Miniweather Application GPU Acceleration

Michel Herrera Sanchez
DLI Instructor Evaluation OpenACC

GPU Computing

Differences between CPU and GPU Architecture

CPU Architecture

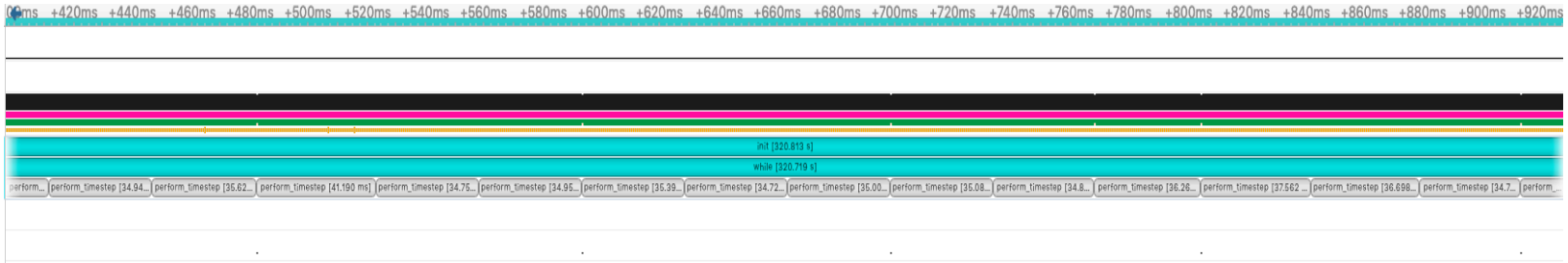
- Designed for Latency workloads
- Lower density of cores concentrations inside the CPU die
- Faster processors speed
- Large Main Memory Capacity and larges L2/L1 Caches
- Complex hardware with Branch Predictor
- Data Parallel Execution using SIMD AVX-512

NVIDIA GPU Architecture

- Designed for high throughput workloads
- High density of cores concentrations inside the GPU die (thousands)
- Slower processors speed
- High Bandwidth Memory Access
- GPU Memory is not integrated in the Host and requires data transfer
- Data Parallel Execution using SIMT Warp Size 32

GPU Computing

Expected speedup



By Amdahl Law : speedup = 1 / (s + p / N)

