
 - 19 passes
==PROF== Disconnected from process 31183
[31183] spmv@127.0.0.1
  main_41_gpu, 2020-Nov-03 17:23:36, Context 1, Stream 13
    Section: GPU Speed Of Light
    ------------------------------ -------------- ------------------------------
    DRAM Frequency                     cycle/usecond                         876.61
    SM Frequency                       cycle/nsecond                           1.25
    Elapsed Cycles                             cycle                        9931386
    Memory [%]                                   %                          92.57
    SOL DRAM                                     %                          92.57
    Duration                                 msecond                           7.94
    SOL L1/TEX Cache                             %                          34.62
    SOL L2 Cache                                 %                          33.42
    SM Active Cycles                           cycle                     9918546.07
    SM [%]                                       %                           3.11
    ------------------------------ -------------- ------------------------------
    OK    The kernel is utilizing greater than 80.0% of the available compute or
          memory performance of the device. To further improve performance, work
          will likely need to be shifted from the most utilized to another unit.
          Start by analyzing workloads in the Memory Workload Analysis section.
```

Command-line output

Initial profile: Need more data

→ `--set full`

Rerun, what does this mean?

```
WRN   Uncoalesced global
access, expected 1015027
transactions, got 8120203
(8.00x) at PC
0x2ac760f91590
```

JÜLICH Forschungszentrum

SPMV ON V100

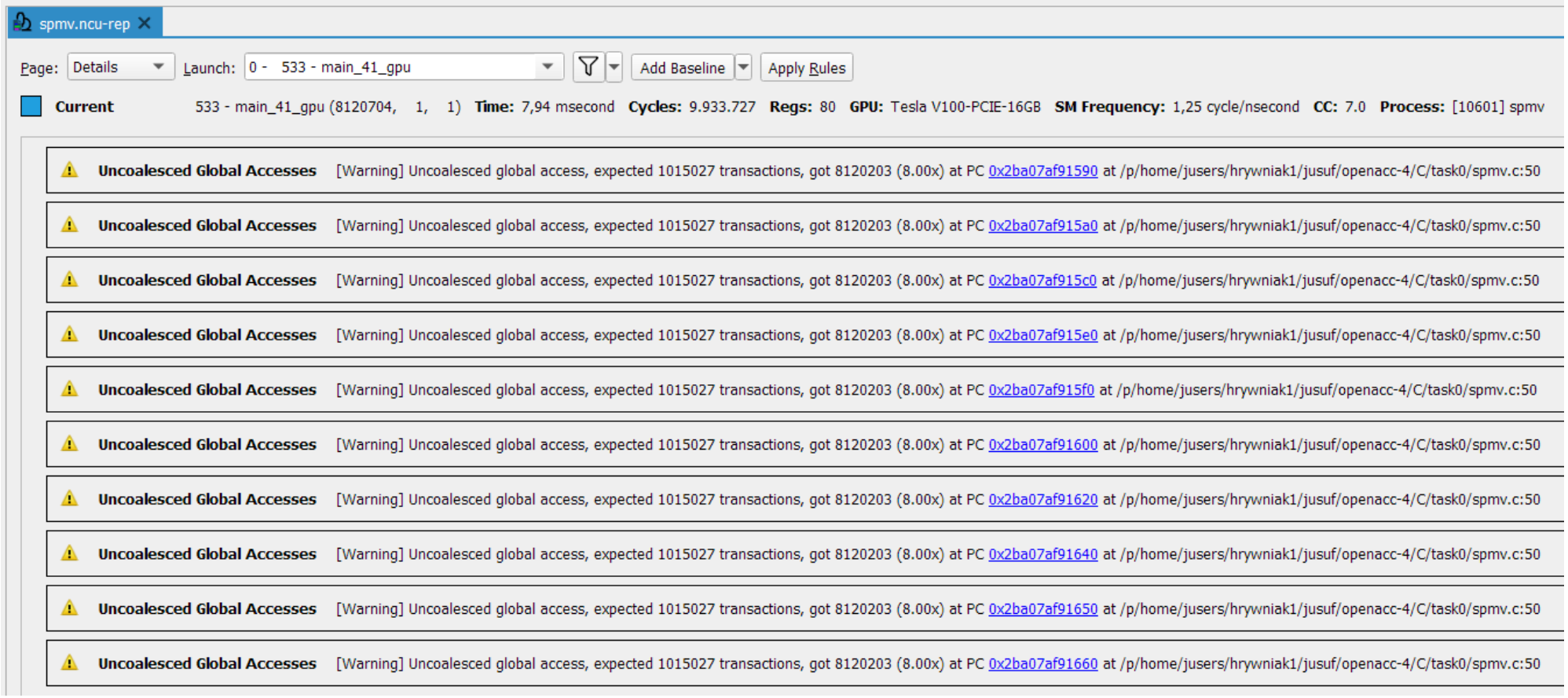
```
$ make
nvc -fast -acc -gpu=lineinfo -Minfo=accel spmv.c -o spmv
main:
     37, Generating copyin(col_ptr[:num_vals],row_ptr[:num_row                ls],x[:num_rows])
[if not already present]
         Generating copy(y[:num_rows]) [if not already present]
     41, Generating Tesla code
         43, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         48, #pragma acc loop seq
     48, Complex loop carried dependence of y-> prevents parallelization
         Loop carried dependence of y-> prevents parallelization
         Loop carried backward dependence of y-> prevents vectorization
```

Better Profiling Information

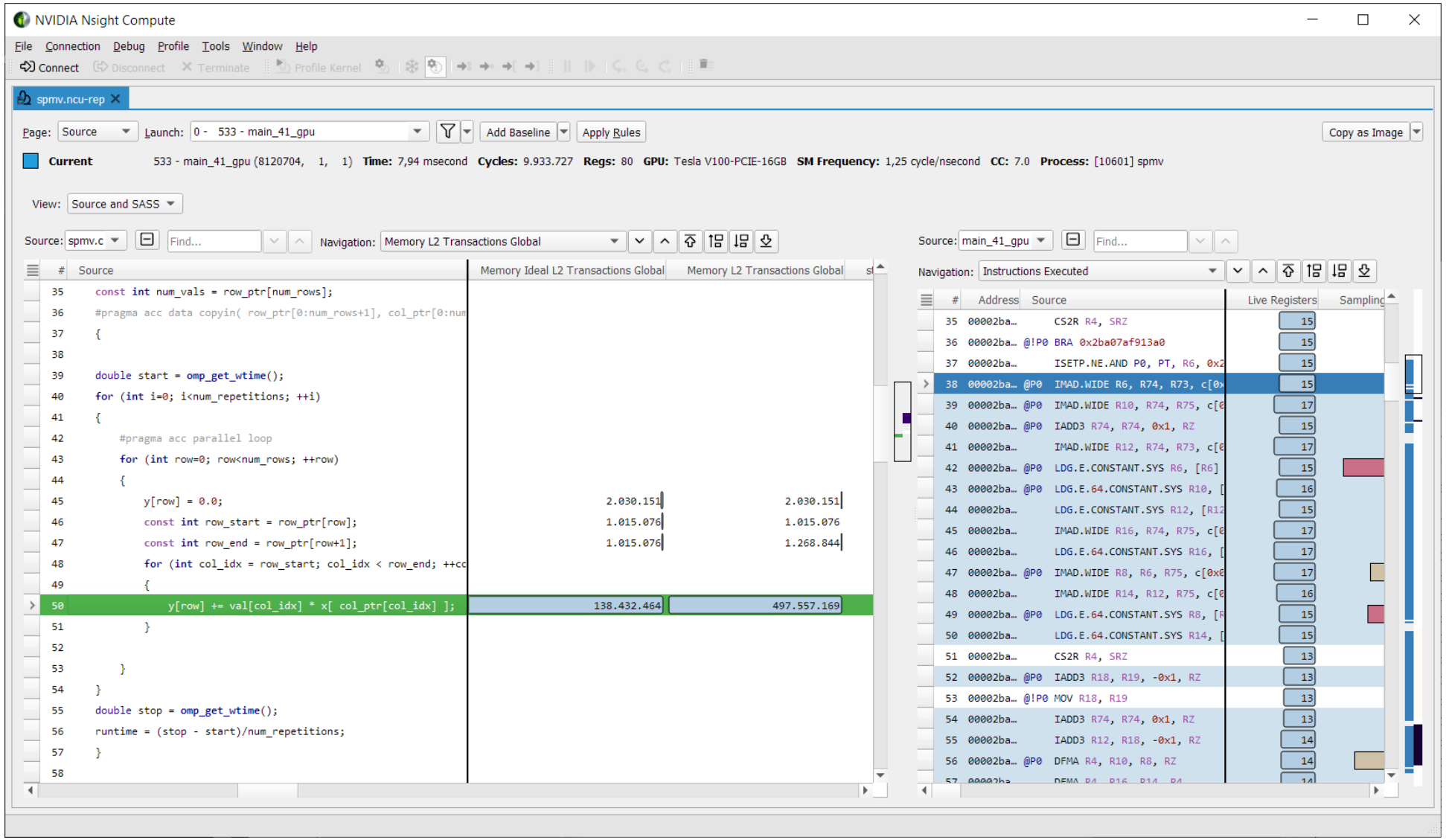
SPMV ON V100

Nsight Compute GUI

Add `-o spmv` to generate a `*.ncu-rep` report file

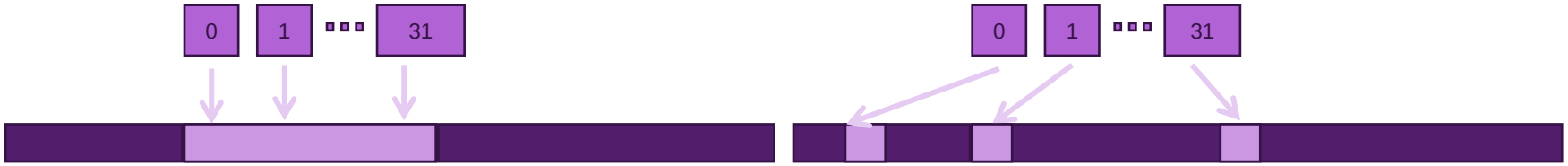


SPMV ON V100



MEMORY COALESCING

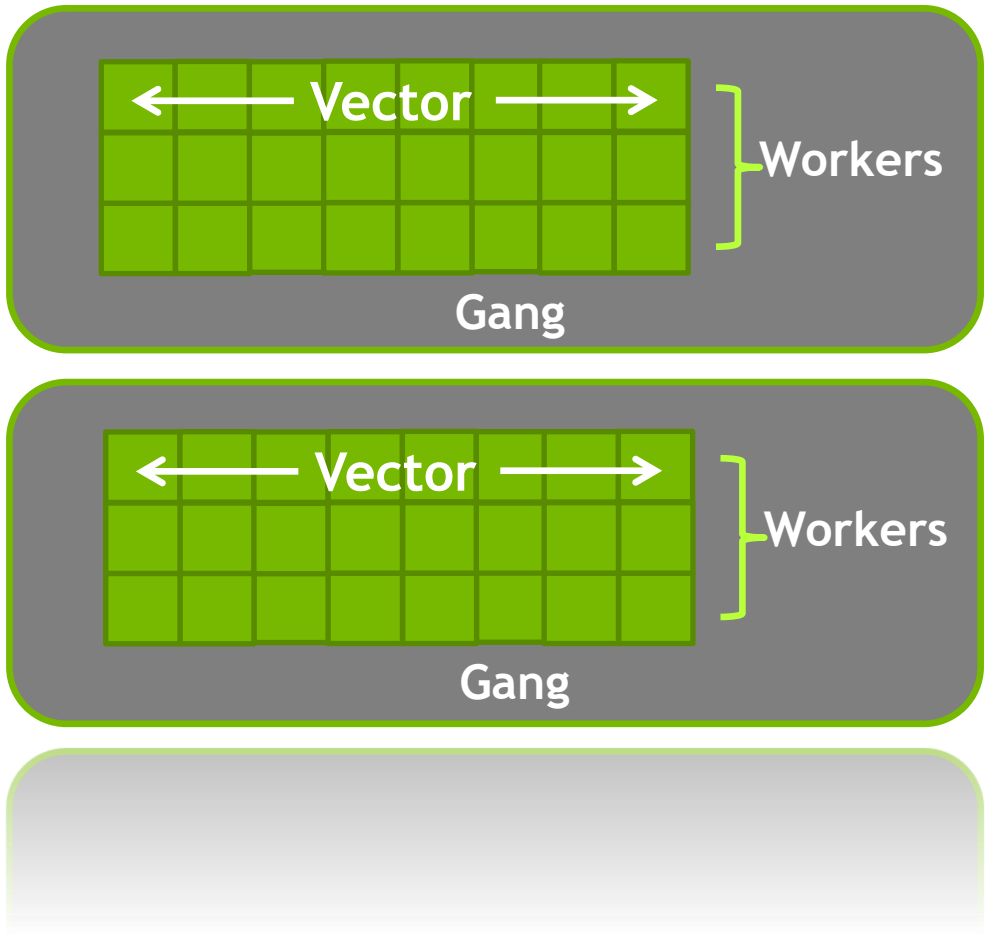
- Coalesced access:
  - A group of 32 contiguous threads („warp“) accessing adjacent words
  - Few transactions and high utilization
