



OpenACC Performance Optimization Workflow

Markus Hrywniak, DevTech Compute | JSC OpenACC course, October 2022



Before We Start

Contents and expectations

- Workflow focus
 - Data-driven via tools
 - Memory coalescing
 - Loop optimizations
- GPUs are *a lot* faster than even 2, 3 years ago
 - Not just FLOPs, also larger caches, ...
- Programmability even more improved
- Unfortunate (?) example: previous SPMV-based exercise
 - "naive" version fastest on A100, faster than previous V100-optimized solution
 - ...actually a good thing for users!
- Goal: Understand how tools (compiler output/profiler) can help
- Performance optimization is very seldomly a straightforward process
 - Tinker and experiment – this is what makes it fun!

General notes

- Important flags for NVHPC Compiler
- Building with lightweight debug information:
 - -gpu=lineinfo -gopt
- 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 NV_ACC_TIME=1 for lightweight profiler on time of data movements and kernels
 - NV_ACC_NOTIFY=1 gives a detailed breakdown of kernel launches and data transfers (bit field)
- More details at <https://docs.nvidia.com/hpc-sdk/compiler/openacc-gs/index.html#env-vars>

NVHPC Runtime Measurements

```
$NV_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 on the Command Line

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

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

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):

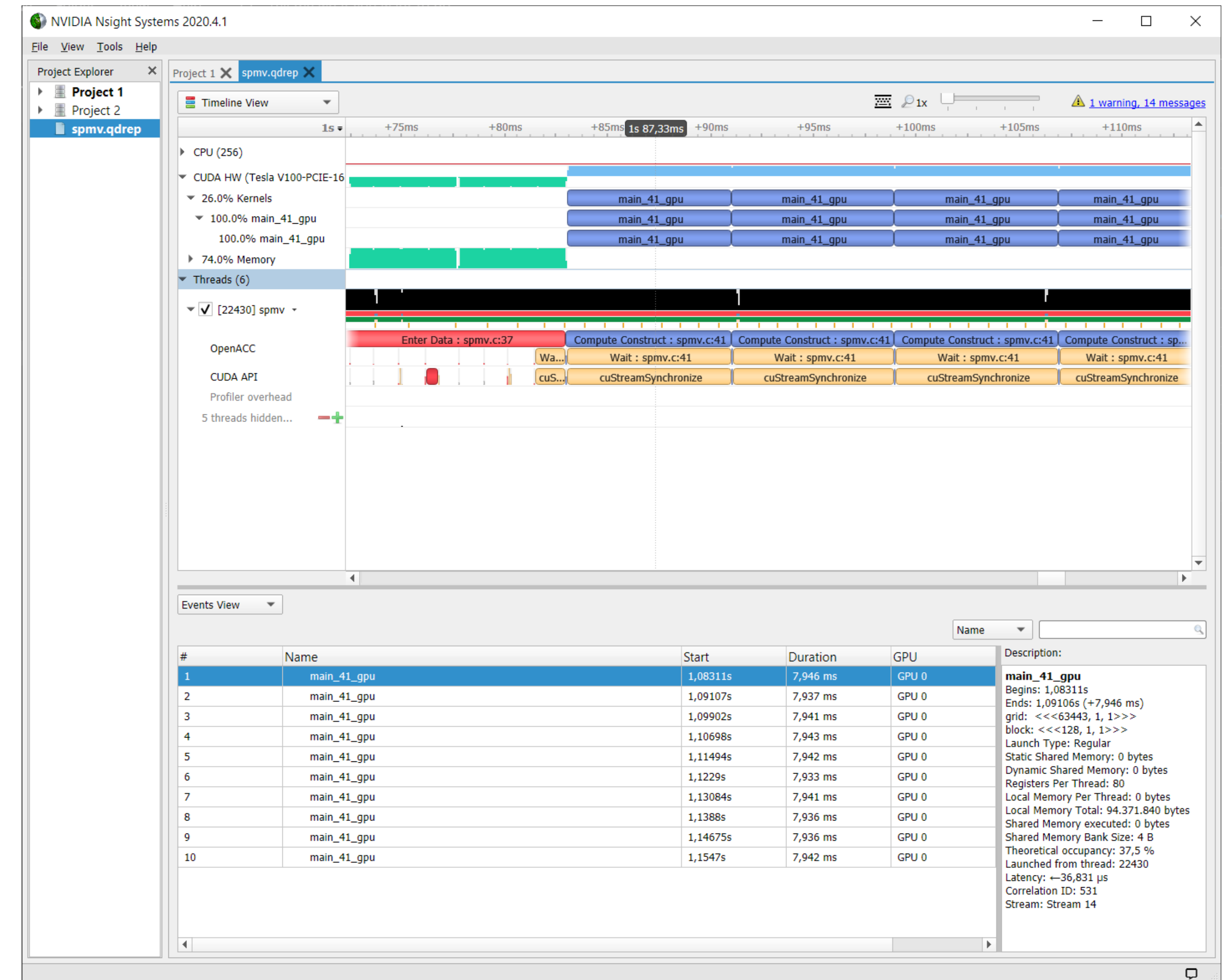
Time(%)	Total Time (ns)	Operations	Average	Minimum	Maximum	Operation
97.8	220817077	168	1314387.4	138847	1522486	[CUDA memcpy HtoD]
2.2	4926174	4	1231543.5	1111096	1271928	[CUDA memcpy DtoH]

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
 - High-level, whole-program visualization for 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

NVIDIA Nsight Systems 2020.4.1

File View Tools Help

Project Explorer

- Project 1
