

OpenACC Performance Optimization

04.10.2017 | J. Kraus (NVIDIA)

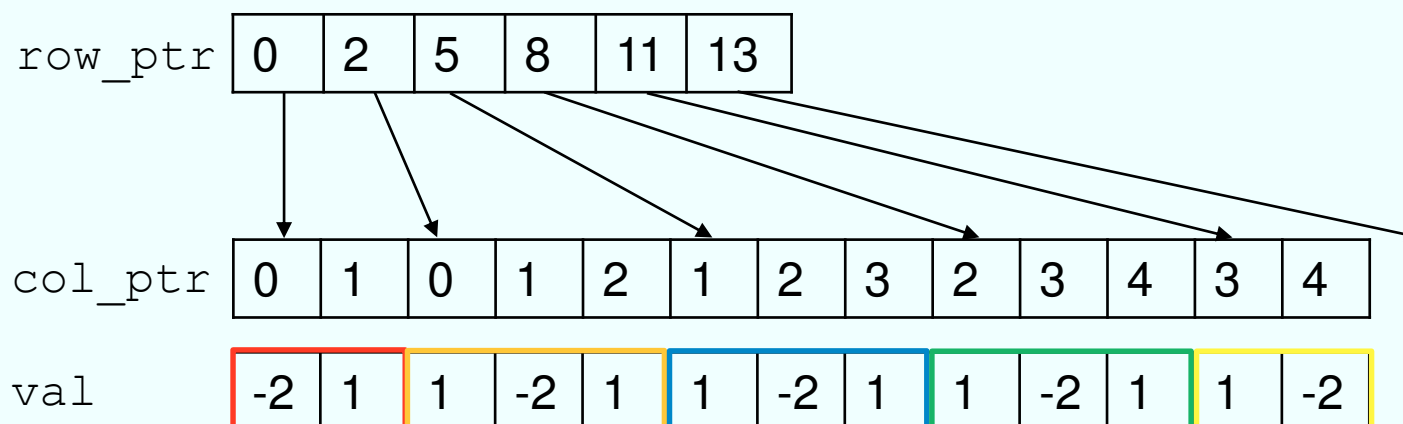
Outline

- Memory coalescing
- Loop optimizations

CSR sparse matrix storage

	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 K80

```
pgcc -fast -acc -ta=tesla -Minfo=accel spmv.c -o spmv
```

```
main:
```

```
    36, Generating
```

```
copyin(row_ptr[:num_rows+1],col_ptr[:num_vals],val[:num_vals],x[:num_rows])
```

```
    Generating copy(y[:num_rows])
```

```
    42, Accelerator kernel generated
```

```
    Generating Tesla code
```

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

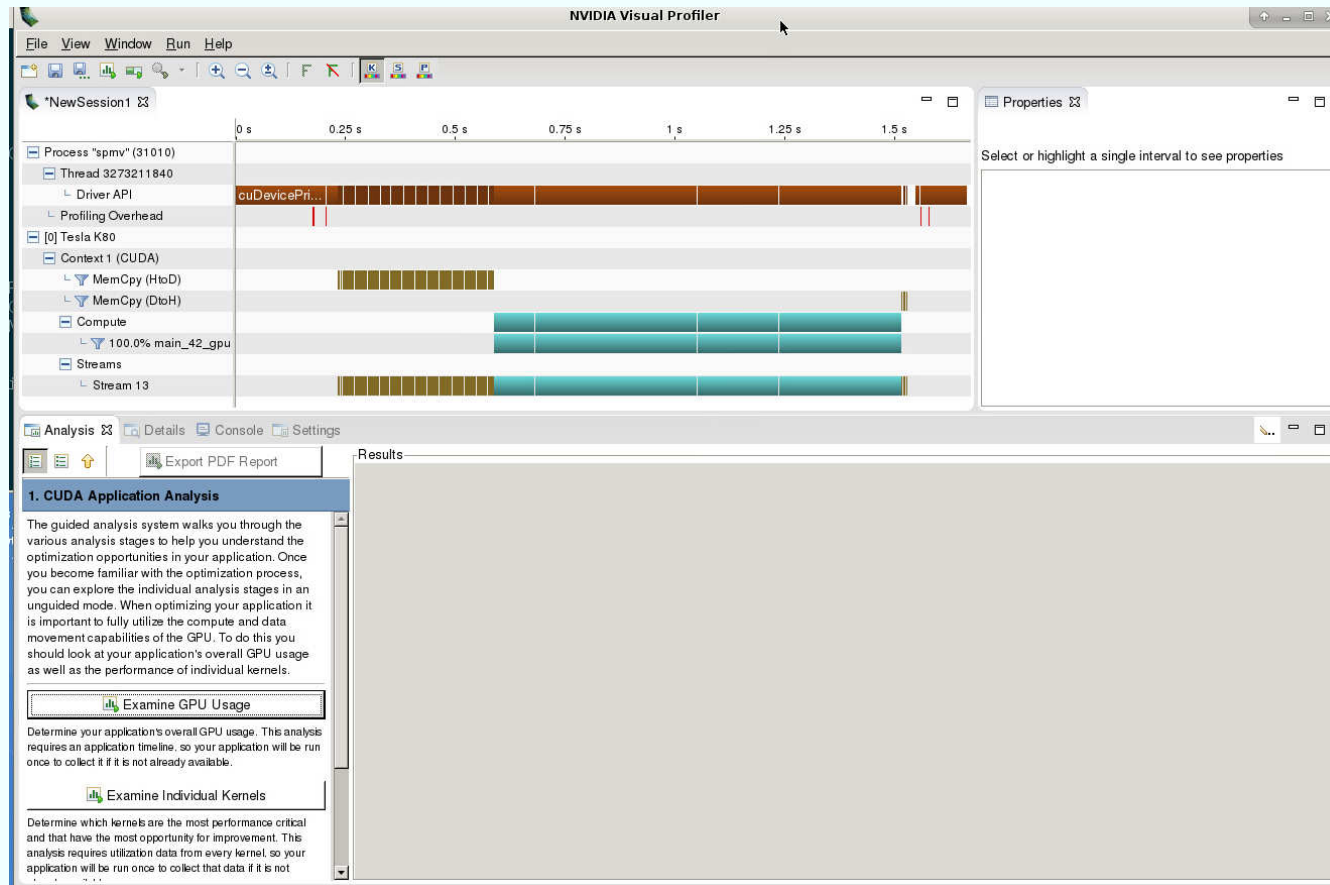
```
    48, Complex loop carried dependence of y-> prevents parallelization
```

```
    Loop carried reuse of y-> prevents parallelization
```

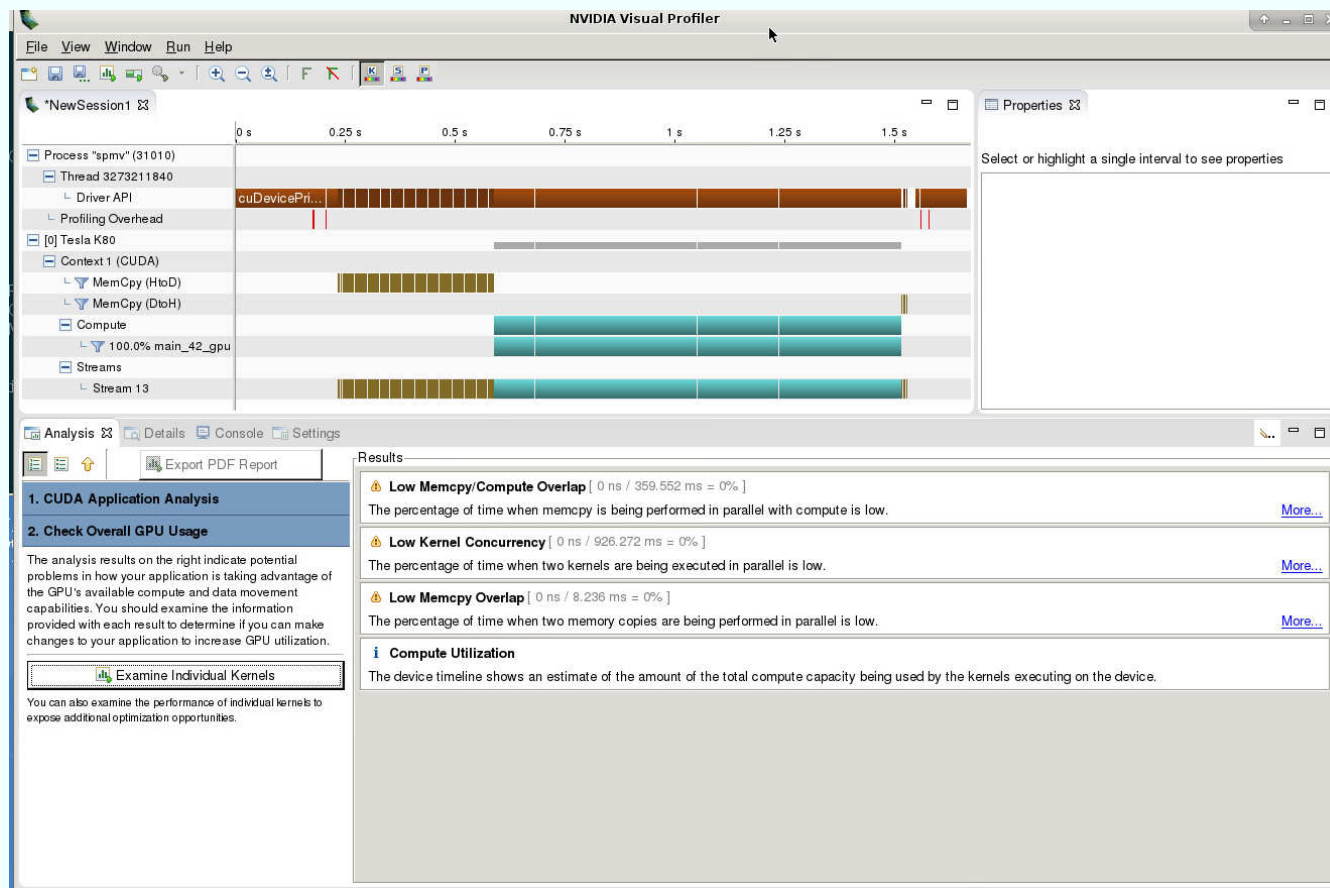
```
./spmv
```

```
Runtime 0.148565 s.
```