- Uncoalesced access:
  - A warp of 32 threads accessing scattered words
  - Many transactions and low utilization
- For best performance threadIdx.x should access contiguously



Coalesced

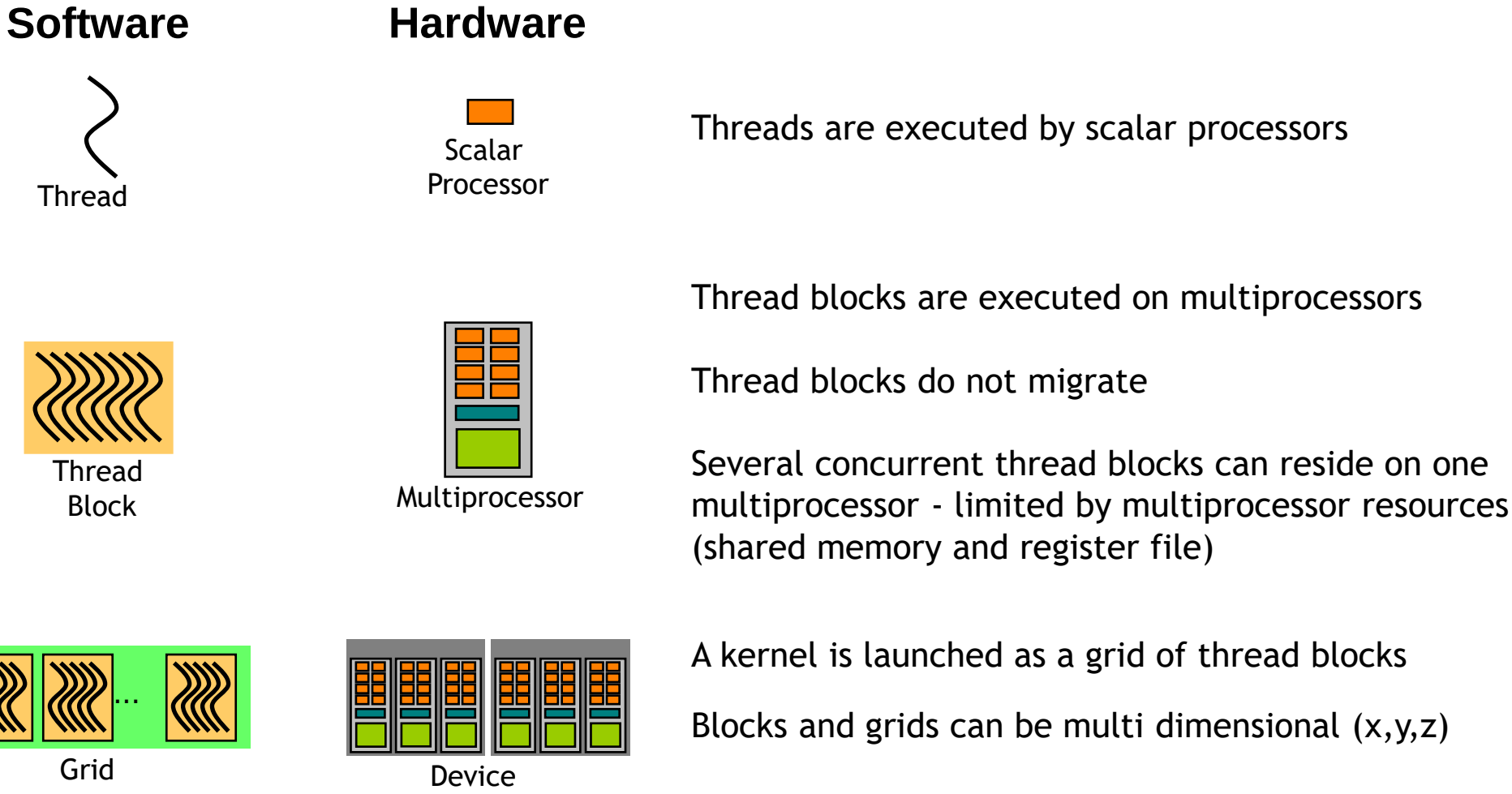
Uncoalesced

OPENACC: 3 LEVELS OF PARALLELISM

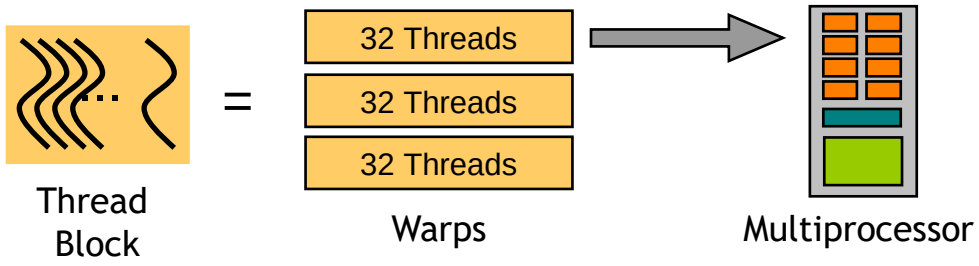


- Vector threads work in lockstep (SIMD/SIMT parallelism)
- Workers have 1 or more vectors
- Gangs have 1 or more workers and share resources (such as a cache, the SM, etc.)
- Multiple gangs work independently of each other

CUDA EXECUTION MODEL



CUDA WARPS



A thread block consists of a groups of warps

A warp is executed physically in parallel (SIMT) on a multiprocessor

Currently all NVIDIA GPUs use a warp size of 32

MAPPING OPENACC TO CUDA

- The compiler is free to do what it wants
- In general
  - gang: mapped to blocks (COARSE GRAIN)
  - worker: mapped to threads (FINE GRAIN)
  - vector: mapped to threads (FINE SIMD/SIMT)
- Exact mapping is compiler dependent
- Performance Tips
  - Use a vector size that is divisible by 32
  - Block size is `num_workers * vector_length`

OPENACC GANG, WORKER, VECTOR CLAUSES

- Gang, worker, vector can be added to a loop clause
- Control the size using the following clauses on the parallel region
  - Parallel: `num_gangs(n), num_workers(n), vector_length(n)`
  - Kernels: `gang(n), worker(n), vector(n)`