- Project 2
- spmvs.qdrep

Project 1 X spmv.qdrep X

Timeline View

0s 0,1s 0,2s 0,3s 0,4s 0,5s 0,6s 0,7s 0,8s 0,9s 1s 1,1s 1,2s

CPU (256)

CUDA HW (Tesla V100-PCIE-1)

- 26.0% Kernels
- 100.0% main_41_gpu
- 74.0% Memory

Threads (6)

- [22430] spmv
- OpenACC
- CUDA API
- Profiler overhead
- 5 threads hidden...

Enter Data : spmv.c:37

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

Name

Description:

- Compute Construct : spmv.c:41
- Begins: 1,08306s
- Ends: 1,09106s (+7,998 ms)
- Construct Kind: Parallel Construct
- Async: -1
- Async Map: 32
- Source File: spmv.c
- Func Name: main

Call stack:

- libcuda.so.450.51.06[9 Frames]
- libcuda.so.450.51.06!cuStreamSynchronize
- libaccdevice.so!__pgi_uacc_cuda_wait

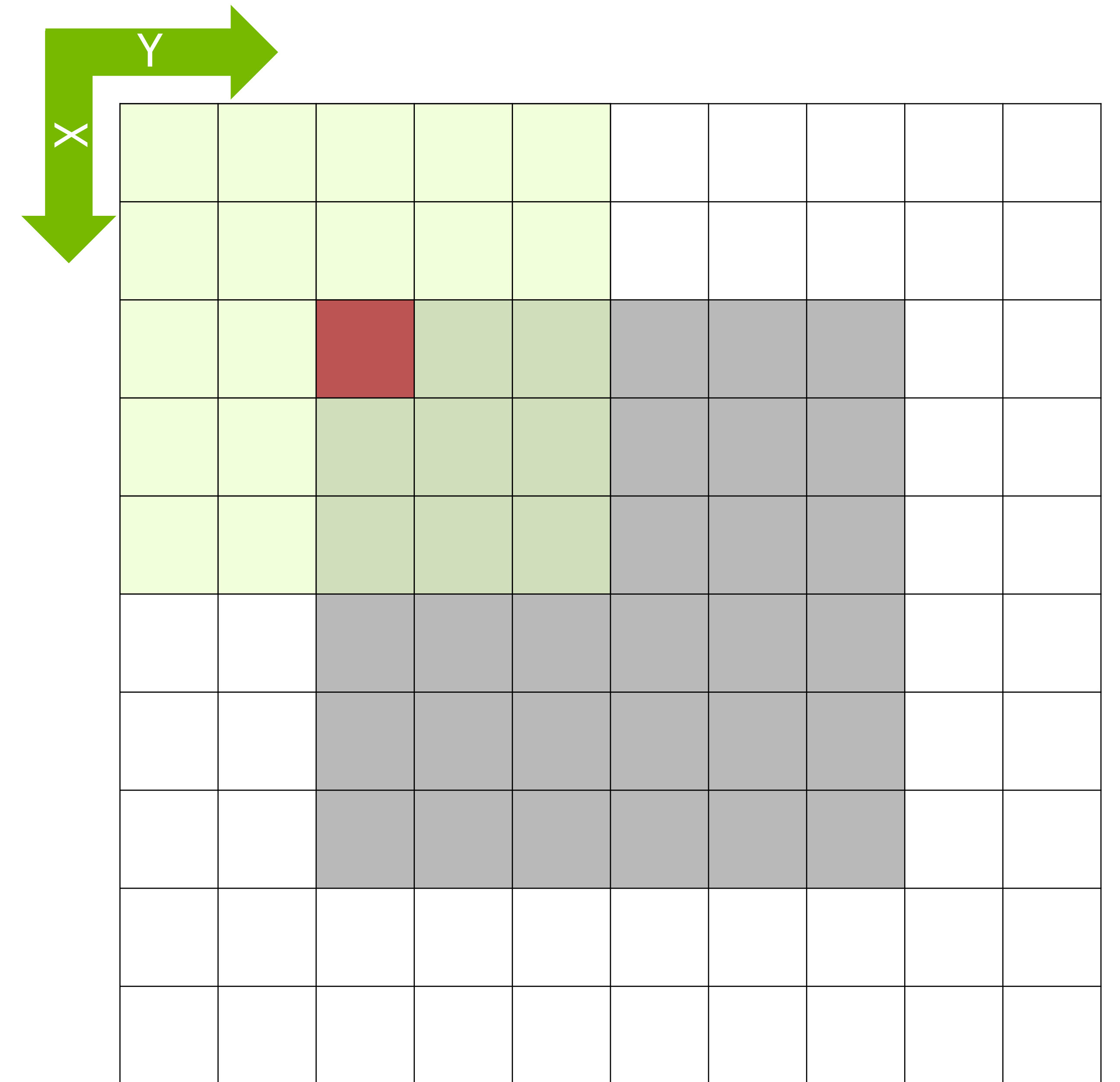
Implementing a Convolution

- Weighted summation over local neighborhood ("stencil")

- Input A, output B, stencil coefficients ω

$$B(x, y) = \sum_{sx} \sum_{sy} \omega(sx, sy) A(x + sx, y + sy)$$

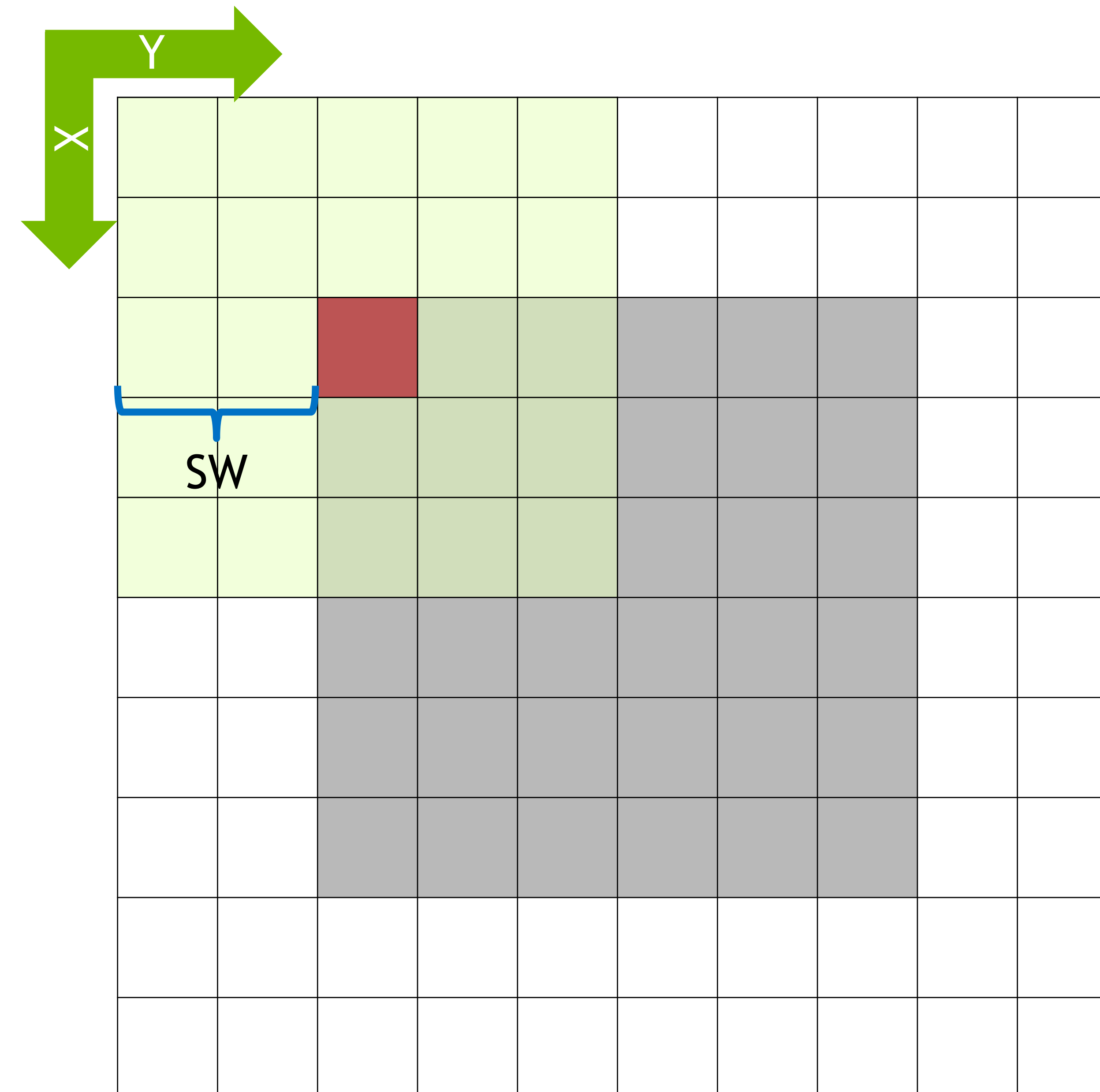
- Jacobi "generalized" – could also be written that way
- "Filter kernel", "stencil", name depends on context
 - size of the stencil etc.
- Image filter applications
 - border detection, gaussian softening
- Halo area at borders



Implementing a Convolution

- Stencil width: Go SW pixels into all directions
 - sum up contributions
 - repeat for all pixels
- Total points $(2\text{ sw} + 1)^2$

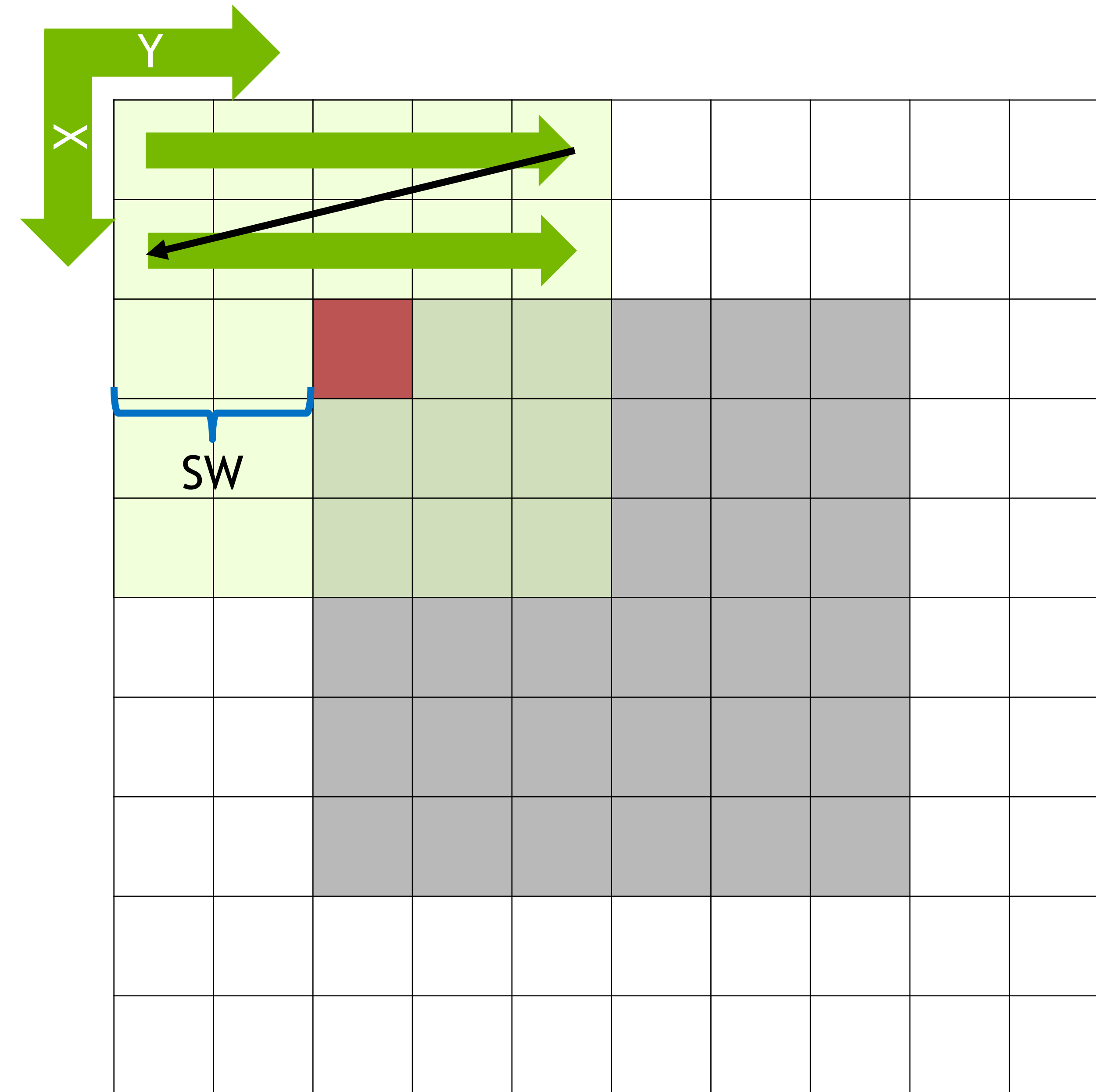
```
for (int x = sw; x < N - sw; ++x) {
    for (int y = sw; y < N - sw; ++y) {
        B[x][y] = 0;
        for (int sx = -sw; sx <= sw; ++sx) {
            for (int sy = -sw; sy <= sw; ++sy) {
                const float val =
                    stencil[sw + sx][sw + sy]
                      * A[x + sx][y + sy];
                B[x][y] += val;
            }
        }
    }
}
```



Implementing a Convolution

- Configurable stencil width
- Explicit looping, no unrolling, for simplicity

