SeSE course: GPU programming for Machine Learning and Data Processing

Debugging and profiling

A look back

- We discussed OpenMP Target, Thrust, Cuda, MinPy, afnumpy, Numba
  - If you already know the related technology, go for that:
    - OpenMP ⇒ OpenMP Target
    - C++ STL ⇒ Thrust
    - numpy ⇒ MinPy, afnumpy, CuPy
    - Numba and CUDA tools of choice if you really want to face the GPU
- `./notebook.py -- gpulibs` brings working `numba` and `cupy`, another "numpy look-alike"
- `./notebook.py -- gpulibs af.sif` brings working `afnumpy`

- CuPy implements part of numpy on GPU
- Allows to write inline CUDA C++ code:

```
>>> x = cp.arange(6, dtype='f').reshape(2, 3)
>>> y = cp.arange(3, dtype='f')
>>> kernel = cp.ElementwiseKernel(
...     'float32 x, float32 y', 'float32 z',
...     '''
...     if (x - 2 > y) {
...       z = x * y;
...     } else {
...       z = x + y;
...     }
...     ''', 'my_kernel')
>>> kernel(x, y)
array([[ 0.,  2.,  4.],
       [ 0.,  4.,  10.]], dtype=float32)
```

- Basic see https://docs.cupy.dev/en/stable/user_guide/basic.html

Kernels in CuPy

- Reduction kernels in CuPy: `https://docs.cupy.dev/en/stable/user_guide/kernel.html`

```
>>> l2norm_kernel = cp.ReductionKernel(
...     'T x',  # input params
...     'T y',  # output params
...     'x * x',  # map
...     'a + b',  # reduce
...     'y = sqrt(a)',  # post-reduction map
...     '0',  # identity value
...     'l2norm'  # kernel name
... )
>>> x = cp.arange(10, dtype=np.float32).reshape(2, 5)
>>> l2norm_kernel(x, axis=1)
array([ 5.477226 , 15.9687195], dtype=float32)
```

- Concept similar to use of Thrust: A reduction is found by
  - Reduction expression: How to reduce the two special variables `a` and `b`
  - Mapping expression: Pre-processing each element to be reduced
  - Post-mapping expression: Transform result after computation is done
  - Identity value

Code debugging

- Debugging is important part of any kind of programming
  - To some extent, the GPU environment is more challenging
- It is good to have tools to verify behavior, but also to ensure correctness
- For performance, measurements (timing measurements) and profiling are essential
  - Even more challenging in a GPU context

Debugging on the GPU: Print variables

- The GPU is a separate chip, with separate memory
- Unless you have a separate rendering surface, you have no way to send information to the user
- You can not call arbitrary host APIs to write to files etc
- But...
  - Nvidia realized that this was a challenge
  - There is a special printf function that "just works"
  - Limited buffer size
  - Can still be useful to check the value of some specific variable (not of all threads!) or simply if some code is ever run

Actual debugger on GPU

- Standard gnu debugger `gdb` has a special version for CUDA code, `cuda-gdb`
  - Can step through both CPU and GPU code
  - `gdb` is a very low-level debugging tool, not very user-friendly
  - Various wrappers around `gdb` available such `ddd`
  - Many of these can be configured to work against `cuda-gdb`

Compiler flags with debugging

- `clang++` needs the flag `-g` (or slightly better `-ggdb3`)
- For `nvcc`, specify `-g -G` to include debugging information for host and device
- Optional: Remove -O3 and replace with -O0 or -Og
  - Reduces optimization, making it easier to keep track of data and execution order
  - `-G` in `nvcc` also disables optimization
  - Alternative option `-lineinfo` tries to add debug information to optimized code

Example of cuda-gdb

- Local variables from OpenMP Target not visible in cuda-gdb when working with LLVM compiler

```
Reading symbols from openmptarget...done.
(cuda-gdb) b 29
Breakpoint 1 at 0x40212b: file openmptarget.cpp, line 29.
(cuda-gdb) r
Starting program: /home/martinkr/teaching/sesegpu/openmptarget
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x2aaab4bc8700 (LWP 33392)]
Loading file /home/martinkr/.keras/datasets/mnist.npz
OpenMP target.
Procesing 10000 * 10000 elements
[New Thread 0x2aaab4dc9700 (LWP 33401)]
[Switching focus to CUDA kernel 0, grid 3, block (2,0,0),
     thread (64,0,0), device 0, sm 2, warp 1, lane 0]

Thread 1 "openmptarget" hit Breakpoint 1,
  __omp_offloading_29_ec694e7_main_l19<<<(128,1,1),(128,1,1)>>> ()
    at /home/martinkr/teaching/sesegpu/openmptarget.cpp:30
30      for (int y = 0; y < side; y++)
(cuda-gdb)
```