```
#pragma acc parallel loop gang worker
for (int row=0; row<num_rows; ++row)
{
  #pragma acc loop vector
  for (int col_idx=row_start; col_idx<row_end; ++col_idx)
```

gang, worker, vector appear once per parallel region

UNDERSTANDING COMPILER OUTPUT

```
41, Generating Tesla code
        43, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

- Compiler is reporting how it is assigning work to the device
  - Gang is being mapped to blockIdx.x
  - Vector is being mapped to threadIdx.x
  - Worker is not used
- This application has a thread block size of 128 and launches as many blocks as necessary

SPMV

```
42:#pragma acc parallel loop
43:for (int row=0; row<num_rows; ++row)
44:{
45:  y[row] = 0.0;
46:  const int row_start = row_ptr[row];
47:  const int row_end = row_ptr[row+1];
48:  for (int col_idx=row_start; col_idx<row_end; ++col_idx)
49:  {
50:    y[row] += val[col_idx] * x[ col_ptr[col_idx] ];
51:  }
52:
53:}
```

Want this loop to parallelize with vector parallelism

```
48, Complex loop carried dependence of y-> prevents parallelization
        Loop carried dependence of y-> prevents parallelization
        Loop carried backward dependence of y-> prevents vectorization
```

SPMV

```
42:#pragma acc parallel loop
43:for (int row=0; row<num_rows; ++row)
44:{
45:  double y_tmp = 0.0;
46:  const int row_start = row_ptr[row];
47:  const int row_end = row_ptr[row+1];
48:  for (int col_idx=row_start; col_idx<row_end; ++col_idx)
49:  {
50:    y_tmp += val[col_idx] * x[ col_ptr[col_idx] ];
51:  }
52:  y[row] = y_tmp;
53:}
```

Sum up in temporary to remove loop carried dependency

SPMV ON V100

```
$ make
nvc -fast -acc -gpu=lineinfo -Minfo=accel spmv.c -o spmv
main:
     37, Generating copyin(col_ptr[:num_vals],row_ptr[:num_rows+1],val[:num_vals],x[:num_rows])
           [if not already present]
         Generating copy(y[:num_rows]) [if not already present]
     41, Generating Tesla code
         43, #pragma acc loop gang /* blockIdx.x */
         48, #pragma acc loop vector(128) /* threadIdx.x */
             Generating implicit reduction(+:y_tmp)
     48, Loop is parallelizable
srun -n 1 ./spmv
Runtime 0.015231 s. (was 0.007956 s)
```