```
for (int x = sw; x < N - sw; ++x) {
  for (int y = sw; y < N - sw; ++y) {
    B[x][y] = 0;
    for (int sx = -sw; sx <= sw; ++sx) {
      for (int sy = -sw; sy <= sw; ++sy) {
        const float val =
          stencil[sw + sx][sw + sy]
            * A[x + sx][y + sy];
        B[x][y] += val;
      }
    }
  }
}
```



Parallelizing on CPU

Via `-acc=multicore`

- Always: Checking correctness
 - Easy to go fast by computing nonsense
- Recall: NV_ACC_TIME for simple measurements
- Code example uses manual timing: Repetitions for averaging
- Solutions: Look at base version of next task's folder

Task 0

Get familiar with the code and establish a baseline

1. Directory: task0/
2. There are several variants you can build. Try "make all"
 1. Launching any of them will print a help
 2. **NOTE:** We will use SW=3 for all tasks – the Makefile targets do this automatically
3. Generate reference data. Use "make create_ref"
 1. This will run the serial version and output the expected data in a .bin file, as we have not yet verified any of the parallel versions
 2. Note the command line, you can run this yourself with any version

```
$ make create_ref
```

```
srun [...] ./conv_serial 3 yes
```

```
Recreating reference data...
```

```
Using stencil width = 3
```

```
Runtime 1194.230588 ms
```

4. Inspect the Makefile, the source conv.c and look for the TODO
 1. Use the correct #pragma to parallelize the outer loop
 2. Compare runtimes of all versions. You can use "make run_all". Write down the runtimes.

Task 0 - Results

- Reference results: Simplest way is to dump data and compare (or run known-good implementation afterwards)
 - Only serial known-good for task 0 – for other tasks, we can copy/link .bin file or use existing GPU version
- Roughly, you should see
 - Serial, Multicore, GPU
 - 1200 ms vs 165 ms vs 56 ms
- Now, to make it faster, look for first clues
- Compiler output
- Profilers: Nsight Systems should **usually** be your first step!

Task 0 – Compiler output

Via `-Minfo=accel`

run_convolution_kernel_and_time:

49, Generating copyout(B[:4096][:]) [if not already present]

Generating copyin(stencil[:stencil_dim][:stencil_dim],A[:4096][:]) [if not already present]

51, Generating NVIDIA GPU code

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

57, #pragma acc loop seq

59, #pragma acc loop seq

60, #pragma acc loop seq

57, Complex loop carried dependence of B->,A->,stencil prevents parallelization

59, Complex loop carried dependence of stencil prevents parallelization

Loop carried dependence of B-> prevents parallelization

Loop carried backward dependence of B-> prevents vectorization

Complex loop carried dependence of B->,A-> prevents parallelization

60, Complex loop carried dependence of stencil,B->,A-> prevents parallelization

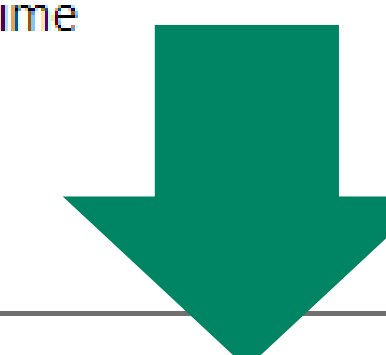
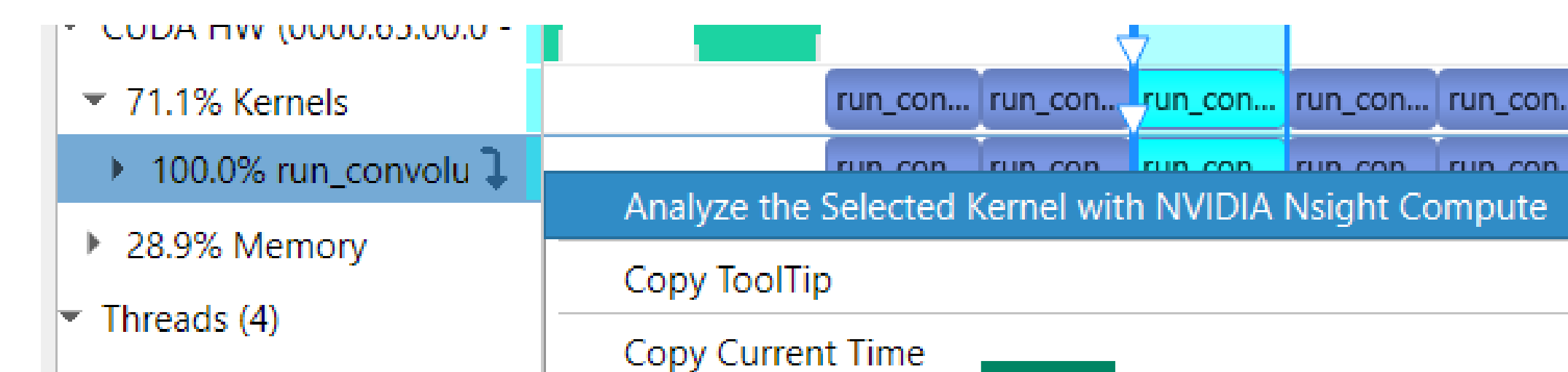
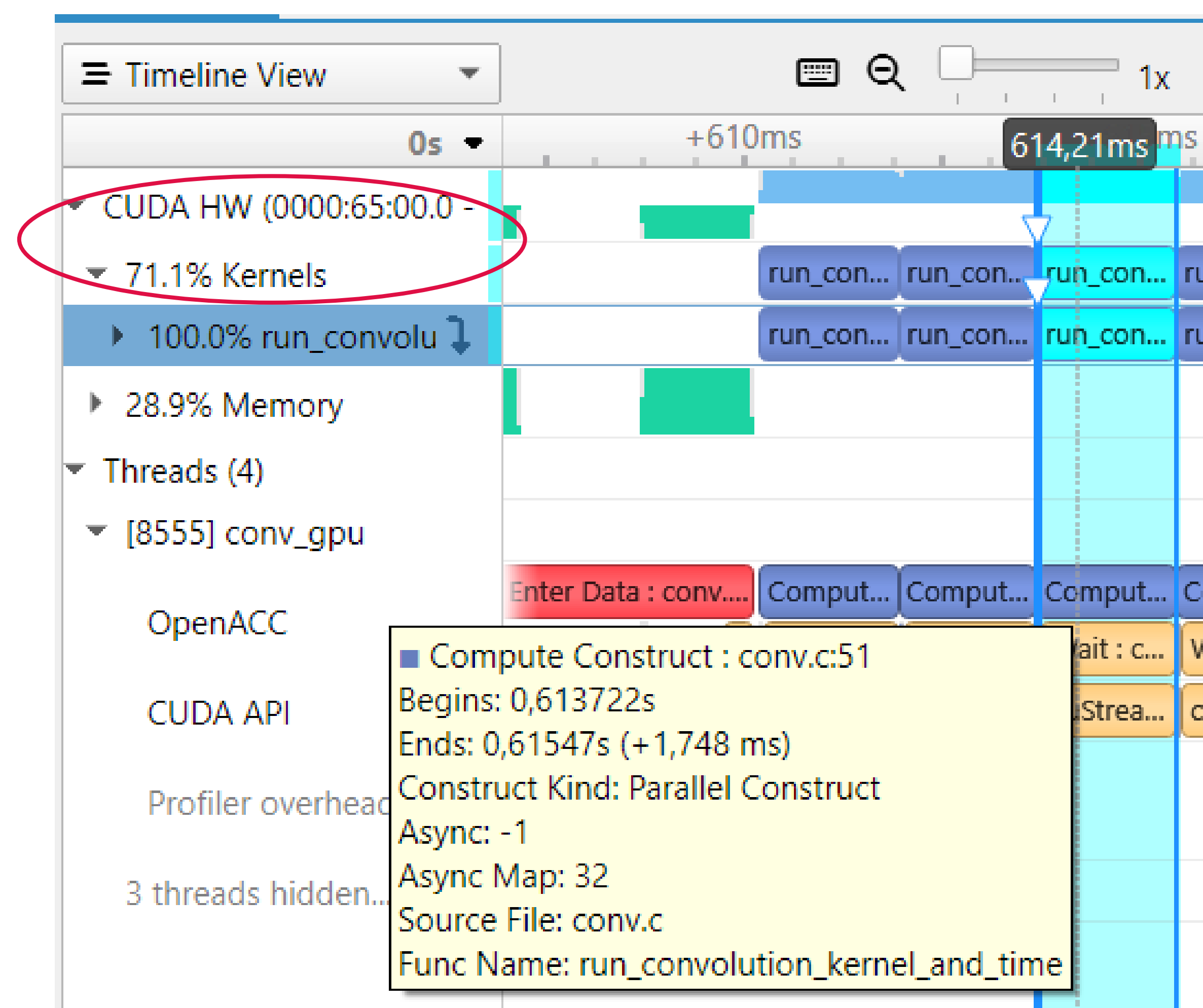
Loop carried dependence of B-> prevents parallelization

Loop carried backward dependence of B-> prevents vectorization

Locating kernels - Nsight Systems timeline

...and how to get to Nsight Compute for kernels

- Record timeline
 - `nsys profile -t cuda,openacc -o task1_initial ./conv_gpu 3`
- Locate kernel and get command line
 - `ncu --kernel-name run_convolution_kernel_and_time_51_gpu --launch-skip 2 --launch-count 1 " ./conv_gpu" 3`



Choose Your Preferences For Analyzing CUDA Kernels

☐ Start NVIDIA Nsight Compute UI to profile the CUDA kernel

The UI will be pre-populated with details from the current report. Preferred option for local and remote profiling.

Location of Nsight Compute UI:

C:/Program Files/NVIDIA Corporation/Nsight Compute 2020.1.1/host/windo ...

☒ Display the command line to use NVIDIA Nsight Compute CLI

Preferred option for profiling manually on the target system.

Note: User must have access to GPU performance counters. See [guide](#) for more details.

☐ Do not ask me again

Use Tools menu to change these settings later

OK Cancel



Using Nsight Compute

- Very powerful and configurable tool
- Command line mode: Useful for quick experiments
- Export into report file, "-o output_filename"
 - Transfer report to local machine, inspect metrics, charts, etc.
- Other recommended options:
 - "--set full" – ensures you record all metrics (collections takes longer)
 - "--import-source on" – ensures the report embeds source file in current optimization state
- Ensure you only record what you need, use the "skip" and "count" options, short -s and -c

Task 1

1. Directory task1/
2. Run the GPU version: Just type "make" (and look at the executed command line)
3. Identify potential issues
 1. Check profiler output, "make ncu"
 2. Also try "make NV_ACC_TIME=1" (or NOTIFY)
 3. Closely look at compiler output.
4. Look for TODO and implement collapse clause
5. Note down new time, and also record a profile via "make profile" (you will need it later)

Task 1 - Results

- NCU output:

Section: Launch Statistics

Block Size		128	
Function Cache Configuration		cudaFuncCachePreferNone	
Grid Size		32	
Registers Per Thread	register/thread	38	
Shared Memory Configuration Size	Kbyte	32.77	
Driver Shared Memory Per Block	Kbyte/block	1.02	
Dynamic Shared Memory Per Block	byte/block	0	
Static Shared Memory Per Block	byte/block	0	
Threads	thread	4096	
Waves Per SM		0.02	

WRN The grid for this launch is configured to execute only 32 blocks, which is less than the GPU's 108 multiprocessors.

Task 1 - Results

- NV_ACC_TIME command line output:

```
$ make NV_ACC_TIME=1
```

```
./conv_gpu 3
```

```
[...]
```

```
51: compute region reached 15 times
```

```
51: kernel launched 15 times
```

```
grid: [32] block: [128]
```

```
elapsed time(us): total=842,551 max=74,560 min=50,411 avg=56,170
```

- Before/after: 56 ms vs. 2.4 ms!
- Relevant profiler output

```
51, Generating NVIDIA GPU code
```

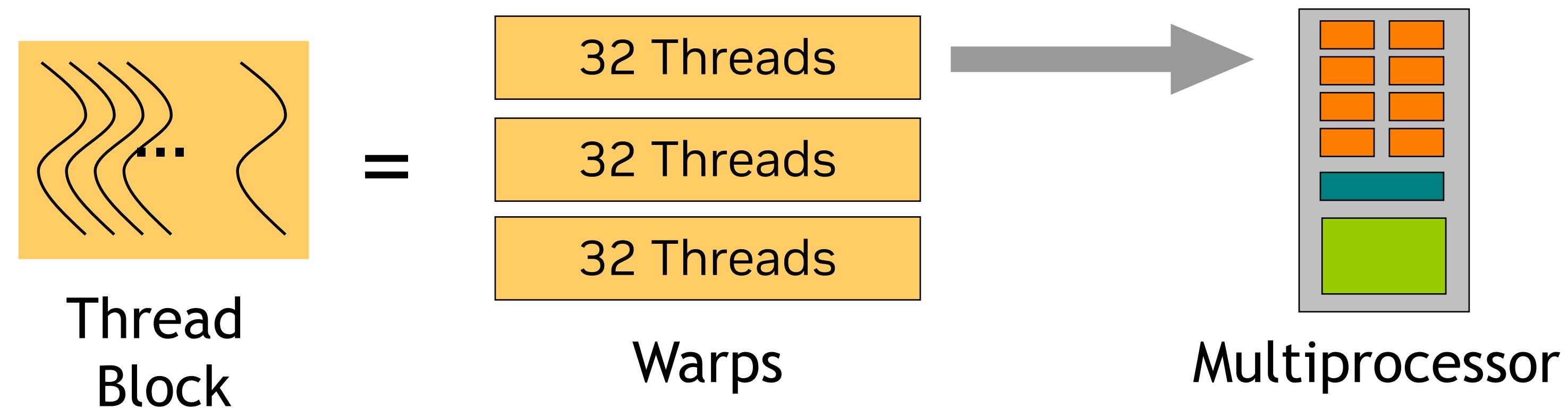
```
56, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
```