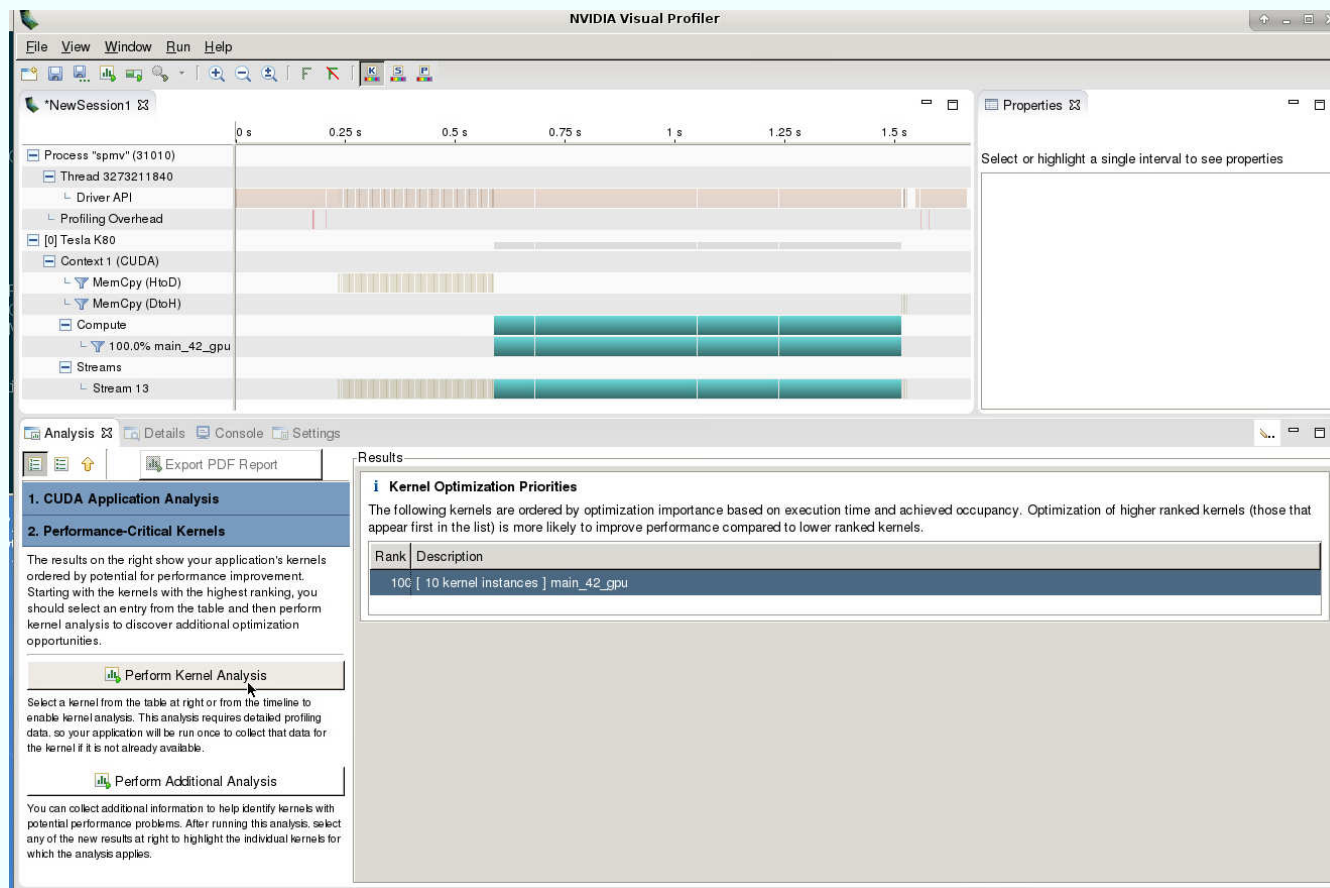
SpMV on K80



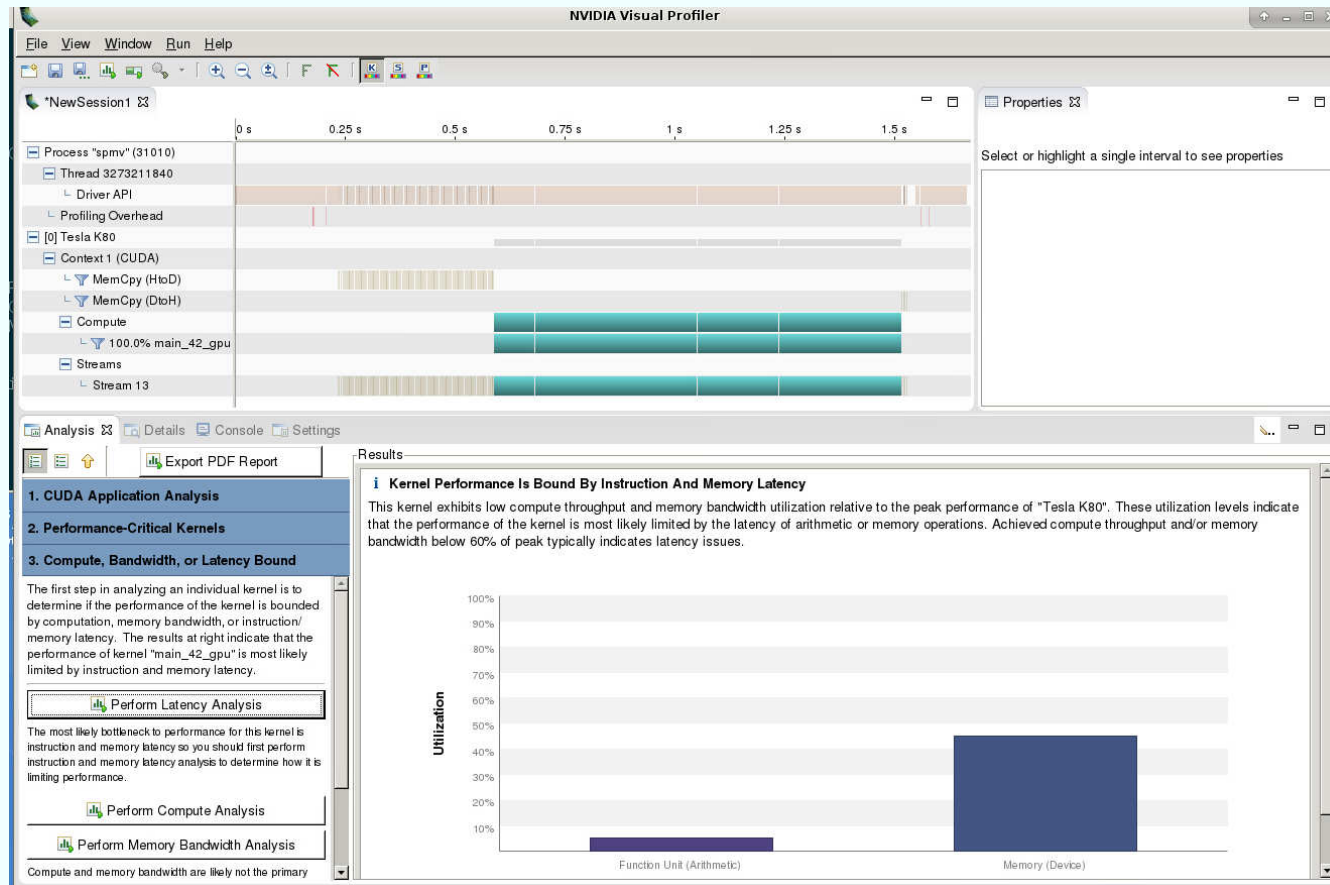
SpMV on K80



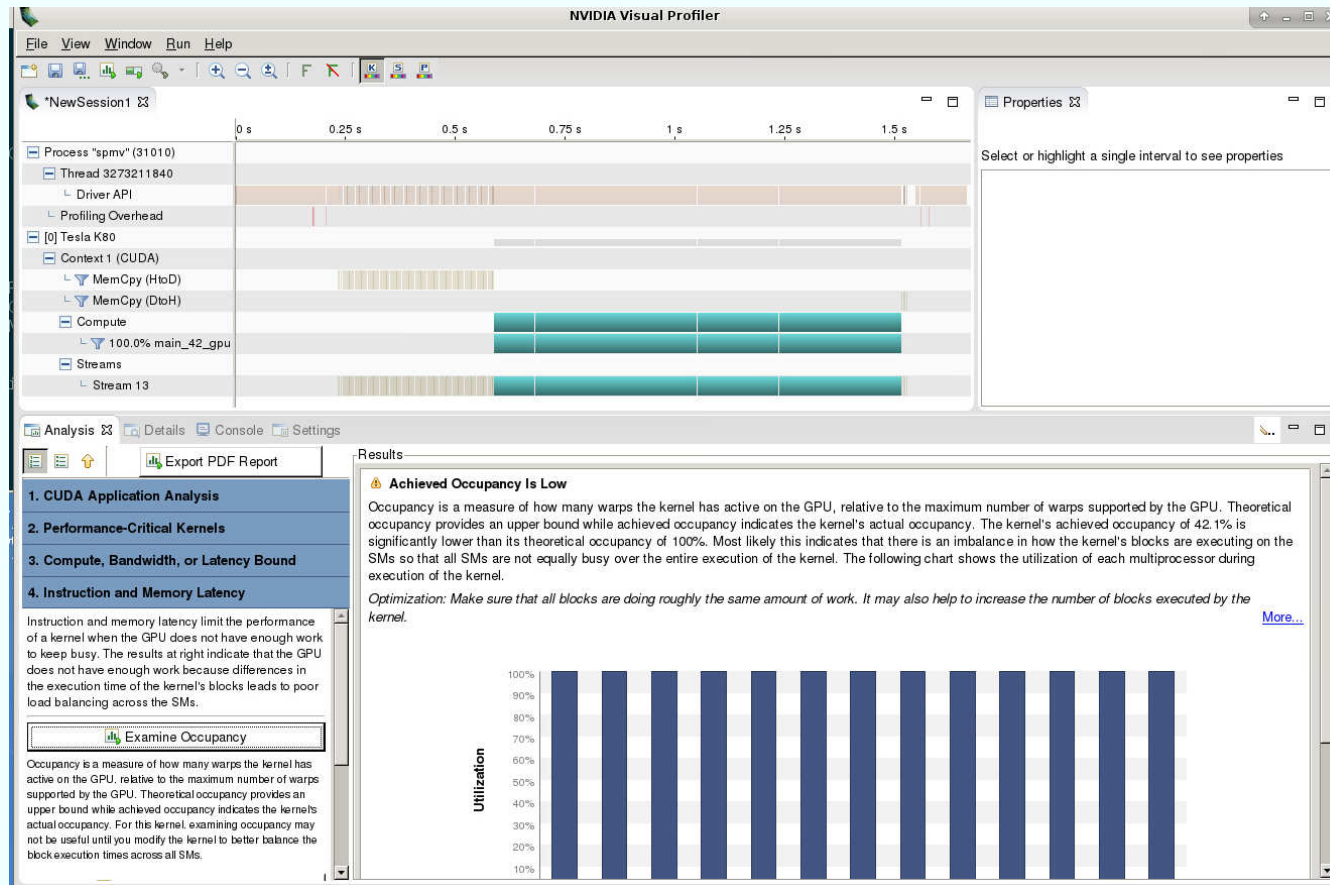
SpMV on K80



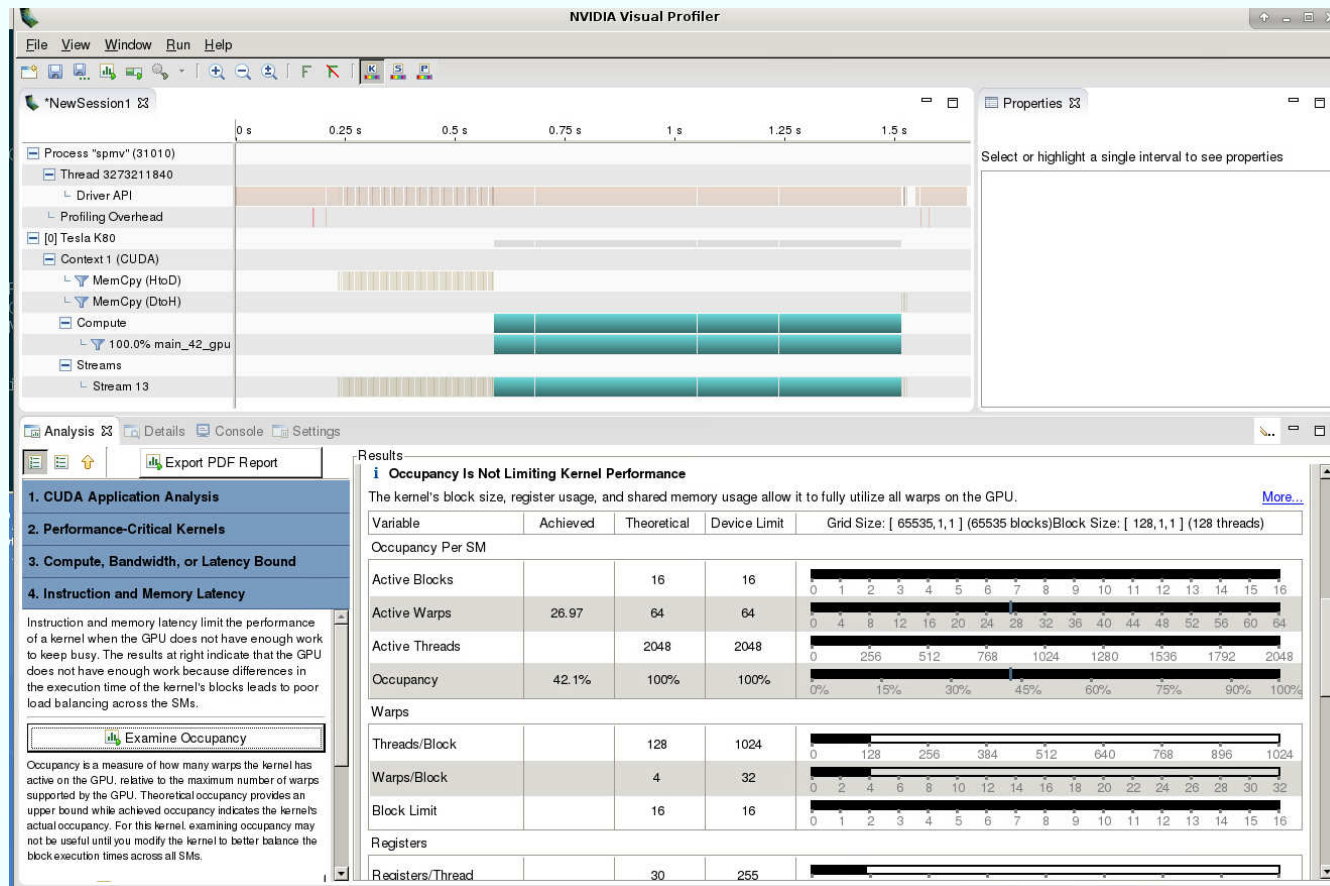
SpMV on K80



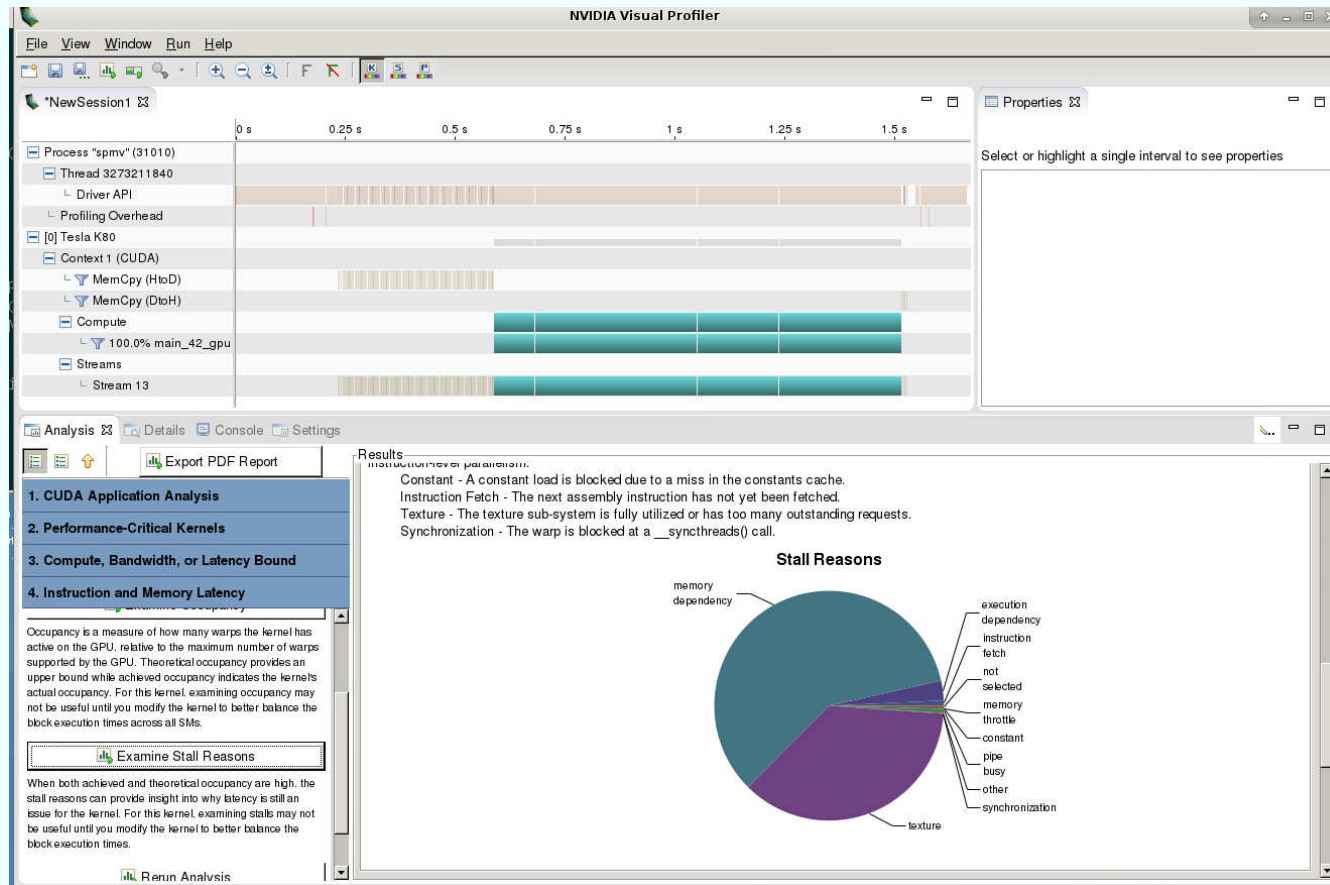
SpMV on K80



SpMV on K80



SpMV on K80



SpMV on K80

Disable usage of
texture cache to see
uncoalesced
memory accesses

```
pgcc -fast -acc -ta=tesla:cc30,lineinfo -Minfo=accel spmv.c -o spmv
```

```
main:
```