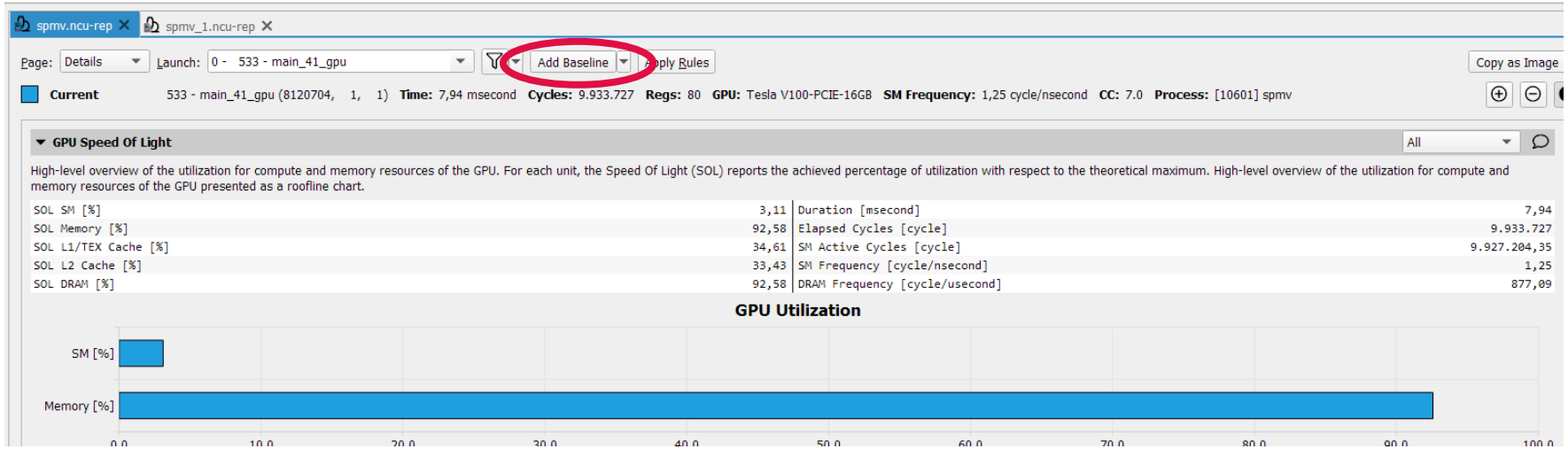
SPMV ON V100

Calling ncu again, using a different output file:

Recommendations

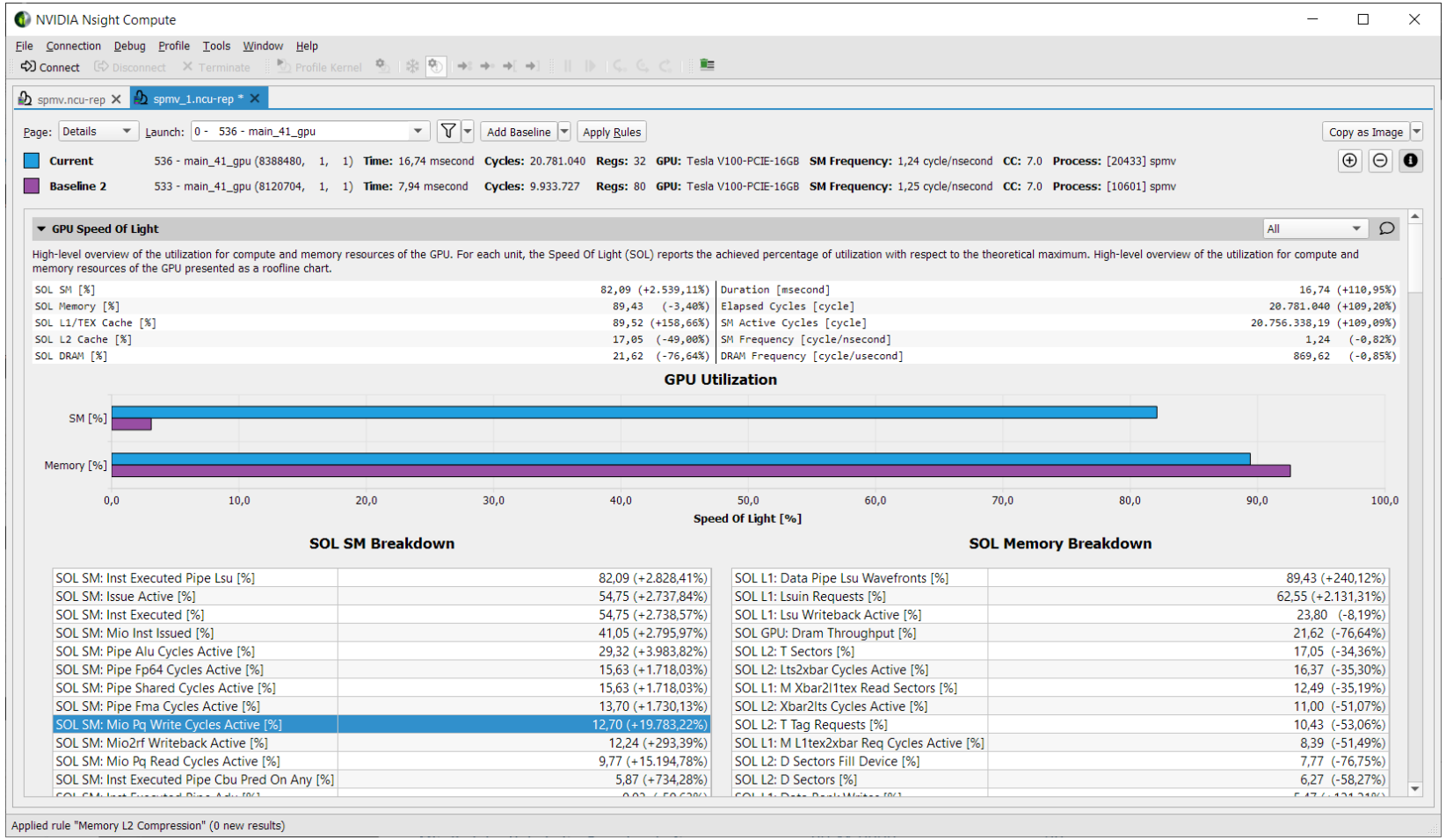
⚠	**Uncoalesced Global Accesses**	[Warning] Uncoalesced global access, expected 54181060 transactions, got 104055413 (1.92x) at PC 0x2b052af91750 at /p/home/jusers/hrywniak1/jusuf/openacc-4/C/task1/spmv.c:50
⚠	**Uncoalesced Global Accesses**	[Warning] Uncoalesced global access, expected 56683807 transactions, got 60724303 (1.07x) at PC 0x2b052af91730 at /p/home/jusers/hrywniak1/jusuf/openacc-4/C/task1/spmv.c:50
⚠	**Uncoalesced Global Accesses**	[Warning] Uncoalesced global access, expected 32401606 transactions, got 34422450 (1.06x) at PC 0x2b052af91700 at /p/home/jusers/hrywniak1/jusuf/openacc-4/C/task1/spmv.c:50

Use previous kernel as baseline



JÜLICH Forschungszentrum

SPMV ON V100



Recommendations

Bottleneck The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing workloads in the Memory Workload Analysis section.