```
57, /* blockIdx.x threadIdx.x collapsed */
```

```
59, #pragma acc loop seq
```

```
60, #pragma acc loop seq
```

CUDA Warps



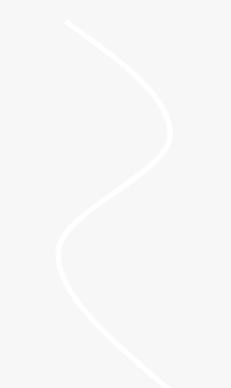
A thread block consists of a groups of warps


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

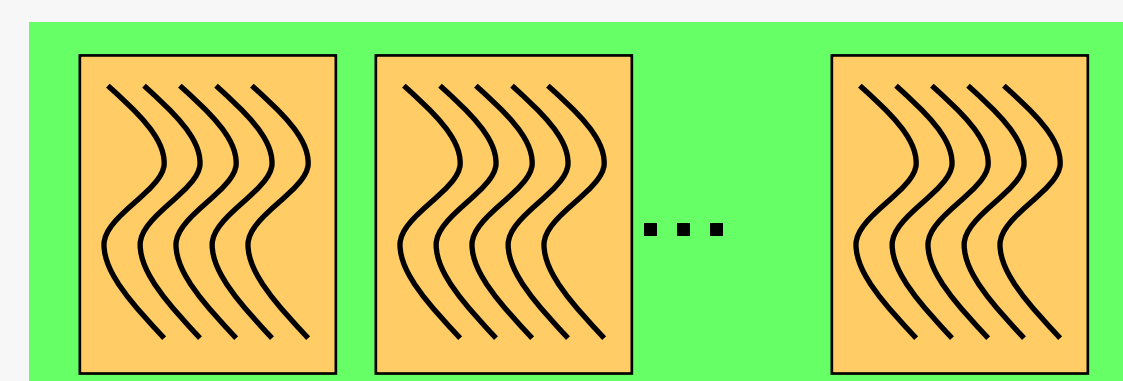
Currently all NVIDIA GPUs use a warp size of 32

CUDA Execution Model


Software

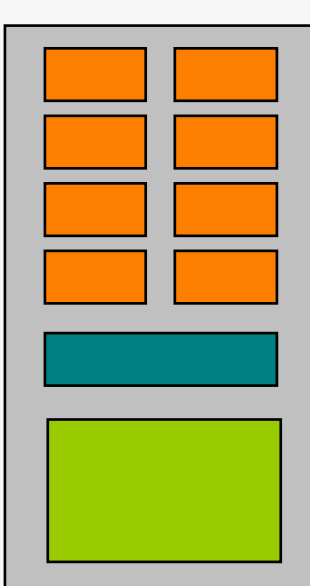

Thread

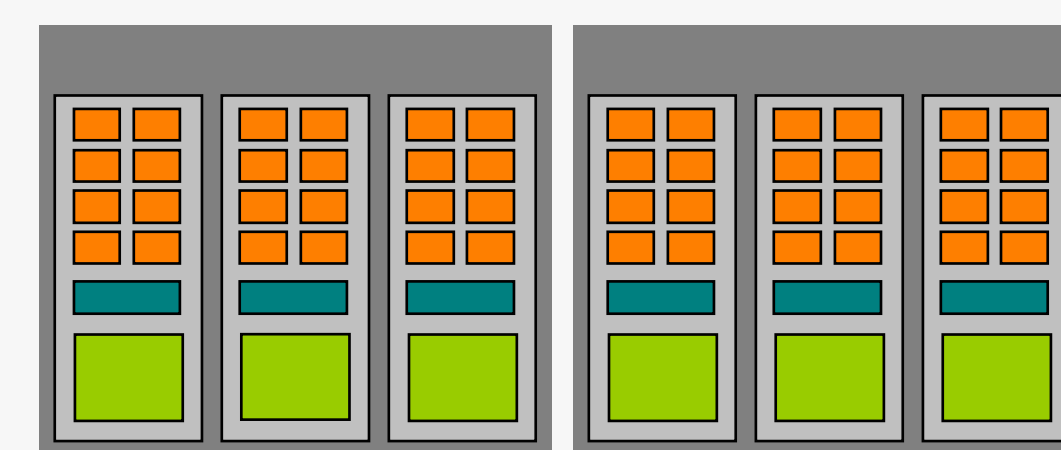

Thread Block


Grid

Hardware


Scalar Processor


Multiprocessor


Device

Threads are executed by scalar processors

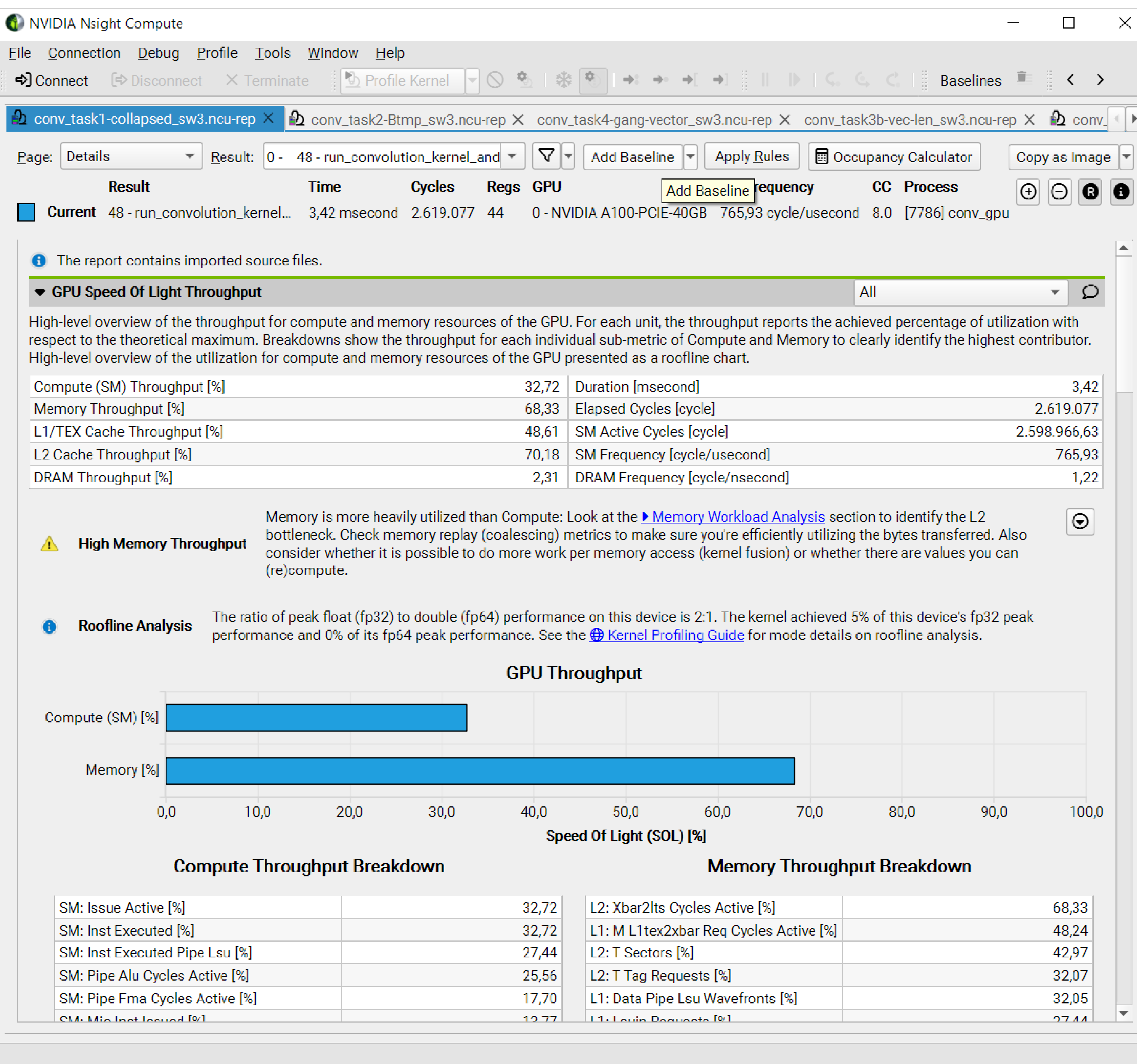
Thread blocks are executed on multiprocessors (SMs)

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)



Using the Nsight Compute GUI

Profile from task 1

- Baseline feature – compare results
- Data-driven,
- SoL throughput: Tables
- Consider warnings/hints from rules

What is a loop-carried dependence?

- Compiler output:

59, Complex loop carried dependence of stencil prevents parallelization

Loop carried dependence of B-> prevents parallelization

Loop carried backward dependence of B-> prevents vectorization

Complex loop carried dependence of B->,A-> prevents parallelization

60, Complex loop carried dependence of stencil,B->,A-> prevents parallelization

Loop carried dependence of B-> prevents parallelization

Loop carried backward dependence of B-> prevents vectorization

- Code this refers to

```
59: for (int sx = -sw; sx <= sw; ++sx) {
60:   for (int sy = -sw; sy <= sw; ++sy) {
61:     const float val =
62:       stencil[sw + sx][sw + sy]
63:       * A[x + sx][y + sy];
64:     B[x][y] += val;
65:   }
}
```

Task 2: Remove the dependence

1. Directory task2/
2. Check compiler output: loop carried dependence
3. Look for TODOs and break the dependency
4. What is the new runtime? Can you see why?
5. Record a profile via "make profile"
 1. Try to compare it with the profile from task 1

Task 2 - Results

- Code should've gotten much slower again, about 180 ms
- The compiler was able to parallelize the inner loop

51, Generating NVIDIA GPU code