```
    36, Generating
```

```
    copyin(row_ptr[:num_rows+1],col_ptr[:num_vals],val[:num_vals],x[:num_rows])
```

```
        Generating copy(y[:num_rows])
```

```
    42, Accelerator kernel generated
```

```
        Generating Tesla code
```

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

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

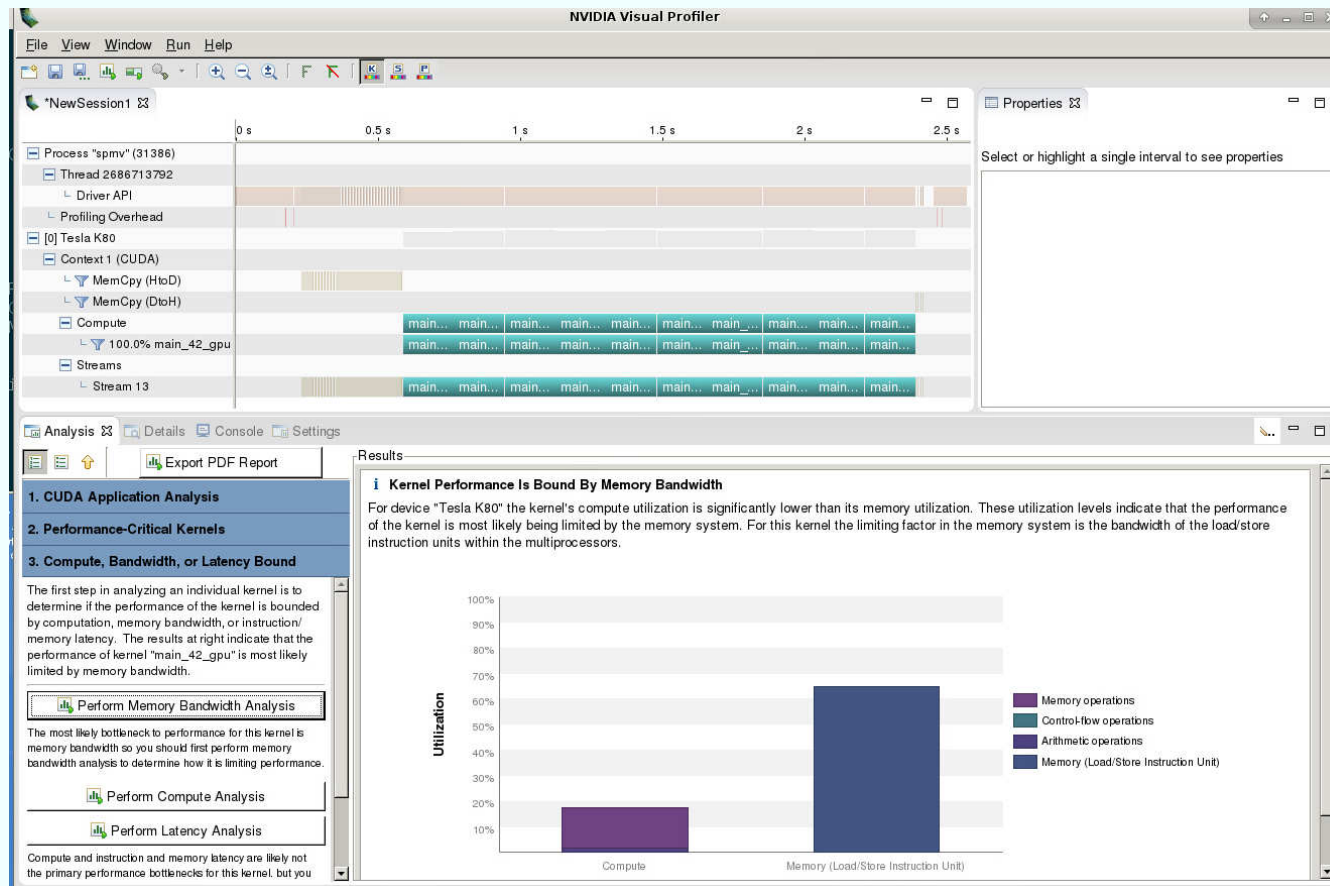
```
        Loop carried reuse of y-> prevents parallelization
```

```
./spmv
```