s -> serial proportion = 320.813/641.532 = 0.50

p -> parallelized part = 1 – s = 0.50

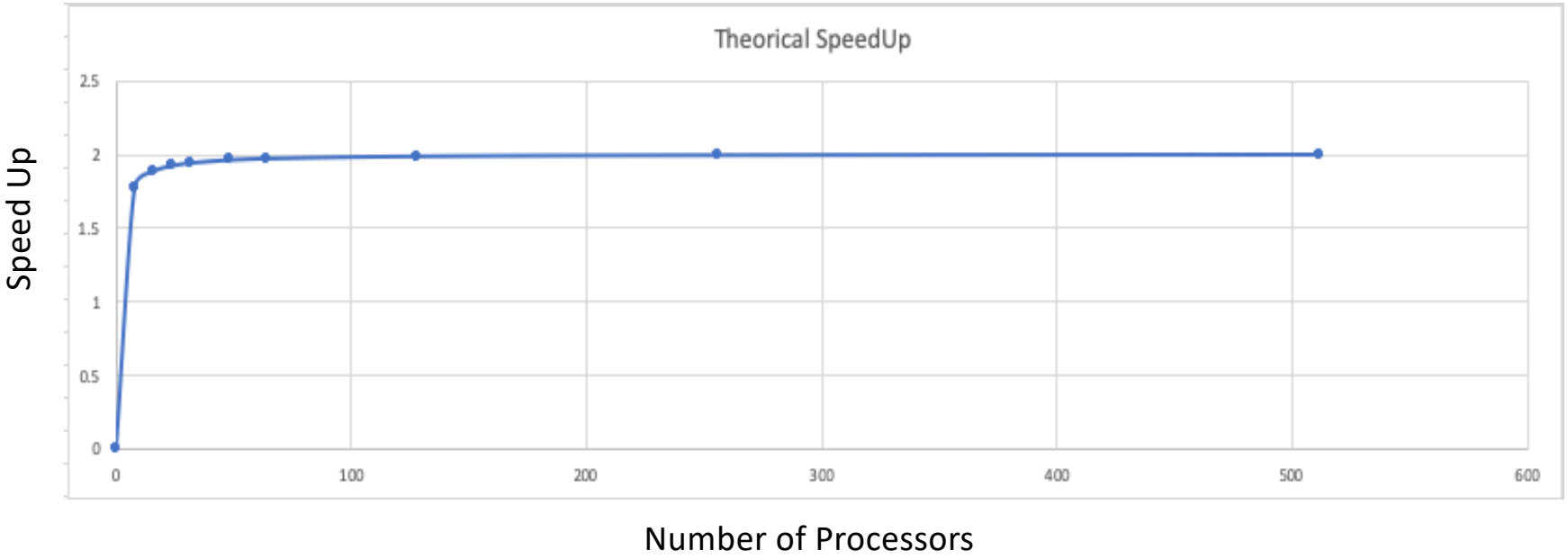
N -> number of processors

Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Range
33.4%	320.813 s	1	320.813 s	320.813 s	320.813 s	320.813 s	0 ns	init
33.3%	320.719 s	1	320.719 s	320.719 s	320.719 s	320.719 s	0 ns	while
33.3%	320.280 s	9000	35.587 ms	35.124 ms	33.589 ms	44.890 ms	1.361 ms	perform_timestep

https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/?subscribe=error#subscribe-blog-blog_subscription-3

GPU Computing

Expected speedup



The expected speed up of MiniWeather application is limited to 2x no matter how many processors (N) are used to run the parallelized proportion.

GPU Computing

Is the problem Memory Bound or Compute Bound?

The compute-to-global-memory-access-ratio allow us determine if the application in memory bound or compute bound

compute-to-global-memory-access-ratio = # floating point operations / memory access

The dominant kernel in the Miniweather application parallelized fraction is :

- **compute_tendencies_x**

```
//Fourth-order-accurate interpolation of the state
vals[ll] = -stencil[0] / 12 + 7 * stencil[1] / 12 + 7 * stencil[2] / 12 - stencil[3] / 12;
//First-order-accurate interpolation of the third spatial derivative of the state (for artificial viscosity)
d3_vals[ll] = -stencil[0] + 3 * stencil[1] - 3 * stencil[2] + stencil[3];
```

stencil[] is a local array variable but it's value is first loaded from memory before usage

compute-to-global-memory-access-ratio for **compute_tendencies_x** kernel is 10/(4.0 * 2) -> 1.25

GPU Computing (V100)

Is the problem Memory Bound or Compute Bound?

Miniweather **compute_tendencies_x**

Compute-Global-Memory-Access-Ratio: 1.25

Applications uses Double precision

NVIDIA GPU V100

Global Memory Bandwidth: 900 GB/s

Double Precision: 7 TFLOPS

Peak Compute-To-Global-Memory-Access-Ratio : 7.7

With a Compute-To-Global-Memory-Access-Ratio of 1.25 the kernel **compute_tendencies_x** throughput will be limited on how fast operands are fetched from memory. This make the dominant kernel **compute_tendencies_x /**application memory-bound. In fact the kernel when ported to GPU V100 will not achieve more than 112.5 GFLOPS.