gdb and cuda-gdb commands

- `r`(un) start program
- `c`(ontinue) resume running
- `n`(ext) step to next line
- `b`(reak point) *lineno* set breakpoint at this line in source code
- `print` *variable* print the value of a variable (or some other expression)
- `info locals` list all local variables
- `where` print call stack
- `info cuda` info on threads for current sm, warps for current sm, lanes in current warp, and many more
- `cuda ...` change currently focused thread/sm/lane/...
- Use `help` command with both of these
- Read more: `https://docs.nvidia.com/cuda/cuda-gdb/index.html#inspecting-program-state`

Look into `info cuda warps`, Initial stage


```
(cuda-gdb) info cuda warps
  Wp Active Lanes Mask Divergent Lanes Mask Active Physical PC Kernel  BlockIdx First Active ThreadIdx
Device 0 SM 0
   0        0x00000002          0xfffffffd 0x0000000000000a90      0   (0,0,0)              (33,0,0)
   1        0x00004000          0xffffbfff 0x0000000000000a90      0   (0,0,0)              (78,0,0)
   2        0x00000001          0x00000000 0x0000000000002ae0      0   (0,0,0)              (96,0,0)
   3        0x7fff0000          0x8000ffff 0x0000000000000a90      0   (0,0,0)              (16,0,0)
   4        0x80000000          0x7fffffff 0x0000000000000a90      0   (1,0,0)              (31,0,0)
   5        0x80000000          0x7fffffff 0x0000000000000a90      0   (1,0,0)              (63,0,0)
*  6        0x00007fff          0xffff8000 0x0000000000000d60      0   (1,0,0)              (64,0,0)
   7        0x00000001          0x00000000 0x0000000000002ae0      0   (1,0,0)              (96,0,0)
   8        0x00000001          0x00000000 0x0000000000002ae0      0  (80,0,0)              (96,0,0)
   9        0x80000000          0x7fffffff 0x0000000000000a90      0  (80,0,0)              (31,0,0)
  10        0x80000000          0x7fffffff 0x0000000000000a90      0  (80,0,0)              (63,0,0)
  11        0x00003fff          0xffffc000 0x0000000000000b60      0  (80,0,0)              (64,0,0)
  12        0x00001ffe          0xffffe001 0x0000000000000a90      0 (120,0,0)              (65,0,0)
  13        0x00000001          0x00000000 0x0000000000002ae0      0 (120,0,0)              (96,0,0)
  14        0x80000000          0x7fffffff 0x0000000000000a90      0 (120,0,0)              (31,0,0)
  15        0x80000000          0x7fffffff 0x0000000000000a90      0 (120,0,0)              (63,0,0)
  16        0x00000000          0x00000000                n/a    n/a       n/a                  n/a
  17        0x00000000          0x00000000                n/a    n/a       n/a                  n/a
  18        0x00000000          0x00000000                n/a    n/a       n/a                  n/a
  19        0x00000000          0x00000000                n/a    n/a       n/a                  n/a
  20        0x00000000          0x00000000                n/a    n/a       n/a                  n/a
  21        0x00000000          0x00000000                n/a    n/a       n/a                  n/a
  22        0x00000000          0x00000000                n/a    n/a       n/a                  n/a
  23        0x00000000          0x00000000                n/a    n/a       n/a                  n/a
  24        0x00000000          0x00000000                n/a    n/a       n/a                  n/a
  25        0x00000000          0x00000000                n/a    n/a       n/a                  n/a
  26        0x00000000          0x00000000                n/a    n/a       n/a                  n/a
  27        0x00000000          0x00000000                n/a    n/a       n/a                  n/a
  28        0x00000000          0x00000000                n/a    n/a       n/a                  n/a
  29        0x00000000          0x00000000                n/a    n/a       n/a                  n/a
  30        0x00000000          0x00000000                n/a    n/a       n/a                  n/a
  31        0x00000000          0x00000000                n/a    n/a       n/a                  n/a
```