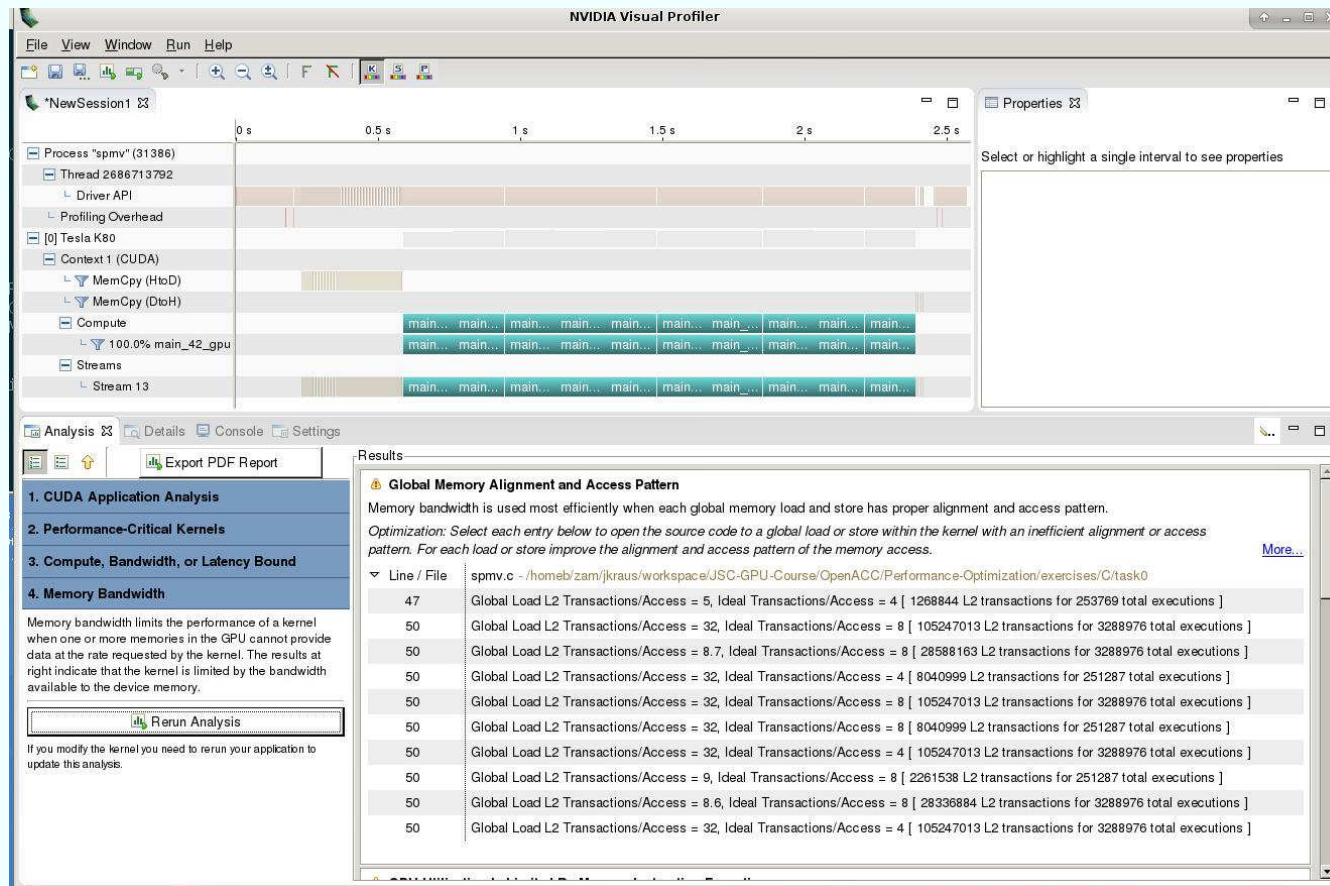
```
Runtime 0.177488 s.
```

Better Profiling
Information

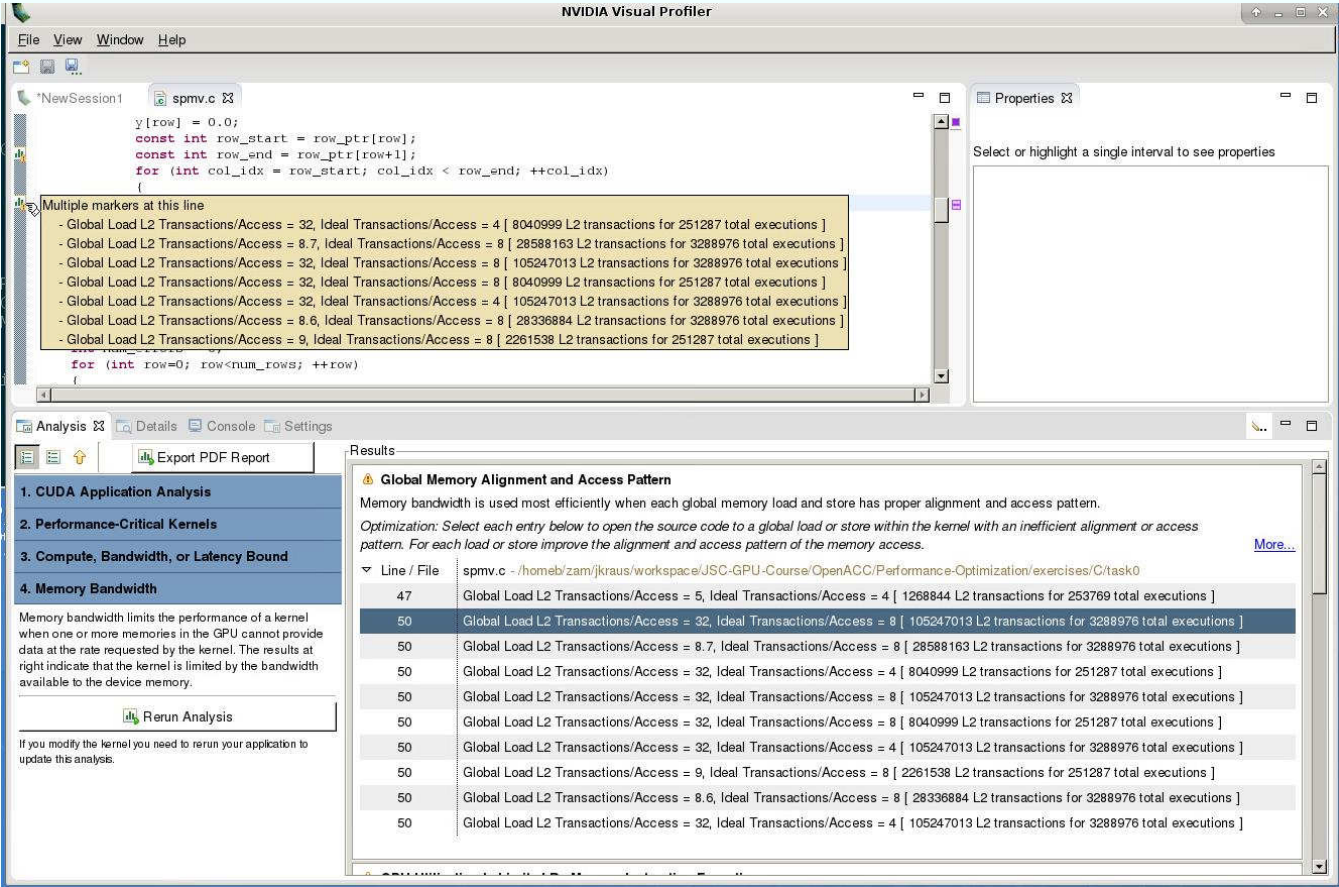
SpMV on K80



SpMV on K80



SpMV on K80



NVIDIA Visual Profiler

File View Window Help

*NewSession1 spmv.c

```

y[row] = 0.0;
const int row_start = row_ptr[row];
const int row_end = row_ptr[row+1];
for (int col_idx = row_start; col_idx < row_end; ++col_idx)
{
    Multiple markers at this line
    - Global Load L2 Transactions/Access = 32, Ideal Transactions/Access = 4 [ 8040999 L2 transactions for 251287 total executions ]
    - Global Load L2 Transactions/Access = 8.7, Ideal Transactions/Access = 8 [ 28588163 L2 transactions for 3288976 total executions ]
    - Global Load L2 Transactions/Access = 32, Ideal Transactions/Access = 8 [ 105247013 L2 transactions for 3288976 total executions ]
    - Global Load L2 Transactions/Access = 32, Ideal Transactions/Access = 8 [ 8040999 L2 transactions for 251287 total executions ]
    - Global Load L2 Transactions/Access = 32, Ideal Transactions/Access = 4 [ 105247013 L2 transactions for 3288976 total executions ]
    - Global Load L2 Transactions/Access = 8.6, Ideal Transactions/Access = 8 [ 28336884 L2 transactions for 3288976 total executions ]
    - Global Load L2 Transactions/Access = 9, Ideal Transactions/Access = 8 [ 2261538 L2 transactions for 251287 total executions ]
    for (int row=0; row<num_rows; ++row)
    {

```

Properties

Select or highlight a single interval to see properties

Analysis Details Console Settings

Export PDF Report

1. CUDA Application Analysis

2. Performance-Critical Kernels

3. Compute, Bandwidth, or Latency Bound

4. Memory Bandwidth

Memory bandwidth limits the performance of a kernel when one or more memories in the GPU cannot provide data at the rate requested by the kernel. The results at right indicate that the kernel is limited by the bandwidth available to the device memory.

Rerun Analysis

If you modify the kernel you need to rerun your application to update this analysis.

Results

Global Memory Alignment and Access Pattern

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

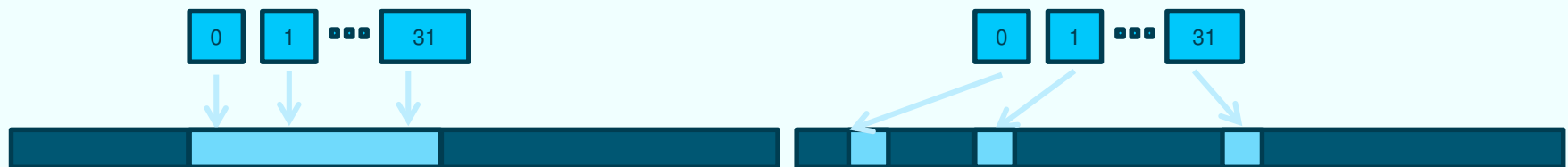
Optimization: Select each entry below to open the source code to a global load or store within the kernel with an inefficient alignment or access pattern. For each load or store improve the alignment and access pattern of the memory access.