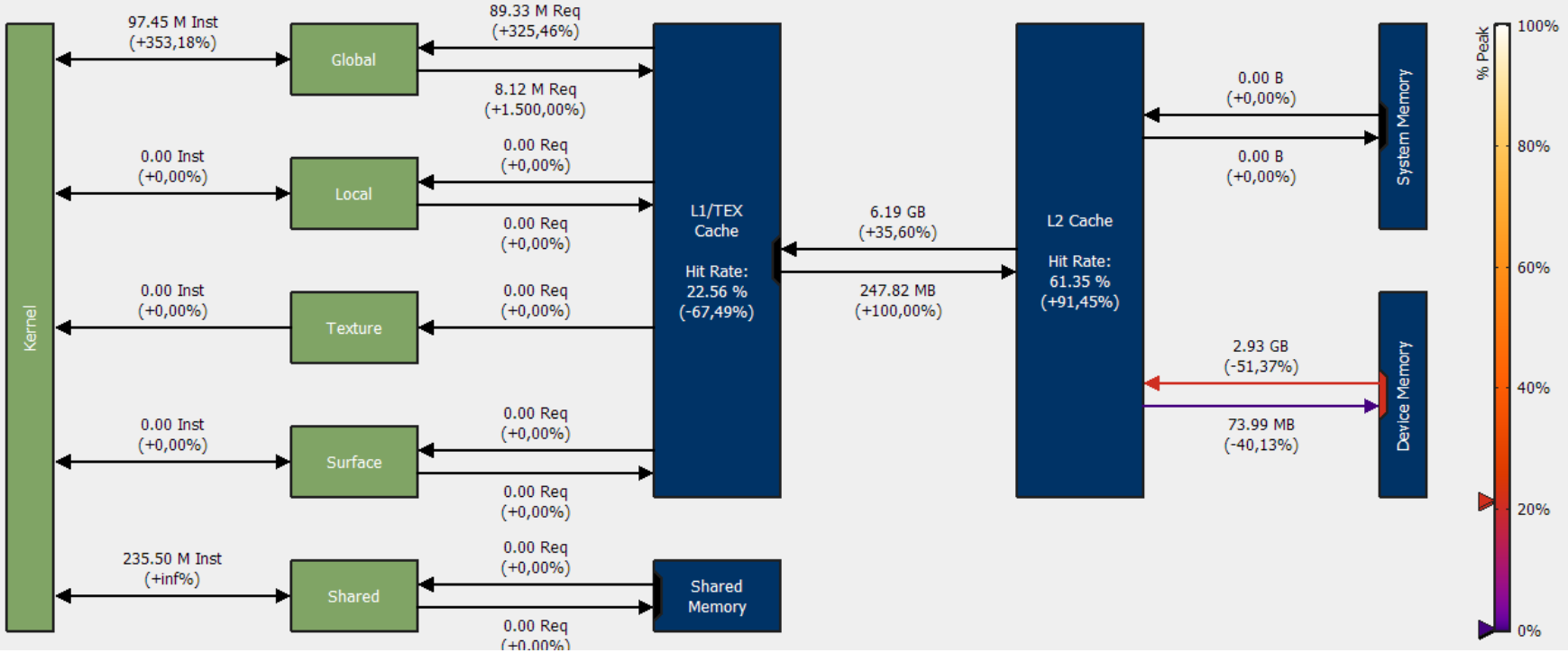
SPMV ON V100

Memory Workload Analysis — All

Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). Detailed chart of the memory units. Detailed tables with data for each memory unit.

Memory Throughput [Gbyte/second]	192,56 (-76,84%)	Mem Busy [%]	89,43 (+173,20%)
L1/TEX Hit Rate [%]	22,56 (-67,49%)	Max Bandwidth [%]	62,55 (-32,43%)
L2 Hit Rate [%]	61,35 (+91,45%)	Mem Pipes Busy [%]	82,09 (+2.539,11%)

Memory Chart



SPMV ON V100

Built-in rules and heuristics can help find possible issues

Compute Workload Analysis ⚠

Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.

Executed Ipc Elapsed [inst/cycle]	2,19 (+2.738,57%)	SM Busy [%]	54,80 (+2.739,67%)
Executed Ipc Active [inst/cycle]	2,19 (+2.740,41%)	Issue Slots Busy [%]	54,80 (+2.739,67%)
Issued Ipc Active [inst/cycle]	2,19 (+2.739,67%)		

Memory Workload Analysis

Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). Detailed chart of the memory units. Detailed tables with data for each memory unit.

Memory Throughput [Gbyte/second]	192,56 (-76,84%)	Mem Busy [%]	89,43 (+173,20%)
L1/TEX Hit Rate [%]	22,56 (-67,49%)	Max Bandwidth [%]	62,55 (-32,43%)
L2 Hit Rate [%]	61,35 (+91,45%)	Mem Pipes Busy [%]	82,09 (+2.539,11%)

Scheduler Statistics ⚠

Summary of the activity of the schedulers issuing instructions. Each scheduler maintains a pool of warps that it can issue instructions for. The upper bound of warps in the pool (Theoretical Warps) is limited by the launch configuration. On every cycle each scheduler checks the state of the allocated warps in the pool (Active Warps). Active warps that are not stalled (Eligible Warps) are ready to issue their next instruction. From the set of eligible warps the scheduler selects a single warp from which to issue one or more instructions (Issued Warp). On cycles with no eligible warps, the issue slot is skipped and no instruction is issued. Having many skipped issue slots indicates poor latency hiding.