Observation from `cuda-gdb`

- 16 warps run on this SM from different blocks (0, 1, 80, 120)
- Maximum of 16 blocks per SM, 32 warps per SM
- Each execution unit has a limited number of registers
- Each block defines its maximum usage of registers/warp and and shared memory per block
  - If a warp function is complex, usage of these resources might increase, decreasing occupancy
- `nvcc` has a flag `-resource-usage` reporting these numbers
  - Check if numbers are high to see what happens
- Try again using the `nvcc` flag `-lineinfo`

- With highly parallel code, with manually computed indices, it is very easy to write to the wrong memory addresses
- `cuda-memcheck` is a tool that was developed to address this problem
  - Runs the code in its original form
  - But with lots of extra checks (far slower than normal)
    - 6 minutes rather than 415 ms for our the prelab3 `./cuda` program
  - Report about invalid accesses
- Since this tool is working directly on the GPU, it can be used with any binary running on the GPU
- E.g. Python with `numba`, TensorFlow, if you believe that they misbehave

Tool modes of `cuda-memcheck`

- `cuda-memcheck` has several subtools
  `cuda-memcheck --tool x`
  - `memcheck` is the default, checking invalid writes
  - `initcheck` verifies that memory is properly initialized before the first read
  - `synccheck` verifies that synchronization primitives do not have hidden usage errors (such as not all lanes participating)
  - `racecheck` checks for race conditions in shared memory, e.g. where writes and reads on different threads do not have a barrier
- Extra flags giving additional information

```
--leak-check <full|no> [Default : no]
        Print leak information for CUDA allocations.
        NOTE: Program must end with cudaDeviceReset() for this to work.

--track-unused-memory <yes|no> [Default : no]
        Check for unused memory allocations. This requires initcheck tool.
```

Code optimization with profilers

- Debuggers useful to retrieve correct allocation and access
- Can also look into some performance characteristics
- Nvidia Nsight profiler main profiling tool
  - Nsight System: Interaction between CPU and GPU, multiple kernels
    - Read more: `https://docs.nvidia.com/nsight-systems/UserGuide/index.html`
  - Nsight Compute: Focus on individual kernels
    - Read more: `https://developer.nvidia.com/blog/using-nsight-compute-to-inspect-your-kernels/`

Capabilities of Nsight Compute

- Measure performance
- Total time usage
- Total number of memory fetches
  - Cache hit rates
  - From different memory levels
- What instructions are stalling?
- Occupancy: How many threads per SM
- Usage of execution resources
  - IPC, overall, when active
  - What units are used
- Our Cuda kernels would show that the floating-point units are not utilized at all
  - We only work with integer data