Line / File

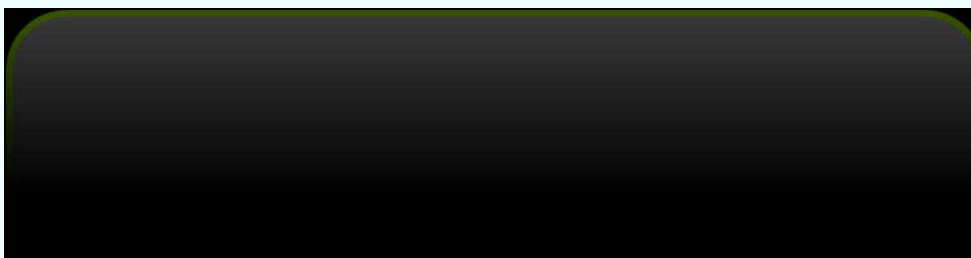
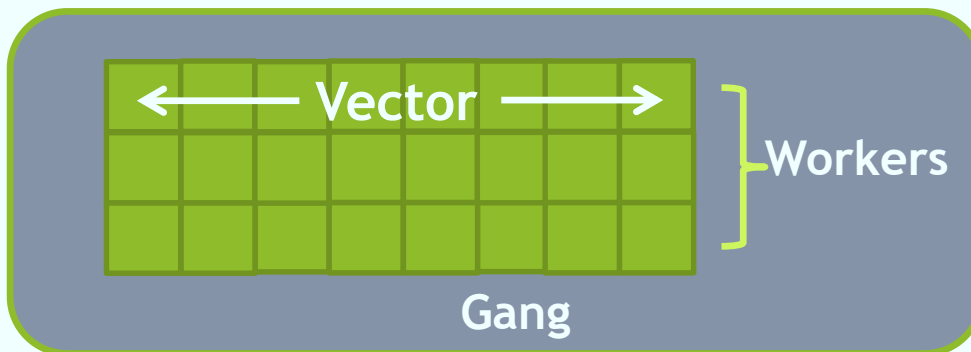
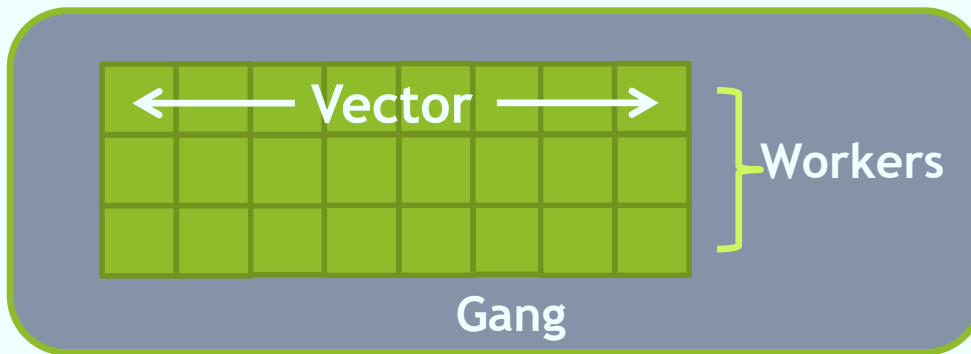
Line	File	Global Load L2 Transactions/Access	Ideal Transactions/Access	Total Executions
47	spm.c - /home/b/zam/jkraus/workspace/JSC-GPU-Course/OpenACC/Performance-Optimization/exercises/C/task0	5	4	1268844 L2 transactions for 253769 total executions
50		32	8	105247013 L2 transactions for 3288976 total executions
50		8.7	8	28588163 L2 transactions for 3288976 total executions
50		32	4	8040999 L2 transactions for 251287 total executions
50		32	8	105247013 L2 transactions for 3288976 total executions
50		32	8	8040999 L2 transactions for 251287 total executions
50		32	4	105247013 L2 transactions for 3288976 total executions
50		9	8	2261538 L2 transactions for 251287 total executions
50		8.6	8	28336884 L2 transactions for 3288976 total executions
50		32	4	105247013 L2 transactions for 3288976 total executions

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



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

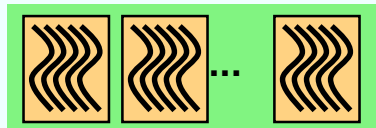
Software



Thread



Thread Block

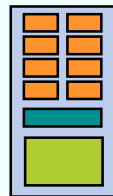


Grid

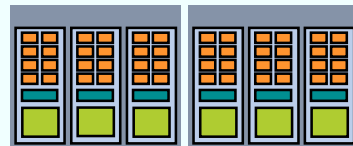
Hardware



Scalar Processor



Multiprocessor



Device

Threads are executed by scalar processors

Thread blocks are executed on multiprocessors

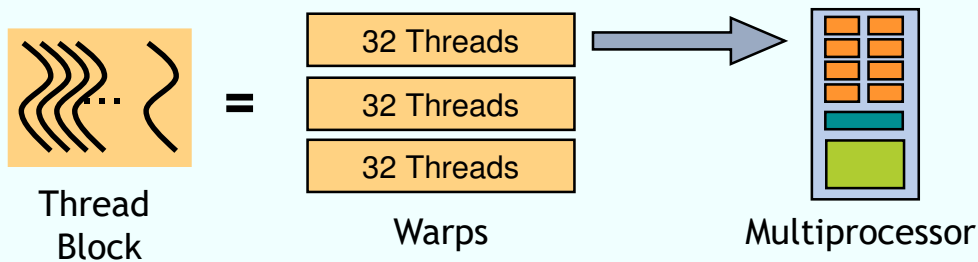
Thread blocks do not migrate

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)

A kernel is launched as a grid of thread blocks

Blocks and grids can be multi dimensional (x,y,z)

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

```
42, Accelerator kernel generated  
    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:     }
```

Want this loop to parallelize
with vector parallelism

48, Complex loop carried dependence of y-> prevents parallelization
Loop carried reuse of y-> prevents parallelization

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 K80

```
pgcc -fast -acc -ta=tesla -Minfo=accel spmv.c -o spmv
```

```
main:
```

```
    36, Generating
```

```
copyin(row_ptr[:num_rows+1],col_ptr[:num_vals],val[:num_vals],x[:num_rows])
```

```
    Generating copy(y[:num_rows])
```

```
    42, Accelerator kernel generated
```

```
    Generating Tesla code
```

```
    43, #pragma acc loop gang /* blockIdx.x */
```

```
    48, #pragma acc loop vector(128) /* threadIdx.x */
```

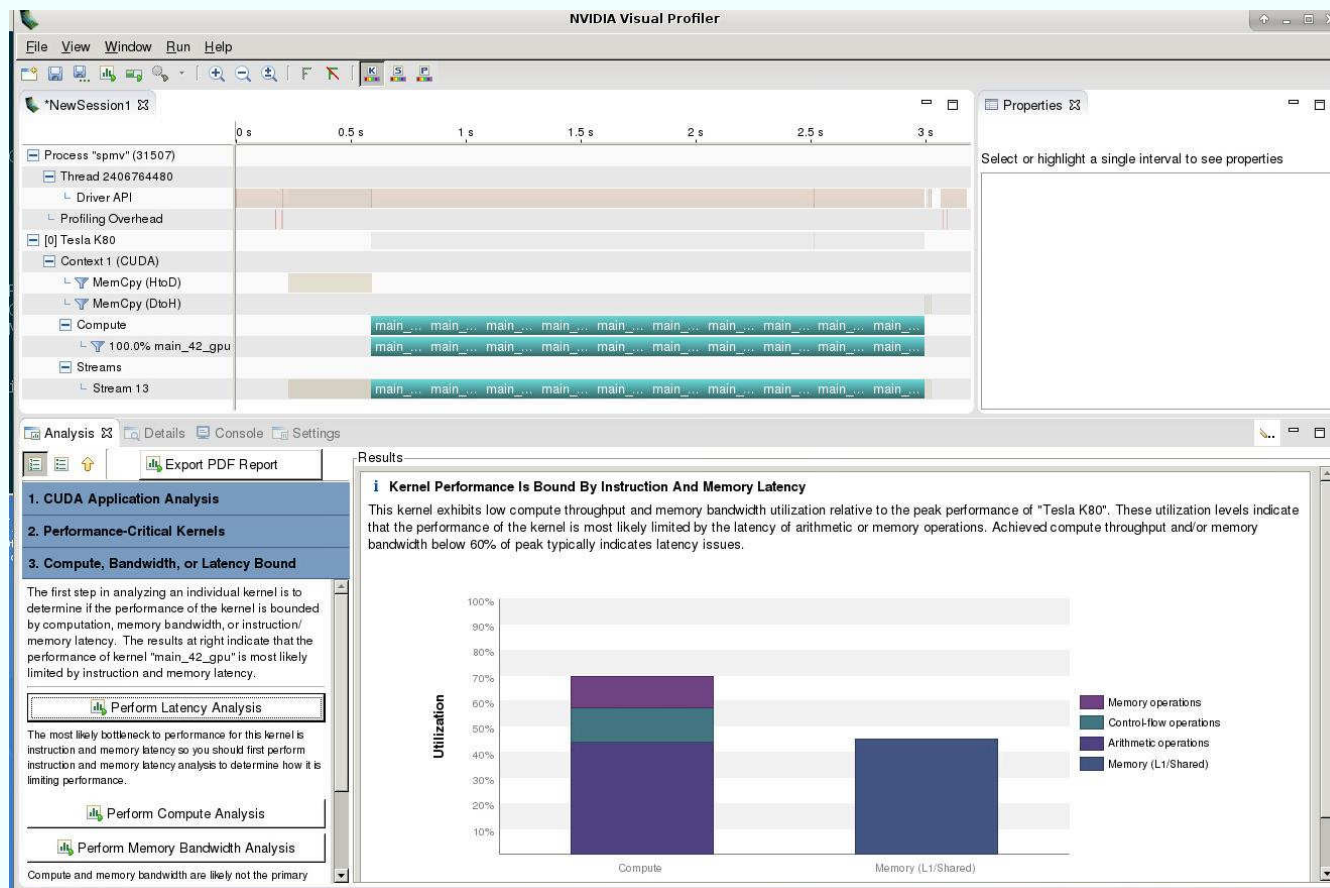
```
    50, Sum reduction generated for y_tmp
```

```
    48, Loop is parallelizable
```

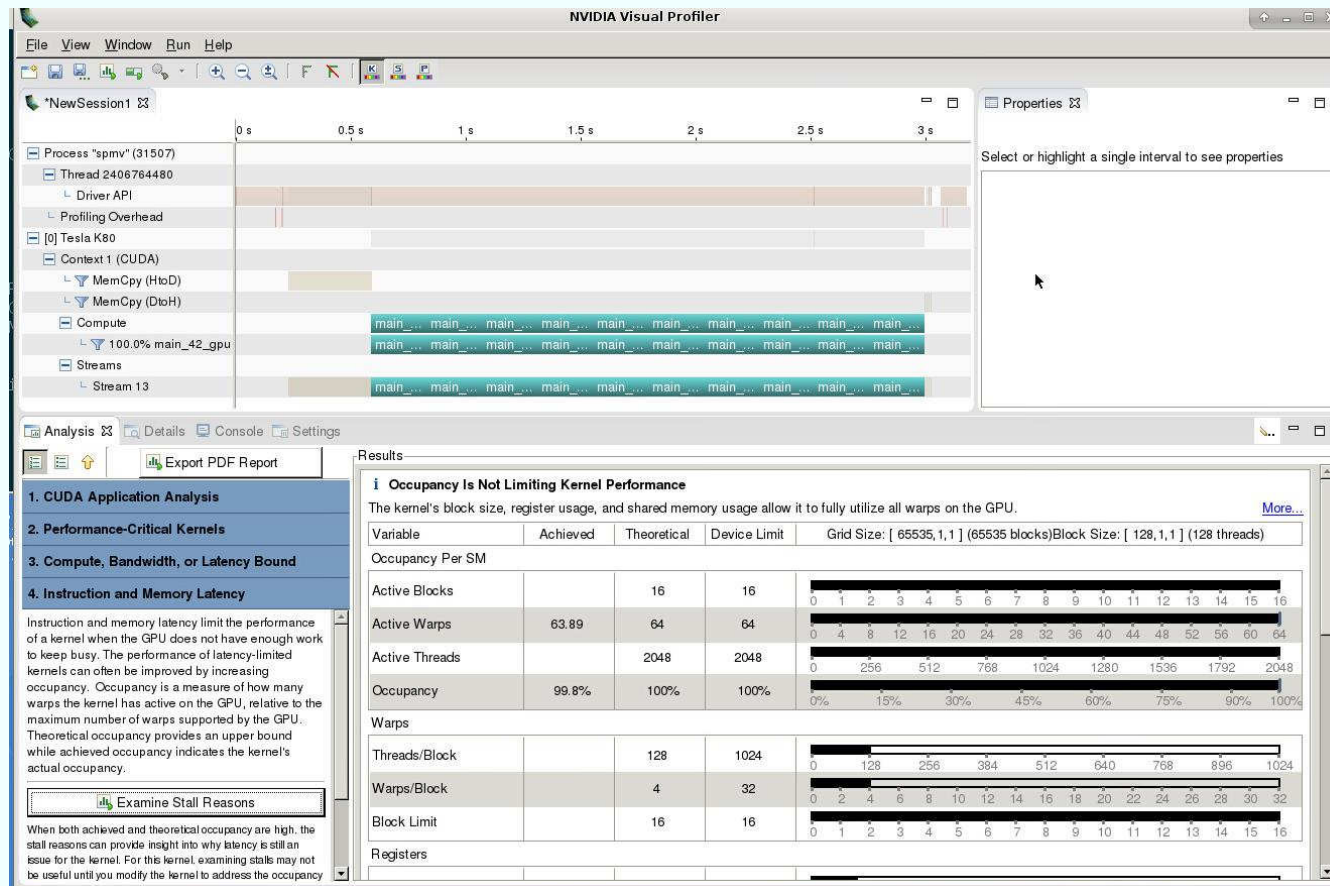
```
./spmv
```

```
Runtime 0.166006 s.
```