Active Warps Per Scheduler [warp]	15,89 (+183,90%)	Instructions Per Active Issue Slot [inst/cycle]	1 (+0,00%)
Eligible Warps Per Scheduler [warp]	1,42 (+3.860,48%)	No Eligible [%]	45,18 (-53,92%)
Issued Warp Per Scheduler	0,55 (+2.723,96%)	One or More Eligible [%]	54,82 (+2.723,96%)

Warp State Statistics ⚠

Analysis of the states in which all warps spent cycles during the kernel execution. The warp states describe a warp's readiness or inability to issue its next instruction. The warp cycles per instruction define the latency between two consecutive instructions. The higher the value, the more warp parallelism is required to hide this latency. For each warp state, the chart shows the average number of cycles spent in that state per issued instruction. Stalls are not always impacting the overall performance nor are they completely avoidable. Only focus on stall reasons if the schedulers fail to issue every cycle. When executing a kernel with mixed library and user code, these metrics show the combined values.

Warp Cycles Per Issued Instruction [warp]	28,99 (-89,95%)	Avg. Active Threads Per Warp	32 (+0,00%)
Warp Cycles Per Issue Active [warp]	28,99 (-89,95%)	Avg. Not Predicated Off Threads Per Warp	21,11 (-30,55%)
Warp Cycles Per Executed Instruction [cycle]	28,99 (-89,95%)		

SPMV ON V100

Built-in rules and heuristics can help find possible issues

Recommendations

⚠ **CPI Stall 'Barrier'**	[Warning] On average each warp of this kernel spends 11.1 cycles being stalled waiting for sibling warps at a CTA barrier. This represents about 38.1% of the total average of 29.0 cycles between issuing two instructions. A high number of warps waiting at a barrier is commonly caused by diverging code paths before a barrier that causes some warps to wait a long time until other warps reach the synchronization point. Whenever possible try to divide up the work into blocks of uniform workloads. Use the Source View's sampling columns to identify which barrier instruction causes the most stalls and optimize the code executed before that synchronization point first.
⚠ **Thread Divergence**	[Warning] Instructions are executed in warps, which are groups of 32 threads. Optimal instruction throughput is achieved if all 32 threads of a warp execute the same instruction. The chosen launch configuration, early thread completion, and divergent flow control can significantly lower the number of active threads in a warp per cycle. This kernel achieves an average of 32.0 threads being active per cycle. This is further reduced to 21.1 threads per warp due to predication. The compiler may use predication to avoid an actual branch. Instead, all instructions are scheduled, but a per-thread condition code or predicate controls which threads execute the instructions. Try to avoid different execution paths within a warp when possible. In addition, ensure your kernel makes use of Independent Thread Scheduling, which allows a warp to reconverge after a data-dependent conditional block by explicitly calling __syncwarp().

SPMV ON V100

```
42:#pragma acc parallel loop
43:for (int row=0; row<num_rows; ++row)
44:{
45:  double y_tmp = 0.0;
46:  const int row_start = row_ptr[row];
47:  const int row_end = row_ptr[row+1];
48:  for (int col_idx=row_start; col_idx<row_end; ++col_idx)
49:  {
50:    y_tmp += val[col_idx] * x[col_ptr[col_idx]];
51:  }
52:  y[row
53:}
```

gang

vector(128)

```
41, Generating Tesla code
        43, #pragma acc loop gang /* blockIdx.x */
        48, #pragma acc loop vector(128) /* threadIdx.x */
            Generating implicit reduction(+:y_tmp)
```

PROVIDING MORE INFORMATION TO THE COMPILER

- We know that each row of the used Matrix has only 27 elements
- Using 128 threads for 27 elements does not make sense
- Let‘s tell the compiler to use fewer threads for each row

SPMV ON V100

```
42:#pragma acc parallel loop vector_length(32)
43:for (int row=0; row<num_rows; ++row)
44:{
45:  double y_tmp = 0.0;
46:  const int row_start = row_ptr[row];
47:  const int row_end = row_ptr[row+1];
48:  for (int col_idx=row_start; col_idx<row_end; ++col_idx)
49:  {
50:    y_tmp += val[col_idx] * x[col_ptr[col_idx]];
51:  }
52:  y[row
53:}
```

gang

vector(32)

```
41, Generating Tesla code
        43, #pragma acc loop gang /* blockIdx.x */
        48, #pragma acc loop vector(32) /* threadIdx.x */
            Generating implicit reduction(+:y_tmp)
```

SPMV ON V100

```
$ make
nvc -fast -acc -gpu=lineinfo -Minfo=accel spmv.c -o spmv
main:
     37, Generating
copyin(col_ptr[:num_vals],row_ptr[:num_rows+1],val[:num_vals],x[:num_rows]) [if not already
present]
         Generating copy(y[:num_rows]) [if not already present]
     41, Generating Tesla code
         43, #pragma acc loop gang /* blockIdx.x */
         48, #pragma acc loop vector(32) /* threadIdx.x */
             Generating implicit reduction(+:y_tmp)
     48, Loop is parallelizable
srun -n 1 ./spmv
Runtime 0.006170 s. (was 0.015231 s)
```

KEEPING THE CODE PERFORMANCE PORTABLE

- The device_type clause allows device specific tuning without harming performance portability
- All clauses following a device_type clause only apply for the given target:

```
#pragma acc parallel loop device_type(NVIDIA) vector_length(32)
for (int row=0; row<num_rows; ++row)
{
```

TASKS

- Task 0: Coalescing memory accesses (repeat)
- Task 1: Use vector_length to improve the warp execution efficiency (repeat what was shown)
- Task 2: Use a new Nsight Compute profile to further improve performance.
  - Hint: Add worker level parallelism to increase the block size to 128 threads (required to get full occupancy).