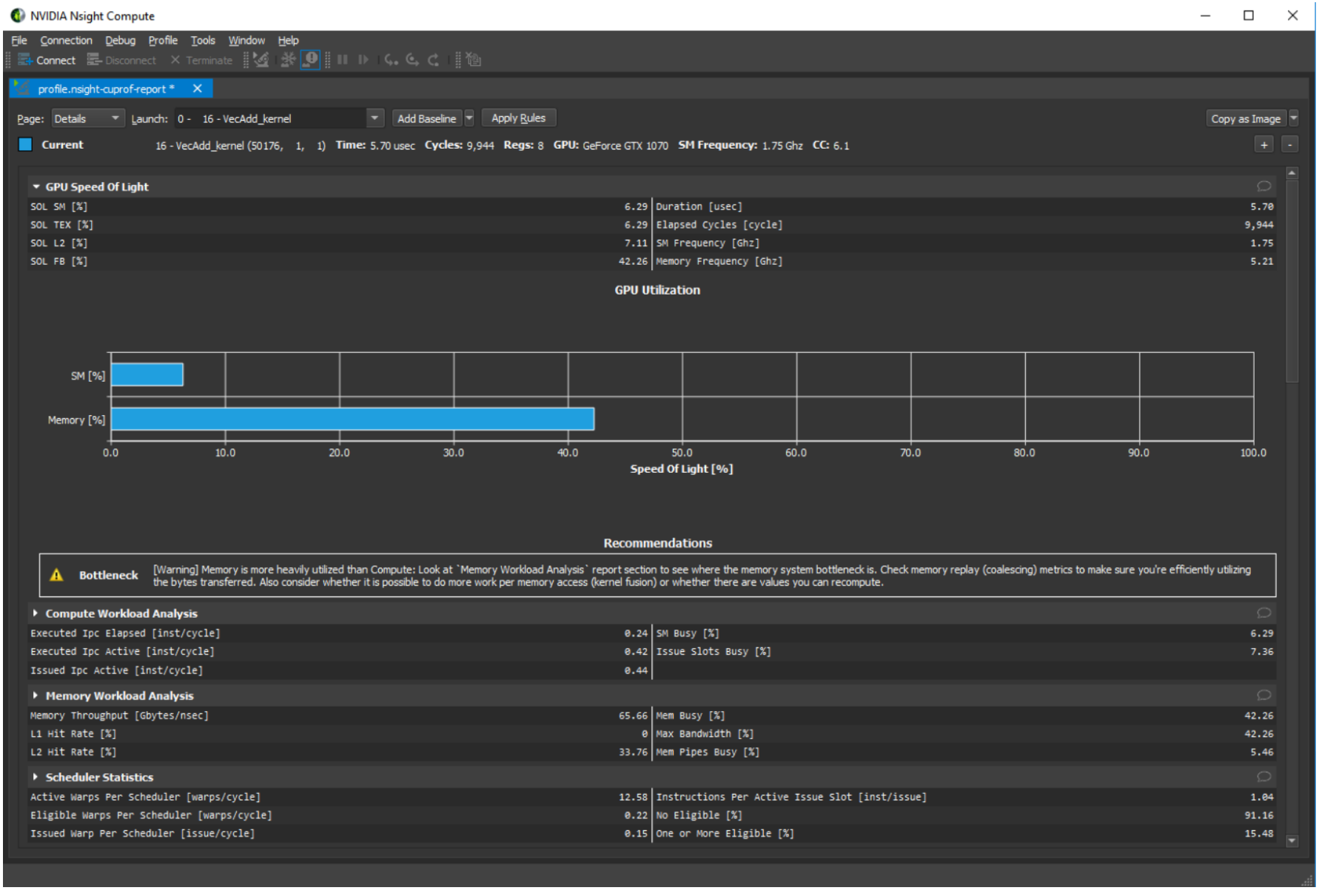
Example output of command-line with Nsight Compute