```
56, #pragma acc loop gang collapse(2) /* blockIdx.x */
```

```
57,    /* blockIdx.x collapsed */
```

```
59, #pragma acc loop seq
```

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

```
    Generating implicit reduction(+:B_tmp)
```

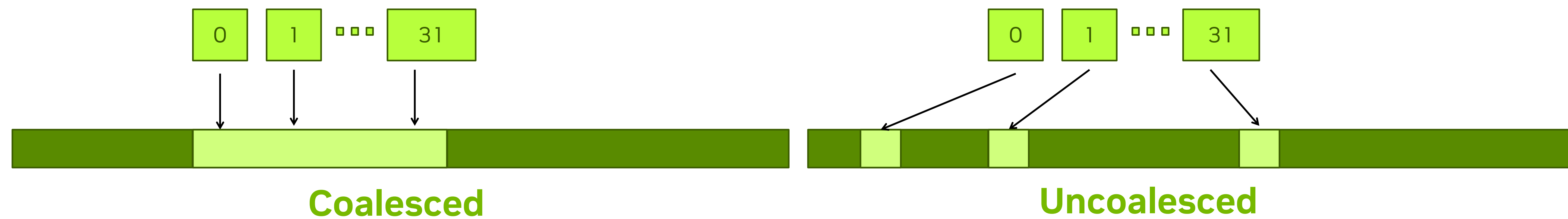
59, Loop is parallelizable

60, Loop is parallelizable

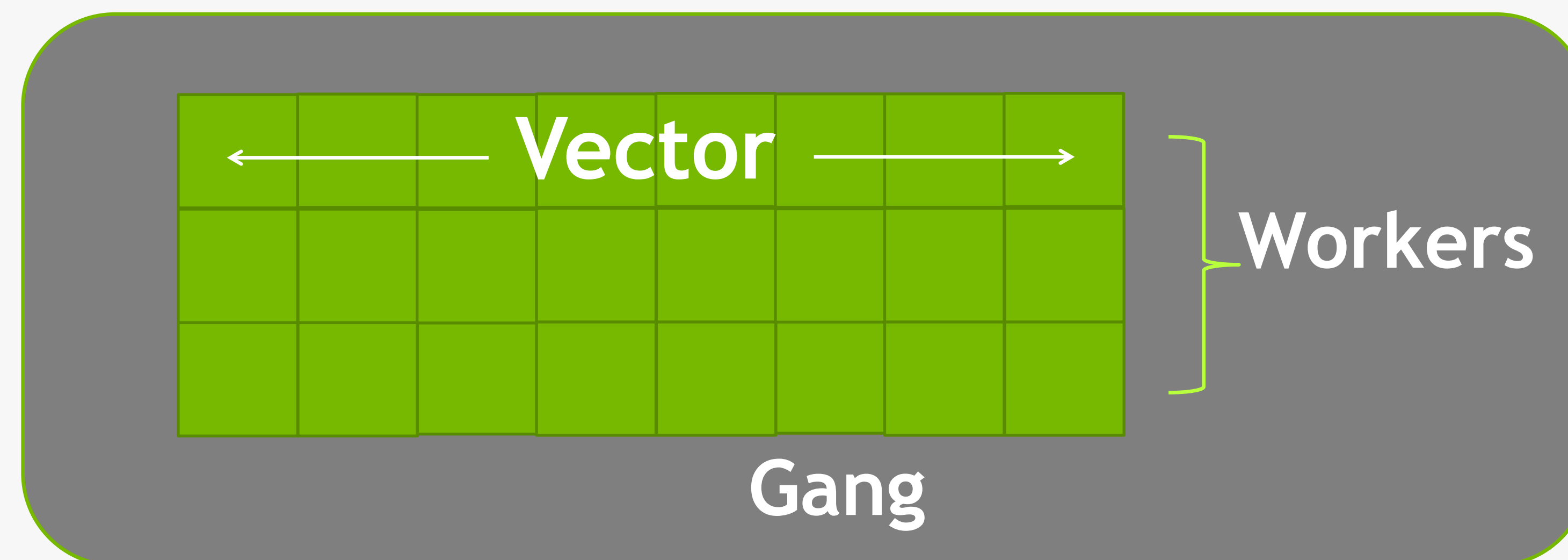
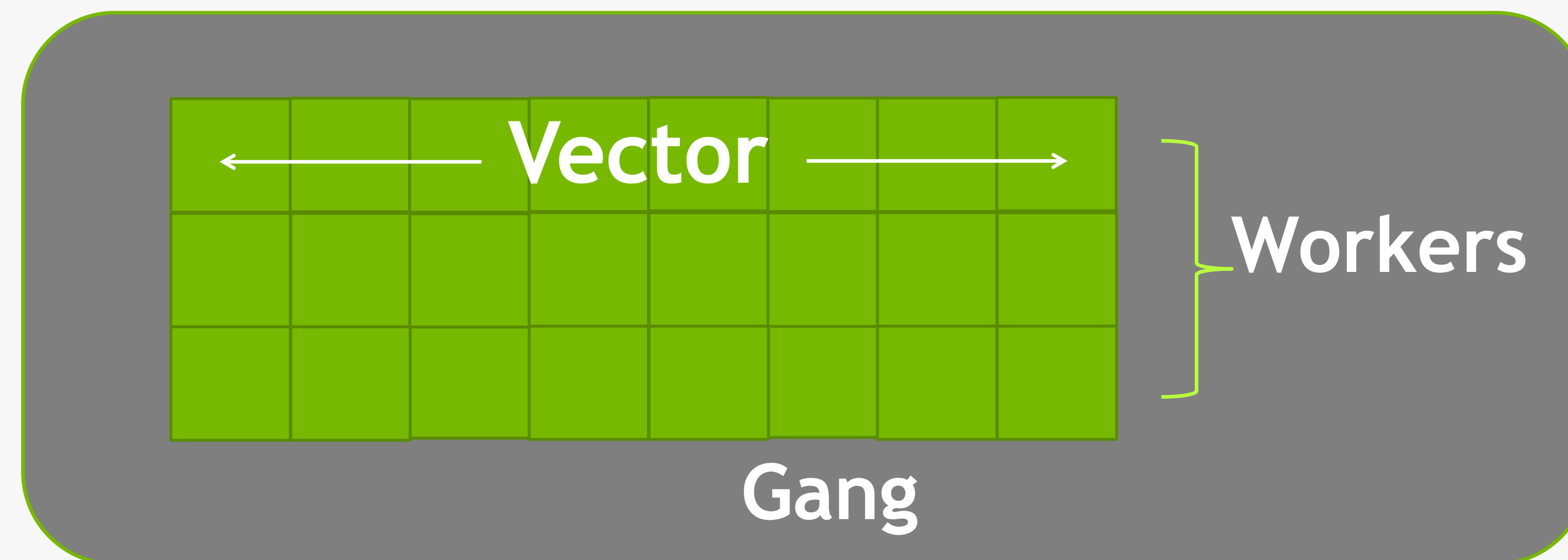
- But why is this slower?
 - Memory access patterns
 - Extra reduction

Memory Coalescing

- Coalesced access:
 - A group of 32 contiguous threads („warp“) accessing adjacent elements
 - Few transactions and high utilization
- Uncoalesced access:
 - A warp of 32 threads accessing scattered elements
 - 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

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 $\text{num_workers} * \text{vector_length}$

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)
 - Note: We have not used "worker" parallelism in our example

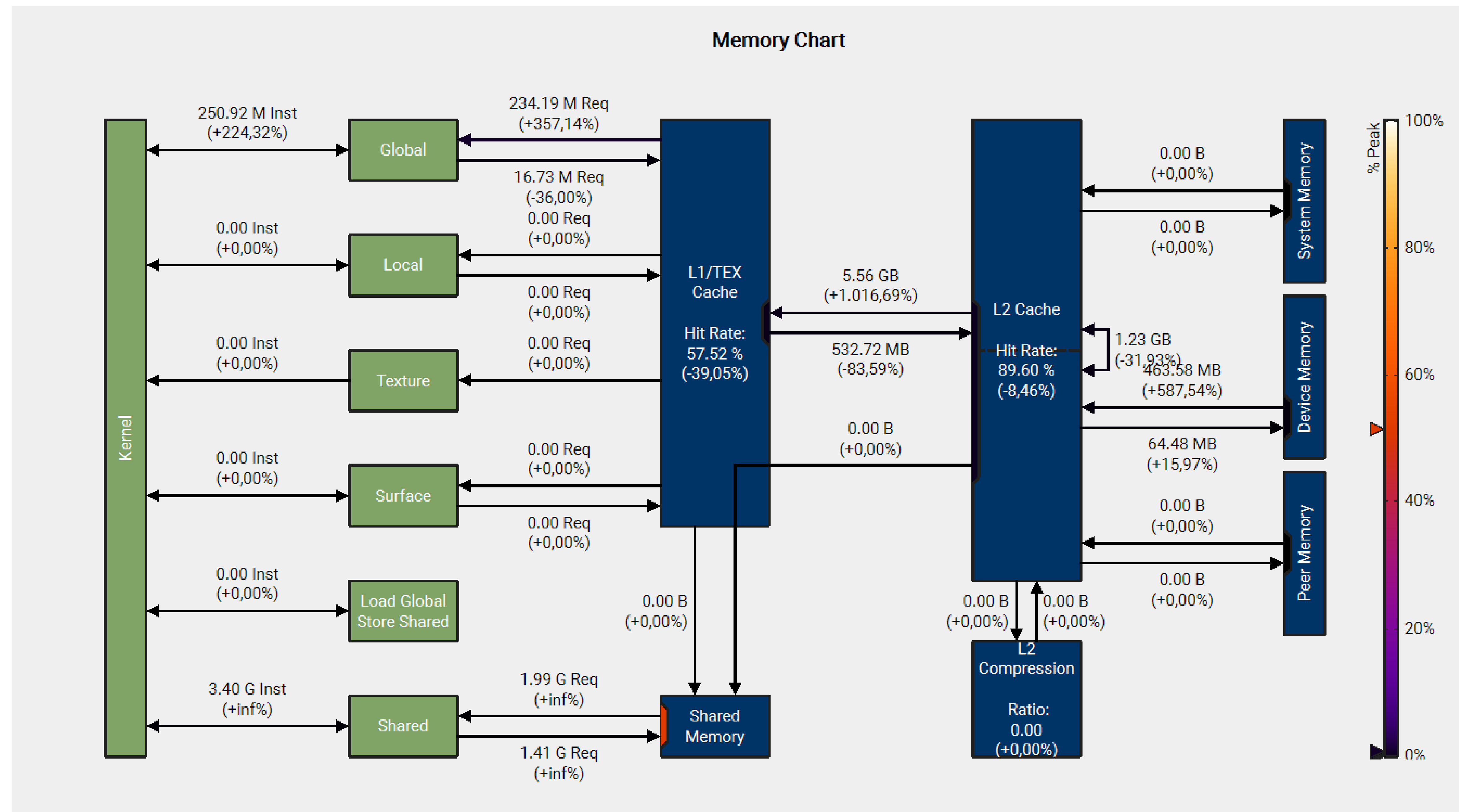
```
#pragma acc parallel loop gang worker  
for (int x = sw; x < N - sw; ++x) {  
...  
    #pragma acc loop vector  
    for (int sx = -sw; sx <= sw; ++sx) {
```



gang, worker, vector appear once per parallel region

Nsight Compute profile

A closer look at Task 2



⚠ Uncoalesced Global Accesses

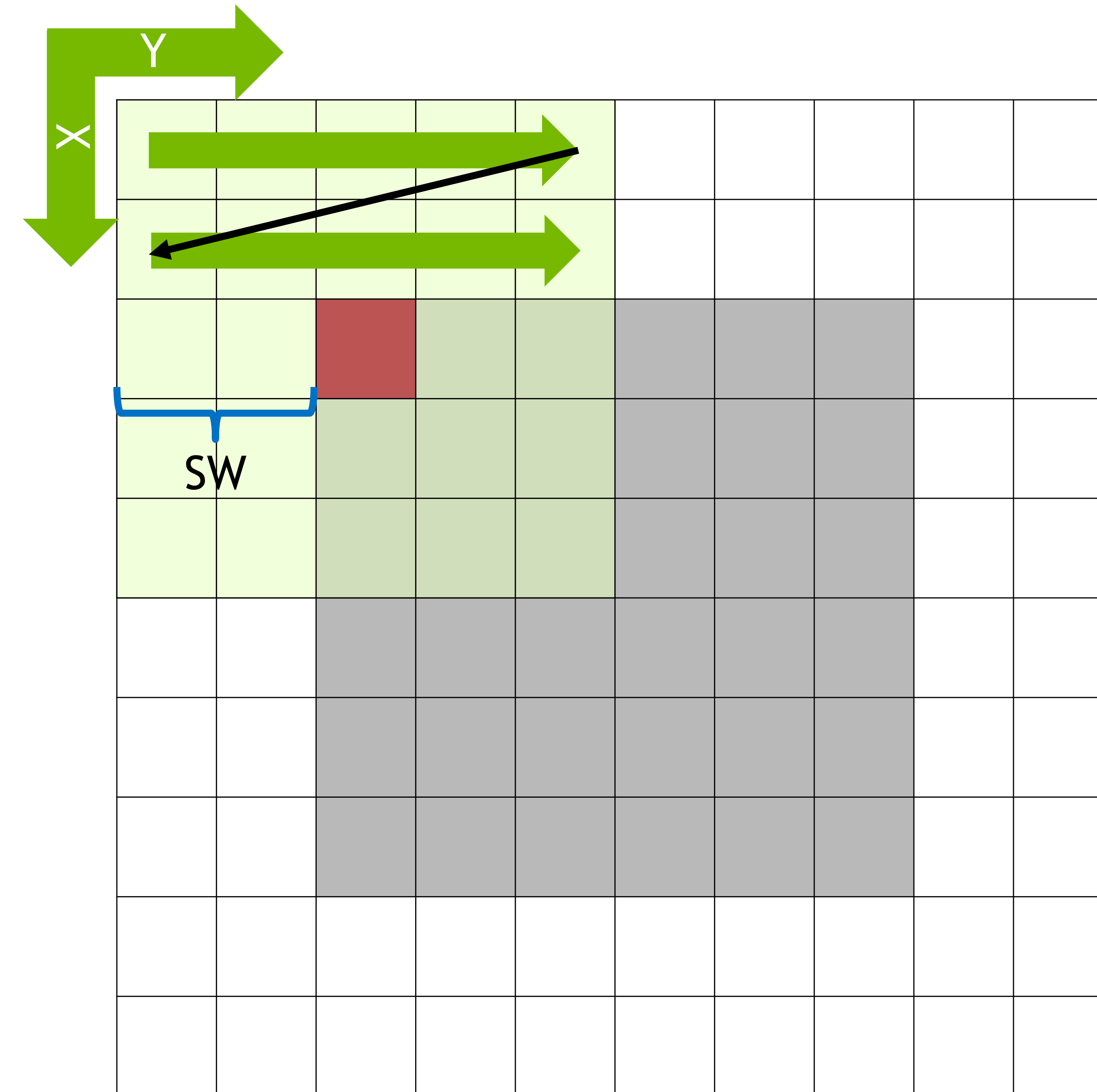
This kernel has uncoalesced global accesses resulting in a total of 188148180 excessive sectors (43% of the total 439069680 sectors). Check the L2 Theoretical Sectors Global Excessive table for the primary source locations. The [CUDA Programming Guide](#) had additional information on reducing uncoalesced device memory accesses.