Multicore Parallelization Results

Application	target	hardware	Time (s)	Speedup
Serial		intel	307.842808	
OpenACC	multicore	Intel 8 cores	82.222548	3.74

Using default values
nx_glob, nz_glob: 400 200 dx,dz: 50.000000 50.000000
dt: 0.166667

Flags to Compile the Code

```
# Copyright (c) 2020 NVIDIA Corporation.  All rights reserved.
CC := pgc++
CFLAGS := -O3 -w
# ACCFLAGS := -ta=tesla -Minfo=accel
# ACCFLAGS := -ta=tesla:managed -Minfo=accel
ACCFLAGS := -ta=multicore -Minfo=accel
LDFLAGS :=  -lnetcdf -ldl
NVTXLIB := -I/opt/nvidia/hpc_sdk/Linux_x86_64/20.9/cuda/11.0/include

miniWeather: miniWeather_openacc.cpp
	${CC} ${CFLAGS} ${ACCFLAGS} -o miniWeather miniWeather_multicore.cpp ${NVTXLIB} ${LDFLAGS}

clean:
	rm -f *.o miniWeather
```

OpenACC directives / optimization

Application Code Sample

```
420     #pragma acc parallel loop private(indt, indf1, indf2, inds)
421     for (ll = 0; ll < NUM_VARS; ll++)
422     {
423         #pragma acc loop vector collapse(2)
424         for (k = 0; k < nz; k++)
425         {
426             for (i = 0; i < nx; i++)
427             {
428                 indt = ll * nz * nx + k * nx + i;
429                 indf1 = ll * (nz + 1) * (nx + 1) + (k) * (nx + 1) + i;
430                 indf2 = ll * (nz + 1) * (nx + 1) + (k + 1) * (nx + 1) + i;
431                 tend[indt] = -(flux[indf2] - flux[indf1]) / dz;
432                 if (ll == ID_WMOM)
433                 {
434                     inds = ID_DENS * (nz + 2 * hs) * (nx + 2 * hs) + (k + hs) * (nx + 2 * hs) + i + hs;
435                     tend[indt] = tend[indt] - state[inds] * grav;
436                 }
437             }
438         }
439     }
```

OpenACC directives / optimization

Application Code Sample Description

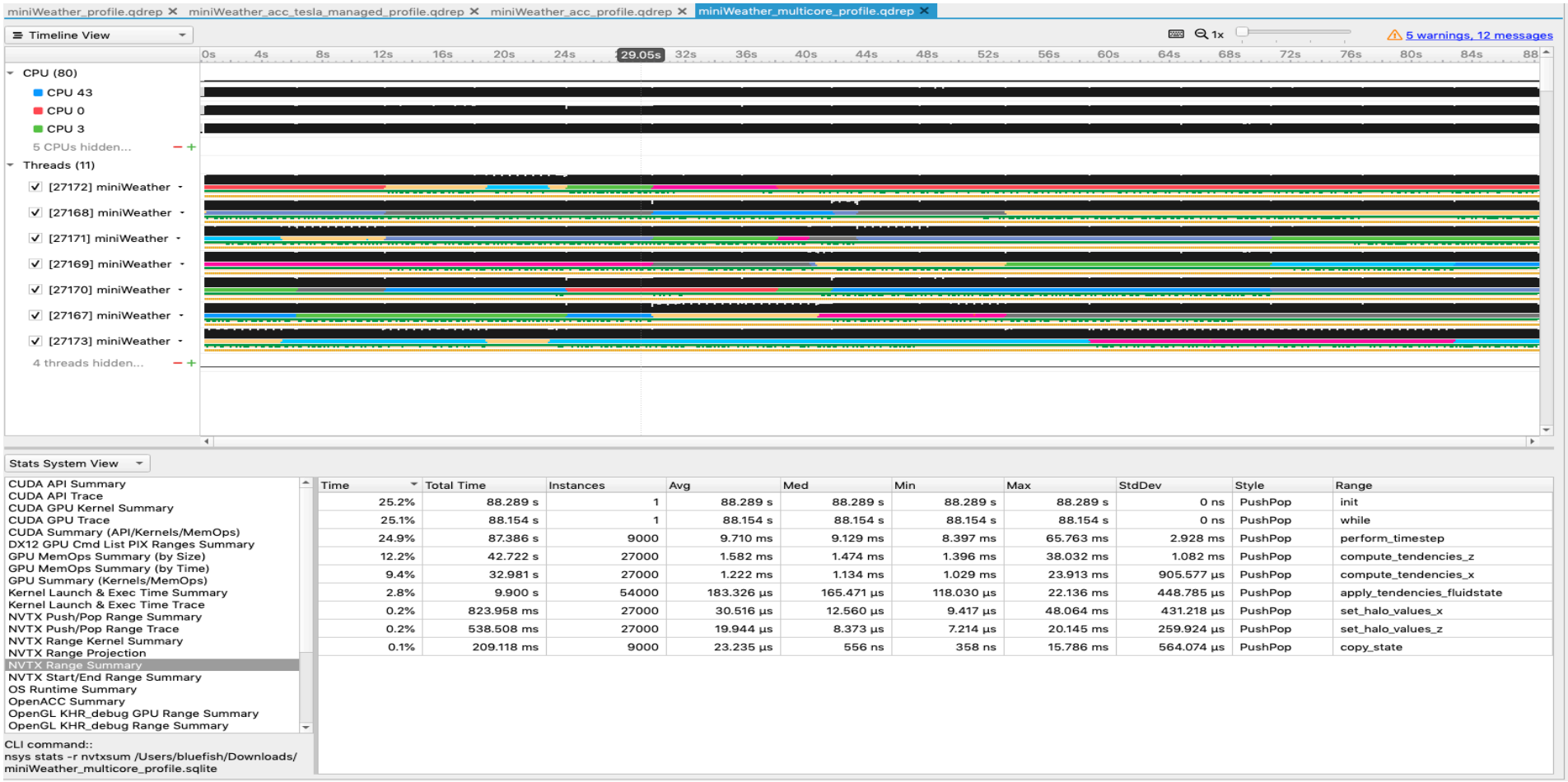
External loop in line 421 is parallelized and distributed among a gang of threads. This is done using the openacc directive #pragma acc parallel loop.