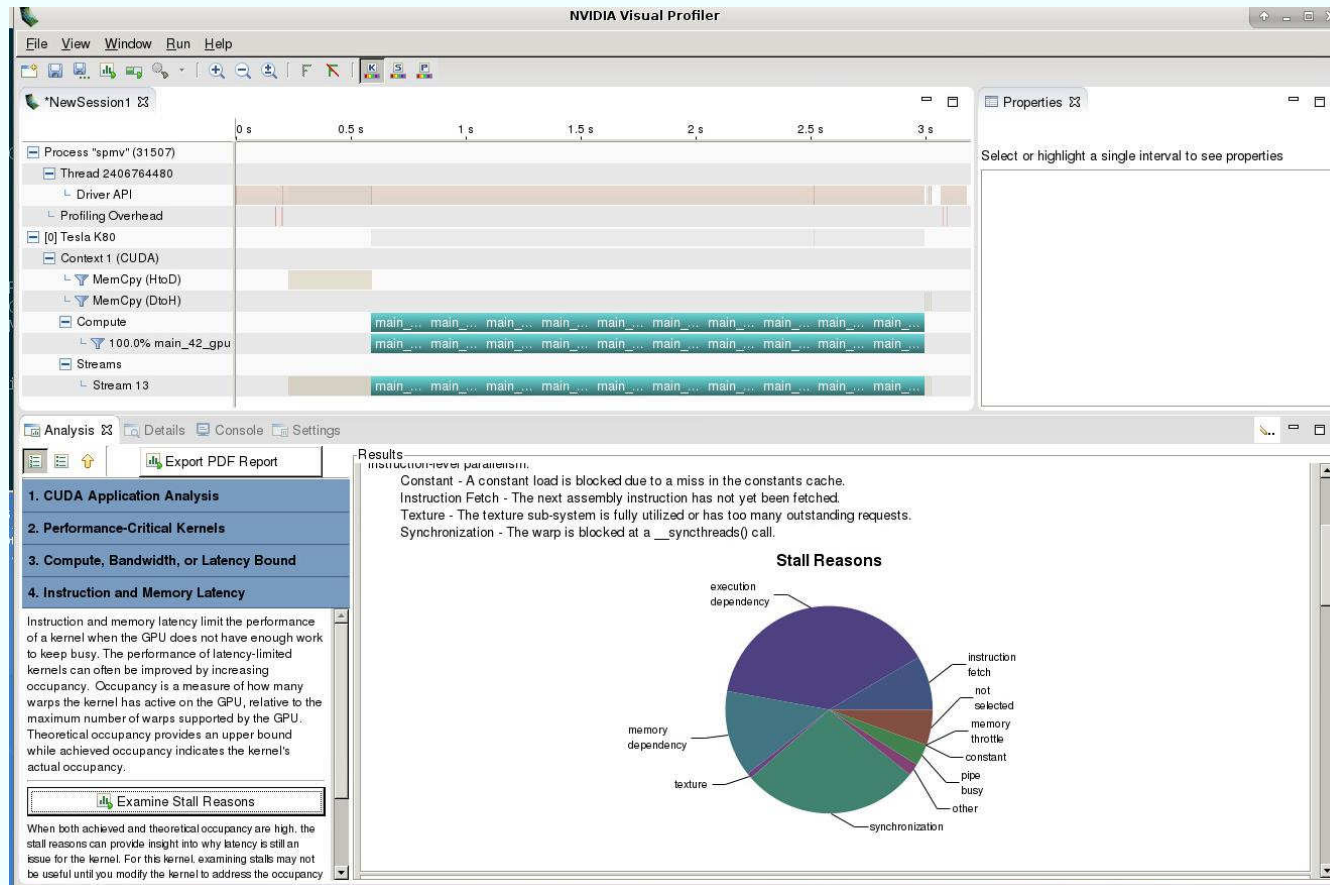
SpMV on K80



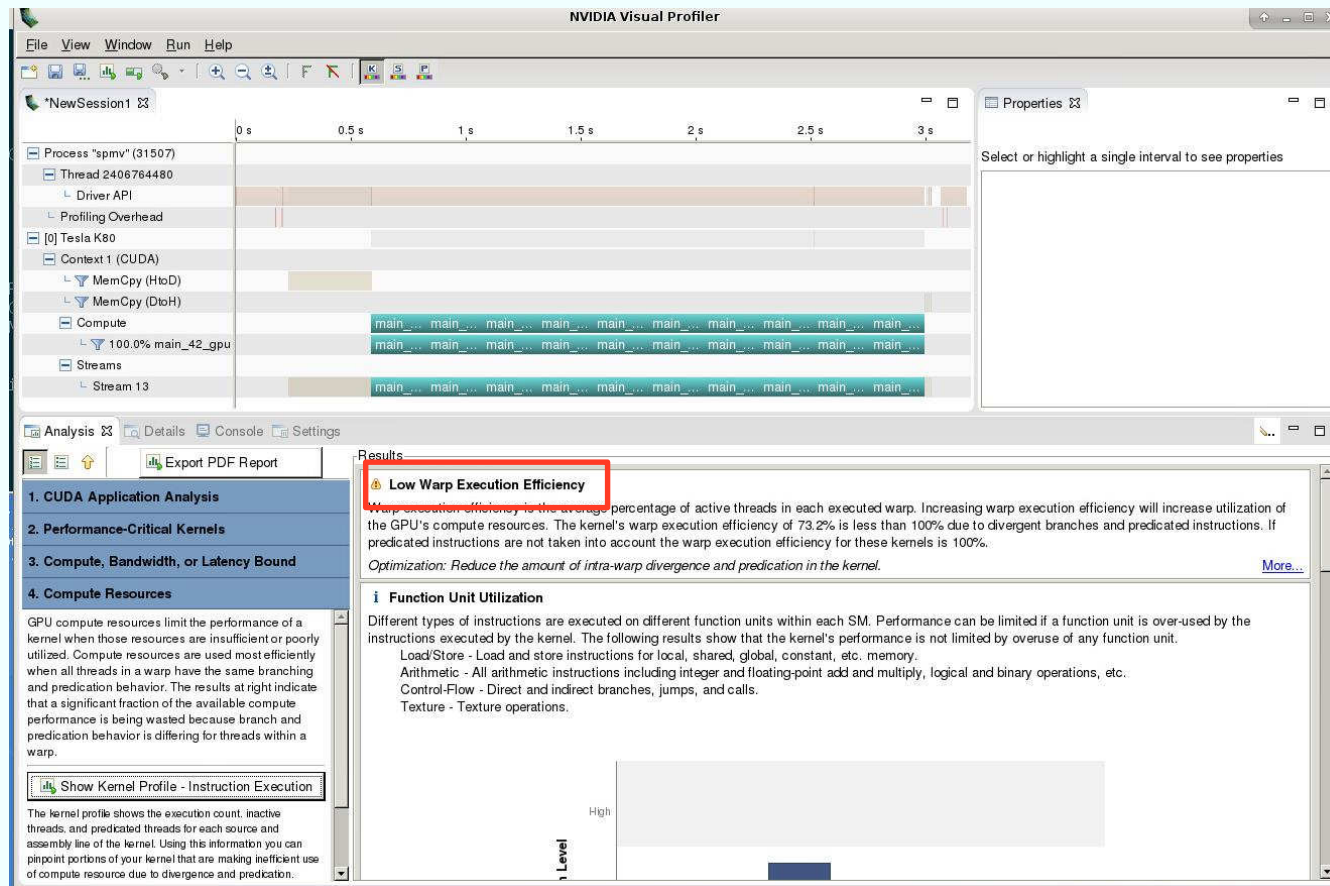
SpMV on K80



SpMV on K80



SpMV on K80



SpMV on K80

```

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]];
  
```

gang

vector(128)

```

42, Accelerator kernel generated
    Generating Tesla code
43, #pragma acc loop gang /* blockIdx.x */
48, #pragma acc loop vector(128) /* threadIdx.x */
50, Sum reduction generated for 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 K80

```

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]];
  
```

gang

vector(32)

```

42, Accelerator kernel generated
    Generating Tesla code
43, #pragma acc loop gang /* blockIdx.x */
48, #pragma acc loop vector(32) /* threadIdx.x */
50, Sum reduction generated for y_tmp
  
```

SpMV on K80

```
pgcc -fast -acc -ta=tesla -Minfo=accel spmv.c -o spmv
```

```
main:
```

```
    36, Generating
```

```
copyin(row_ptr[:num_rows+1],col_ptr[:num_vals],val[:num_vals],x[:num_rows])
```