Task 3: Understand slowdown and test fixes

1. Directory task3/
2. Use compiler output, and Nsight Compute, try to understand memory access patterns
 1. Look for memory traffic, worse cache hit rates, uncoalesced access %,
 1. also mem tables: global load vs. store – less stores (reduction), but a lot more loads
3. See TODOs to implement one variant (optionally: draw yourself a diagram on paper!)
 1. Outer stencil as vector loop
 2. Add vector_length(32)
4. Record a profile and compare again

Task 3 - Results

- Outer stencil as vector loop: 47 ms
 - Add vector_length(32): 17.3 ms
 - Still: Have not recovered original performance
- Why does length help? Only 49 points, i.e. $(3+1+3)^2$



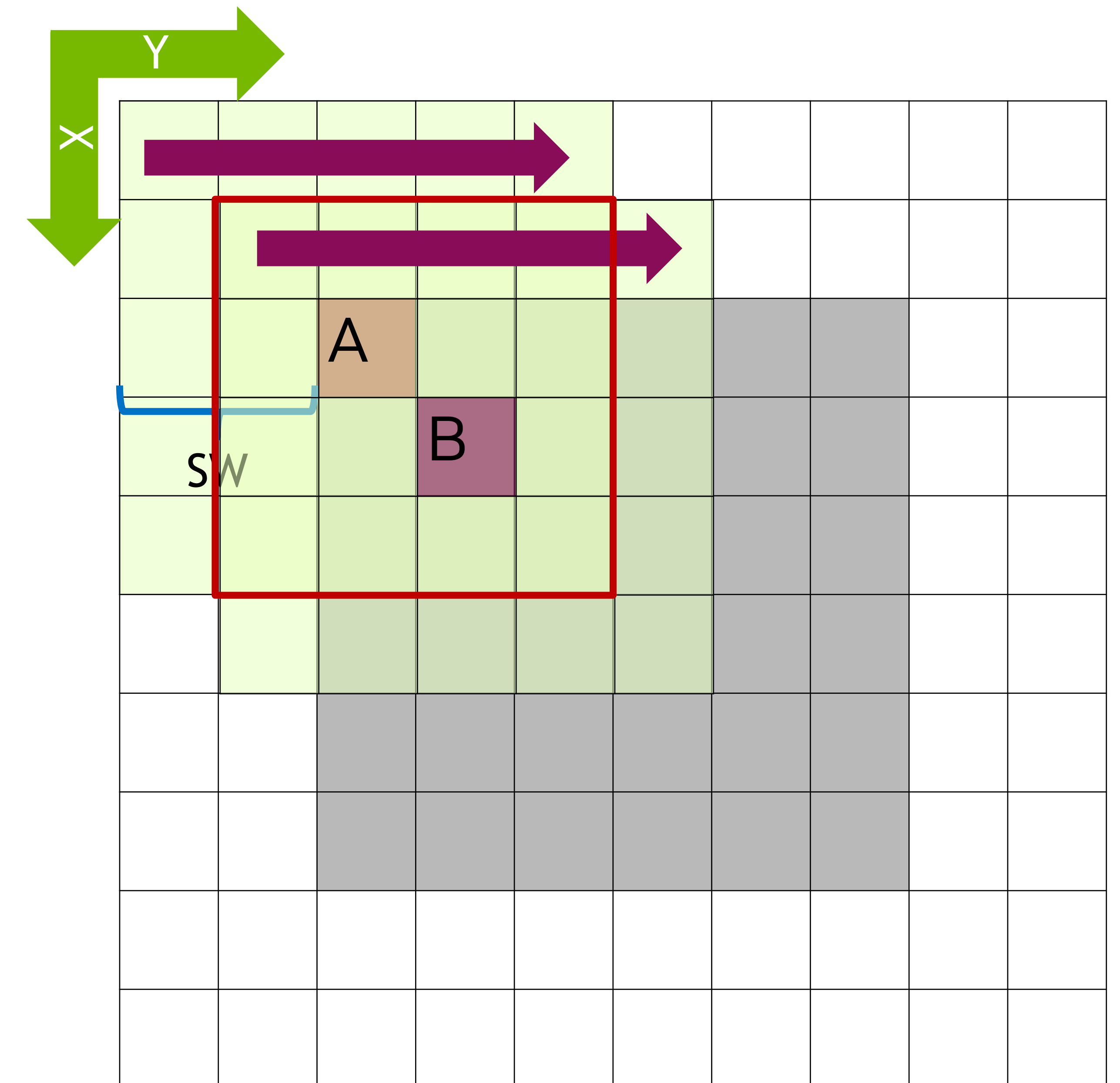
Task 4: Recover and improve performance

1. Directory task4/ (and solution/)
2. Follow the single TODO and measure the runtime
3. Record a profile (again "make profile")
4. Can you find clues on the optimality of the solution?
 1. Hint: look at the roofline diagram
5. Can you improve it further?

Task 4 - Results

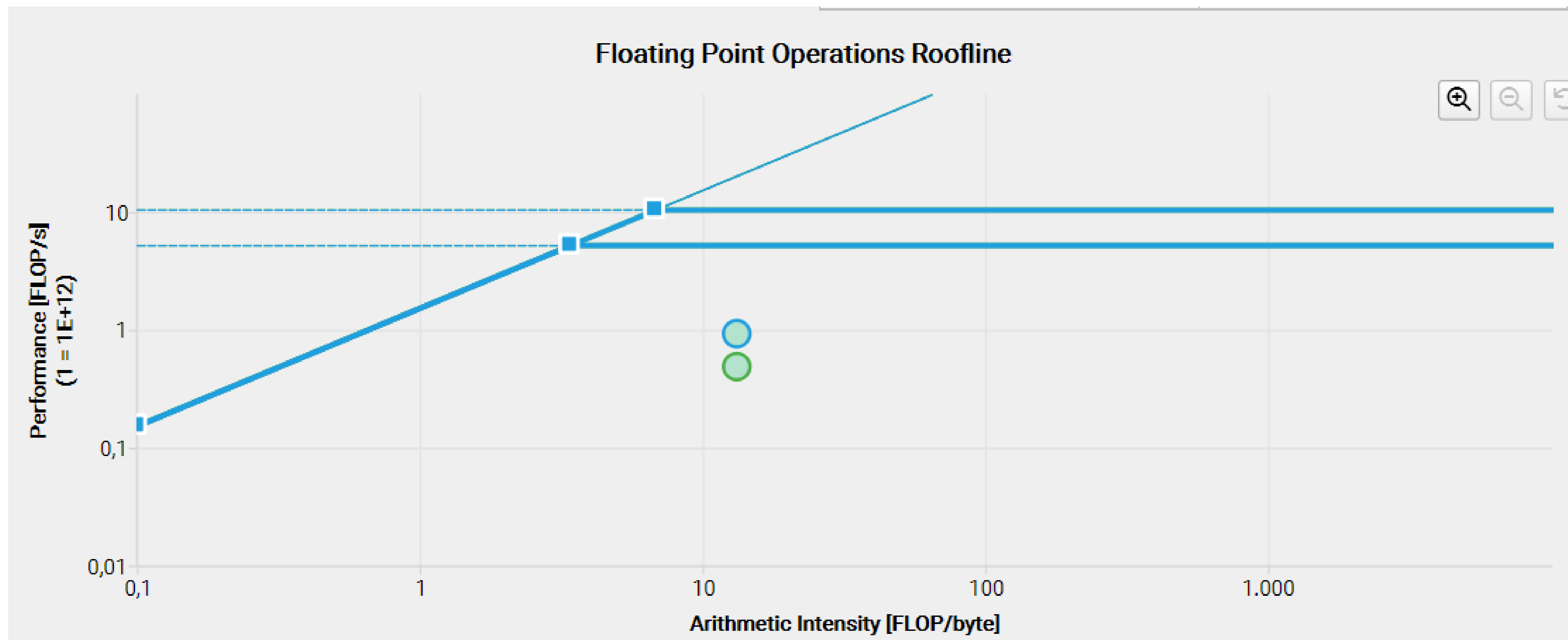
- Runtime about 1.5 ms
- Memory is now more cache-friendly
 - 56, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
- 56: `for (int x = sw; x < N - sw; ++x) {`
- 57: `for (int y = sw; y < N - sw; ++y) {`
- Accesses to input data:

 `A[x + sx][y + sy];`
- Outer loop (blocks) over rows, inner loop (threads) over columns