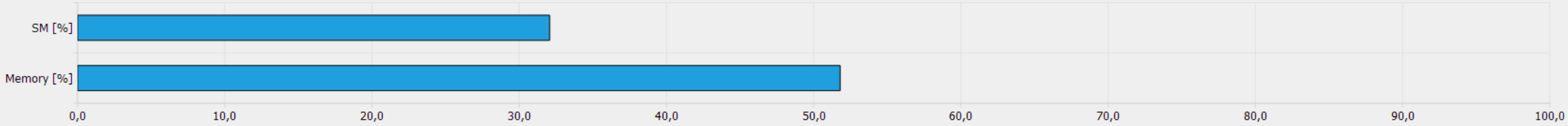
SPMV ON V100

GPU Speed Of Light ⚠ All

High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

SOL SM [%]	32,07	Duration [msecond]	6,43
SOL Memory [%]	51,80	Elapsed Cycles [cycle]	8.007.139
SOL L1/TEX Cache [%]	43,93	SM Active Cycles [cycle]	7.996.284,86
SOL L2 Cache [%]	41,31	SM Frequency [cycle/nsecond]	1,25
SOL DRAM [%]	51,80	DRAM Frequency [cycle/usecond]	874,32

GPU Utilization



Recommendations

⚠ **Bottleneck** [Warning] This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of this device. Achieved compute throughput and/or memory bandwidth below 60.0% of peak typically indicate latency issues. Look at Scheduler Statistics and Warp State Statistics for potential reasons.

⚠ **Issue Slot Utilization** [Warning] Every scheduler is capable of issuing one instruction per cycle, but for this kernel each scheduler only issues an instruction every 2.0 cycles. This might leave hardware resources underutilized and may lead to less optimal performance. Out of the maximum of 16 warps per scheduler, this kernel allocates an average of 15.67 active warps per scheduler, but only an average of 1.16 warps were eligible per cycle. Eligible warps are the subset of active warps that are ready to issue their next instruction. Every cycle with no eligible warp results in no instruction being issued and the issue slot remains unused. To increase the number of eligible warps either increase the number of active warps or reduce the time the active warps are stalled.

⚠ **CPI Stall 'Long Scoreboard'** [Warning] On average each warp of this kernel spends 17.4 cycles being stalled waiting for a scoreboard dependency on a L1TEX (local, global, surface, texture) operation. This represents about 56.3% of the total average of 31.0 cycles between issuing two instructions. To reduce the number of cycles waiting on L1TEX data accesses verify the memory access patterns are optimal for the target architecture, attempt to increase cache hit rates by increasing data locality or by changing the cache configuration, and consider moving frequently used data to shared memory.

SPMV ON V100

- All hints (latency-bound, low utilization, stalls) → resources on the GPU are idle
- „Occupancy is the percentage of the hardware‘s ability [...] that is actively in use“

Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	50	Block Limit Registers [block]	48
Theoretical Active Warps per SM [warp]	32	Block Limit Shared Mem [block]	384
Achieved Occupancy [%]	49,36	Block Limit Warps [block]	64
Achieved Active Warps Per SM [warp]	31,59	Block Limit SM [block]	32

SPMV

```
42:#pragma acc parallel loop device_type(NVIDIA) gang worker \
       vector_length(32)
43:for (int row=0; row<num_rows; ++row)
44:{
45:   double y_tmp = 0.0;
46:   const int row_start = row_ptr[row];
47:   const int row_end = row_ptr[row+1];
48:   for (int col_idx=row_start; col_idx<row_end; ++col_idx)
49:   {
50:     y_tmp += val[col_idx] * x[ col_ptr[col_idx] ];
51:   }
52:   y[row] = y_tmp;
53:}
```

gang, worker(4)

vector(32)

```
41, Generating Tesla code
        43, #pragma acc loop gang, worker(4) /* blockIdx.x threadIdx.y */
        48, #pragma acc loop vector(32) /* threadIdx.x */
            Generating implicit reduction(+:y_tmp)
            Vector barrier inserted for vector loop reduction
```

UNDERSTANDING COMPILER OUTPUT (RECAP)

```
41, Generating Tesla code
        43, #pragma acc loop gang, worker(4) /* blockIdx.x threadIdx.y */
        48, #pragma acc loop vector(32) /* threadIdx.x */
            Generating implicit reduction(+:y_tmp)
            Vector barrier inserted for vector loop reduction
```

- Compiler is reporting how it is assigning work to the device
  - Gang is being mapped to blockIdx.x
  - Worker is being mapped to threadIdx.y
  - Vector is being mapped to threadIdx.x
- This application has a thread block size of 4x32 and launches as many blocks as necessary

SPMV ON V100

```
$ make
nvc -fast -acc -gpu=lineinfo -Minfo=accel spmv.c -o spmv
main:
     37, Generating copyin(col_ptr[:num_vals],row_ptr[:num_rows+1],val[:num_vals],x[:num_rows])
                 [if not already present]
         Generating copy(y[:num_rows]) [if not already present]
     41, Generating Tesla code
         43, #pragma acc loop gang, worker(4) /* blockIdx.x threadIdx.y */
         48, #pragma acc loop vector(32) /* threadIdx.x */
             Generating implicit reduction(+:y_tmp)
             Vector barrier inserted for vector loop reduction
     48, Loop is parallelizable
srun -n 1 ./spmv
Runtime 0.004860 s. (was 0.006170 s)
```

CONCLUSIONS

- The Nsight Profilers can be used to identify performance bottlenecks in applications and OpenACC Kernels
- Coalescing memory accesses is important for performance
- Using loop clauses allows to provide runtime information (approximate length of matrix rows) to the compiler for better performance.