```
    Generating copy(y[:num_rows])
```

```
    42, Accelerator kernel generated
```

```
    Generating Tesla code
```

```
    43, #pragma acc loop gang /* blockIdx.x */
```

```
    48, #pragma acc loop vector(32) /* threadIdx.x */
```

```
    50, Sum reduction generated for y_tmp
```

```
    48, Loop is parallelizable
```

```
./spmv
```

```
Runtime 0.119796 s. (was 0.166006 s)
```

Keeping the code performance portable

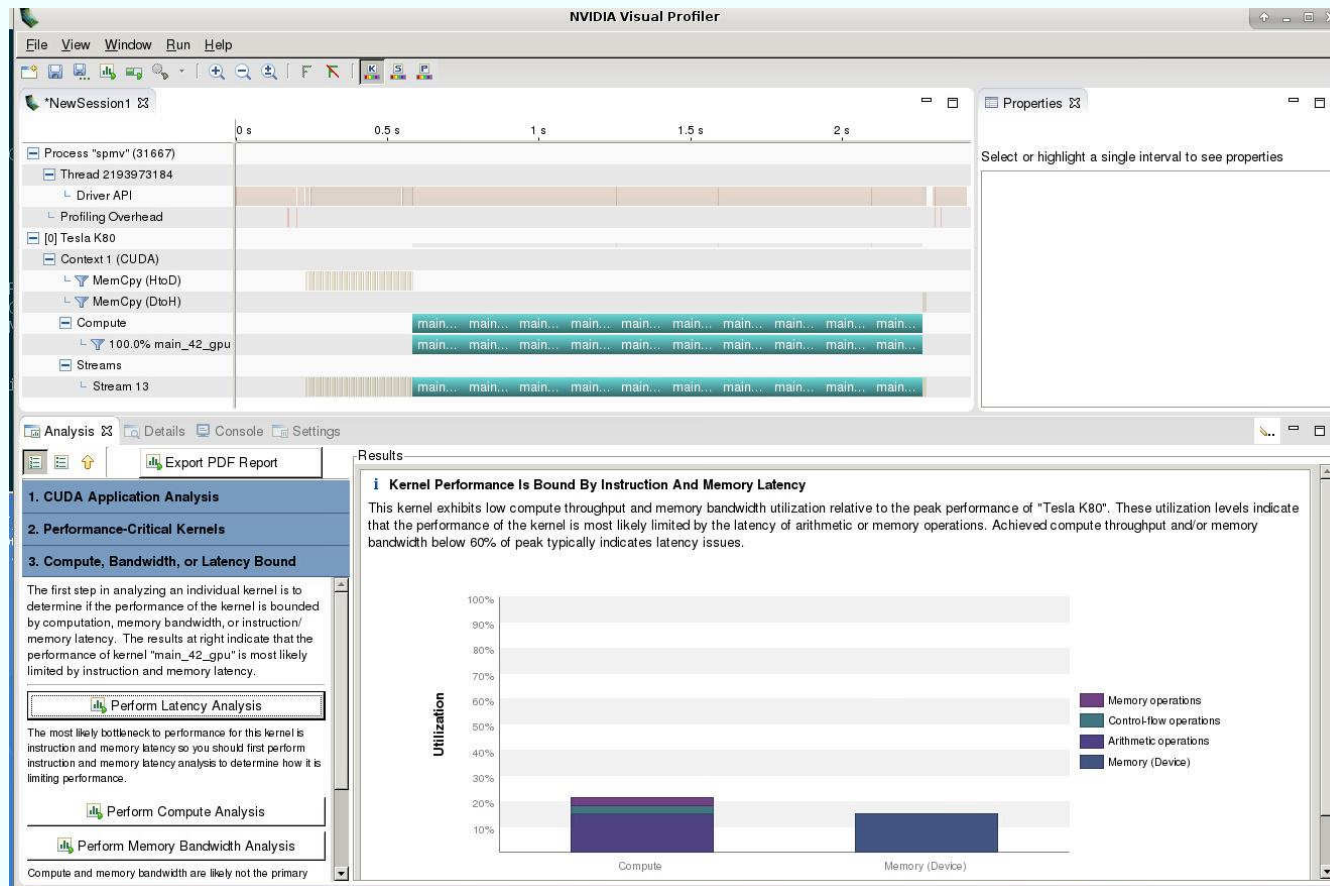
- The `device_type` clause allows device specific tuning without harming performance portability
- All clauses following a `device_type` clause only apply for the given target:

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

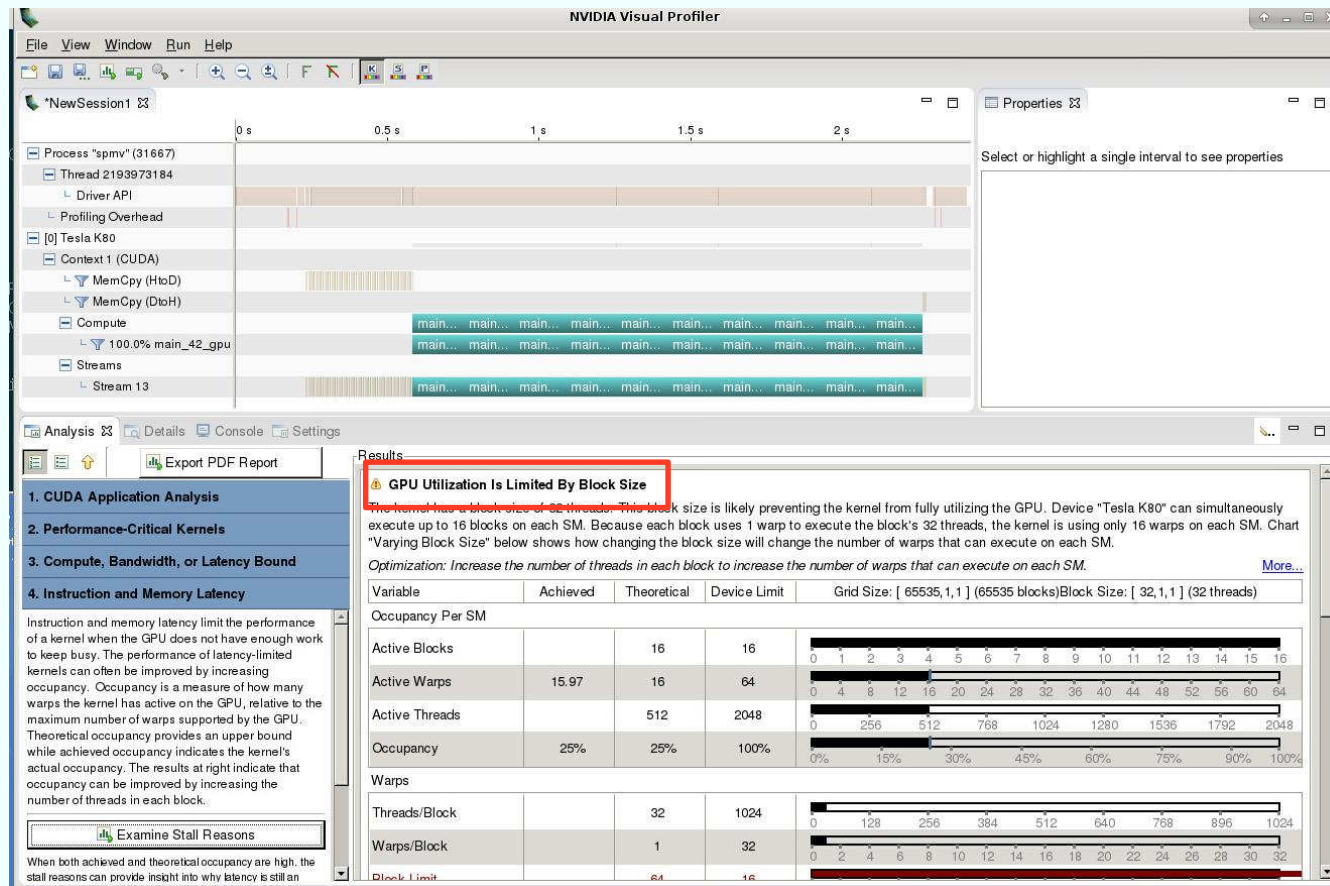
Tasks

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

SpMV on K80



SpMV on K80



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:   {

```

gang, worker(4)

vector(32)

42, Accelerator kernel generated

Generating Tesla code

43, #pragma acc loop gang, worker(4) /* blockIdx.x threadIdx.y */

48, #pragma acc loop vector(32) /* threadIdx.x */

50, Sum reduction generated for y_tmp

Understanding Compiler Output (recap)

```
42, Accelerator kernel generated
    Generating Tesla code
43, #pragma acc loop gang, worker(4) /* blockIdx.x threadIdx.y */
48, #pragma acc loop vector(32) /* threadIdx.x */
50, Sum reduction generated for y_tmp
```

- 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 K80

```
pgcc -fast -acc -ta=tesla -Minfo=accel spmv.c -o spmv
```

```
main:
```

```
    36, Generating
```

```
copyin(row_ptr[:num_rows+1],col_ptr[:num_vals],val[:num_vals],x[:num_rows])
```

```
    Generating copy(y[:num_rows])
```

```
    42, Accelerator kernel generated
```

```
    Generating Tesla code
```

```
    43, #pragma acc loop gang, worker(4) /* blockIdx.x threadIdx.y */
```

```
    48, #pragma acc loop vector(32) /* threadIdx.x */
```

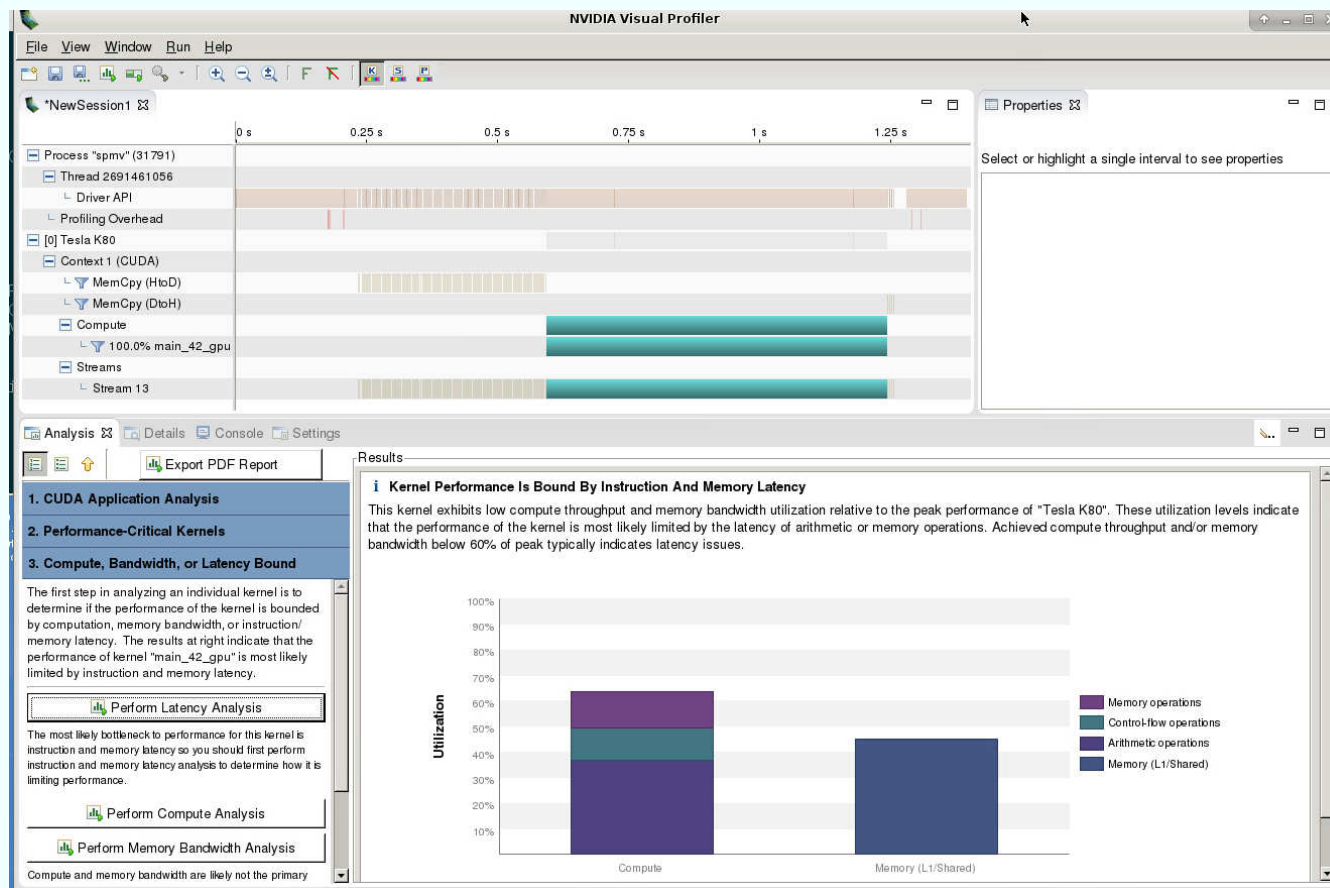
```
    50, Sum reduction generated for y_tmp
```

```
    48, Loop is parallelizable
```