- Run as `ncu ./executable`

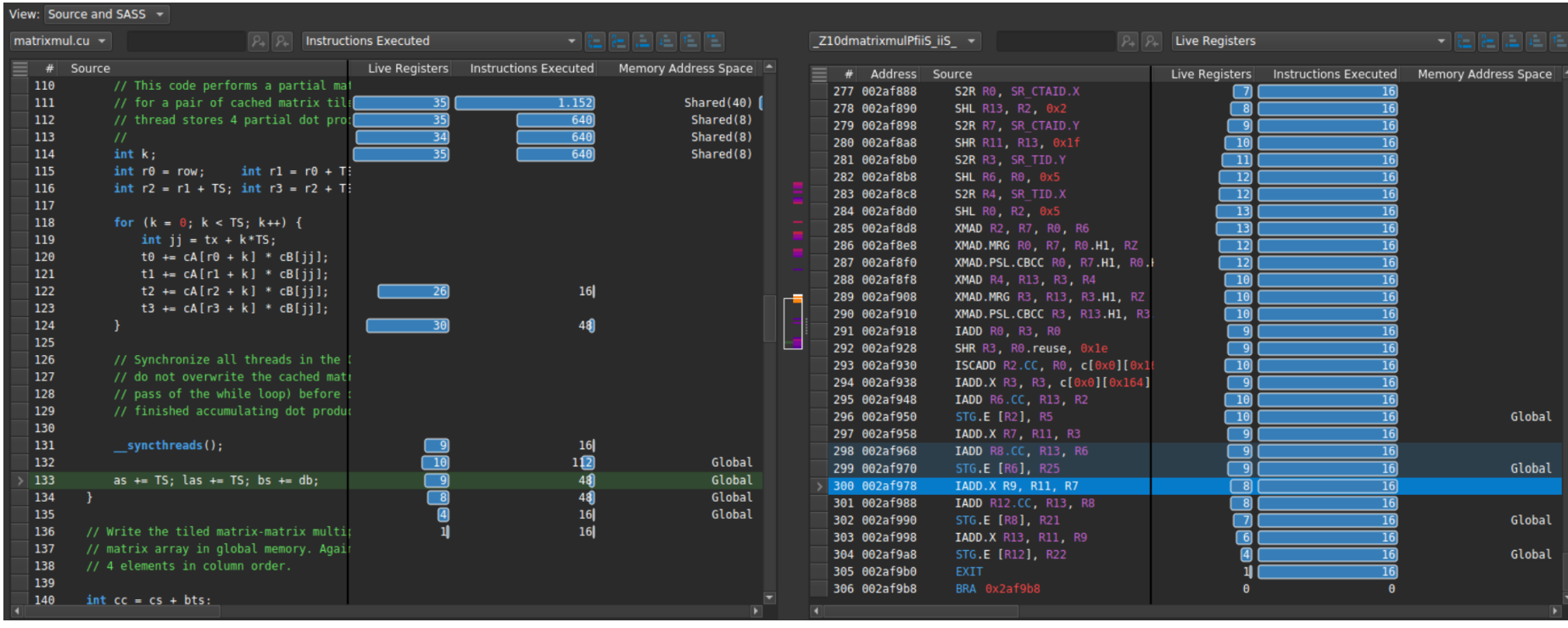
```
compute_triad(int, float, const float *, const float *, float *), 2021-Nov-11 18:21:16, Context 1, Strea
  Section: GPU Speed Of Light Throughput
  ---------------------------------------- ------------- ------------------------------
  DRAM Frequency                          cycle/nsecond                           4.96
  SM Frequency                            cycle/usecond                         580.57
  Elapsed Cycles                                  cycle                        521,701
  Memory [%]                                          %                          89.74
  DRAM Throughput                                     %                          89.74
  Duration                                      usecond                         898.59
  L1/TEX Cache Throughput                             %                          30.13
  L2 Cache Throughput                                 %                          29.97
  SM Active Cycles                                cycle                     501,442.58
  Compute (SM) [%]                                    %                          23.35
  ---------------------------------------- ------------- ------------------------------
  INF   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.

  Section: Launch Statistics
  ---------------------------------------- ------------- ------------------------------
  Block Size                                                                       512
  Function Cache Configuration                                  cudaFuncCachePreferNone
  Grid Size                                                                     38,061
  Registers Per Thread                    register/thread                          16
  Shared Memory Configuration Size                  Kbyte                       32.77
  Driver Shared Memory Per Block               byte/block                           0
  Dynamic Shared Memory Per Block              byte/block                           0
  Static Shared Memory Per Block               byte/block                           0
  Threads                                         thread                    19,487,232
  Waves Per SM                                                                 475.76
  ---------------------------------------- ------------- ------------------------------
```

Nsight Compute: Sample view with GUI



Nsight Compute: Look into assembly code



- View allows to inspect the time spent for instructions, register pressure, memory accesses
- View into source code and corresponding assembler code to find out what actually happens

Recommendations for code tuning

- Choose a systematic approach for performance improvements (holds both on GPU and CPU)
- Approach the analysis with a hypothesis
  - What am I expecting to happen and why (think about hardware capabilities and algorithm)?
  - How can I verify or disprove hypothesis?
- Try to isolate one aspect of behavior from other factors
  - Analyze scaling of times with increase in size of dataset – is behavior correct?
  - Analyze run time as a function of resources – "strong scaling"?
  - If you can, try on more than one hardware or configuration (compiler, framework, etc.)
- Try to link real-world cases with simplified problems, cross-checking validity of simplification
- If I claim performance is optimal, try to compare compute performance or memory bandwidth to machine specs
  - In our examples, we are pretty far away from optimal performance (both on CPU and GPU)

Profilers for performance analysis

- Measuring timings as we do in our examples already reveal a trend in performance
- Timers useful when assessing several variants – try to test hypotheses why some choice was better than the other
- Profilers take some time to learn, but they make it much easier to formulate hypotheses and evaluate experiments
  - Especially useful when you do not have a complete overview of algorithms or insight into the computing parts of a code
  - Allows you to refine the hypothesis for the next set of experiments
- Profilers most powerful with intermediate level of knowledge in algorithm and performance
  - In my own HPC research, I often "only" use timers and low-level (text-based) output (such as performance counters) more than GUI-based profiles, because I know the algorithms and the hardware well