- Temporal and spatial locality: Each thread loops over its stencil area
 - Neighboring threads from surrounding blocks share cached data



Roofline model

- Powerful tool to judge how well hardware is utilized
- FLOPs vs. AI – "how often is each transferred byte used"
- Compute bound, but below roof
 - Inefficiencies in memory accesses



Summarizing the Steps

Task	Action taken	Time [ms]
Serial version, starting point	None	1200
Task 0	Add "parallel loop"	165 (multicore) 56 (GPU)
Task 1	Collapse outer loops, expose parallelism	2.4
Task 2	Break loop-carried dependency, causes compiler to apply "vector" to innermost loop, add reduction	180 (slowdown)
Task 3a	Move "vector" to second-outermost loop	46
Task 3b	Use shorter vector_length(32)	17.3
Task 4	Outermost loop as "gang vector"	1.5

Further tinkering

- Compare stencil size: How does runtime scale with it?
 - small size $sw=2$, latency effects
- Experiment with different stencil sizes (2, 3, 5)
 - don't forget reference data
- You can adapt the stencil to actually perform image filtering operations
 - Simple image loading libraries available
 - Note: We did not normalize stencil coefficients – don't forget to do so

Conclusion

- The Nsight Profilers can be used to identify performance bottlenecks in applications and OpenACC Kernels
 - But most importantly, combine with knowledge of code
 - Closely look at compiler output
- Coalescing memory accesses is important for performance
 - Ordering of loop clauses can have large impact
- Iterative methods and workflow – look for clues, experiment, compare!