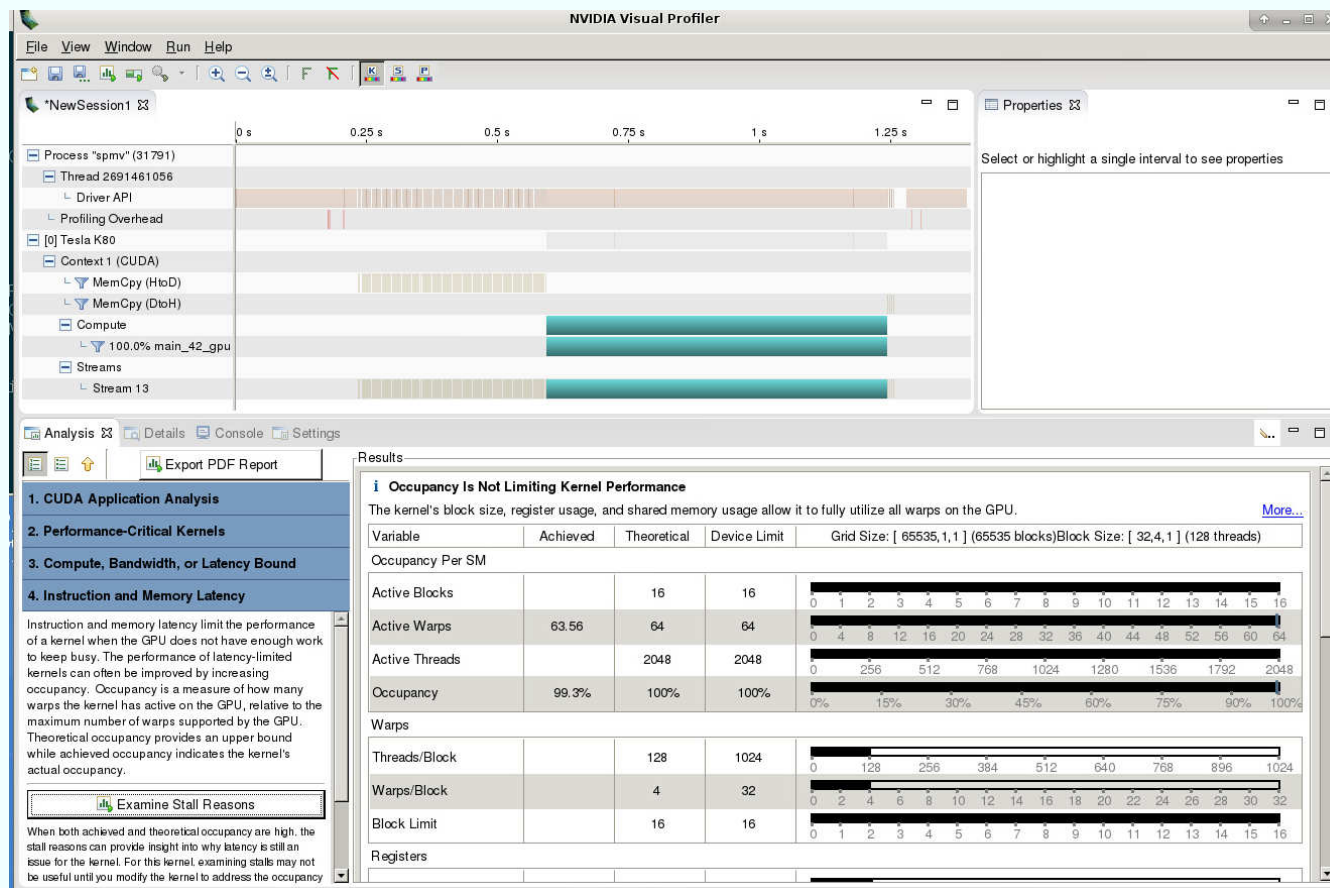
```
./spmv
```

```
Runtime 0.047039 s. (was 0.119796 s and 0.166006 s)
```

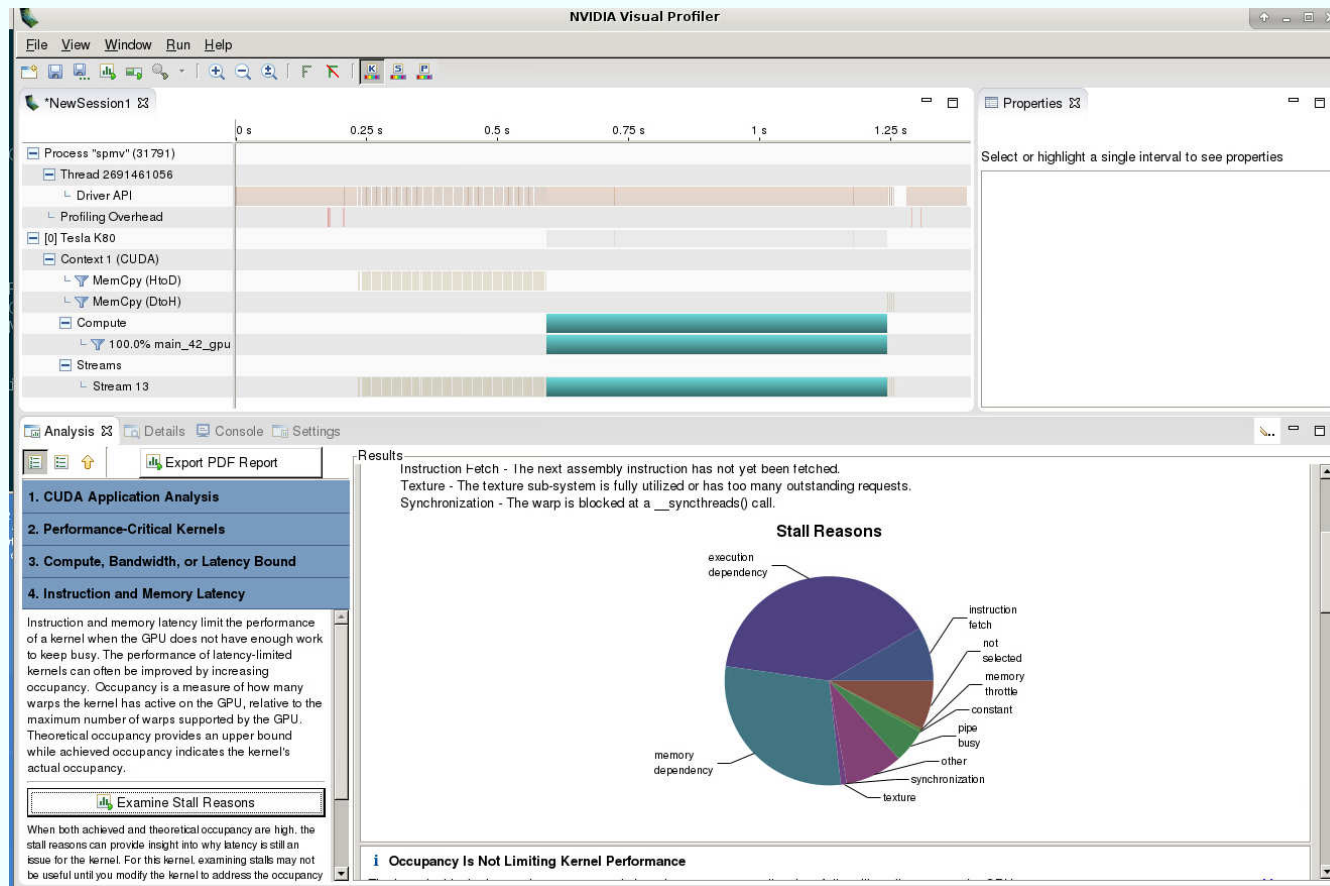
SpMV on K80



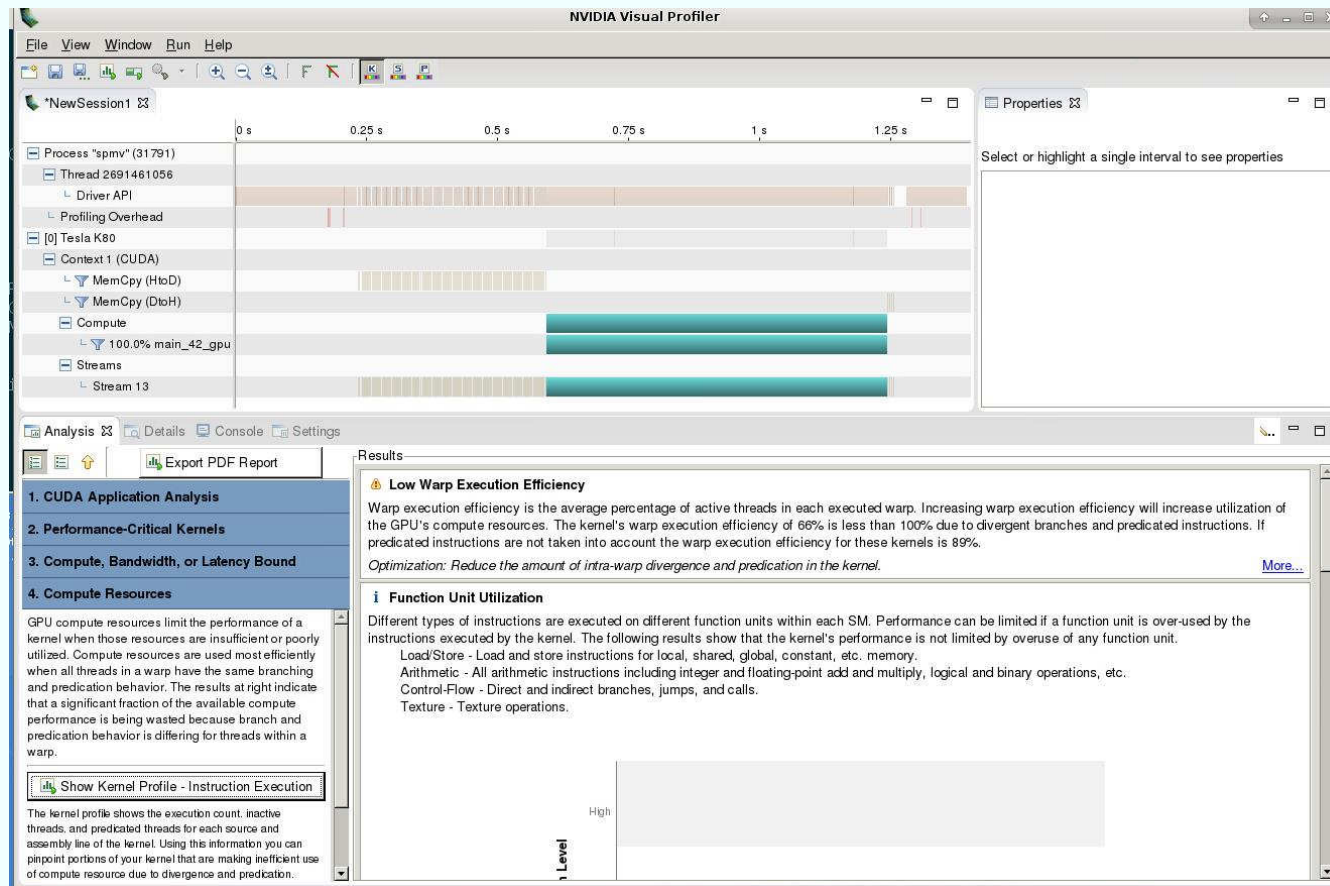
SpMV on K80



SpMV on K80



SpMV on K80



Conclusions

- The NVIDIA Visual Profiler can be used to identify performance bottlenecks in 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.