Variables that could be potentially shared by multiples threads are declared private and local to each thread in the gang. Line 420

Inner loops at line 424 and 426 are collapsed in one loop and vectorized.

OpenACC directives / optimization

Multicore nsys system profiler output and application timeline



Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Style	Range
25.2%	88.289 s	1	88.289 s	88.289 s	88.289 s	88.289 s	0 ns	PushPop	init
25.1%	88.154 s	1	88.154 s	88.154 s	88.154 s	88.154 s	0 ns	PushPop	while
24.9%	87.386 s	9000	9.710 ms	9.129 ms	8.397 ms	65.763 ms	2.928 ms	PushPop	perform_timestep
12.2%	42.722 s	27000	1.582 ms	1.474 ms	1.396 ms	38.032 ms	1.082 ms	PushPop	compute_tendencies_z
9.4%	32.981 s	27000	1.222 ms	1.134 ms	1.029 ms	23.913 ms	905.577 µs	PushPop	compute_tendencies_x
2.8%	9.900 s	54000	183.326 µs	165.471 µs	118.030 µs	22.136 ms	448.785 µs	PushPop	apply_tendencies_fluidstate
0.2%	823.958 ms	27000	30.516 µs	12.560 µs	9.417 µs	48.064 ms	431.218 µs	PushPop	set_halo_values_x
0.2%	538.508 ms	27000	19.944 µs	8.373 µs	7.214 µs	20.145 ms	259.924 µs	PushPop	set_halo_values_z
0.1%	209.118 ms	9000	23.235 µs	556 ns	358 ns	15.786 ms	564.074 µs	PushPop	copy_state

GPU Parallelization Results

Application	target	hardware	Time (s)	Speedup
Serial		intel	307.842808	
OpenACC	multicore	Intel 8 cores	82.222548	3.74
OpenACC	tesla:managed	GPU V100	11.630973	26.46
OpenACC	tesla	GPU V100	12.010639	25.63

Using default values
nx_glob, nz_glob: 400 200 dx,dz: 50.000000 50.000000
dt: 0.166667

Flags to Compile the Code

```
# Copyright (c) 2020 NVIDIA Corporation.  All rights reserved.
CC := pgc++
CFLAGS := -O3 -w
ACCFLAGS := -ta=tesla -Minfo=accel
# ACCFLAGS := -ta=tesla:managed -Minfo=accel
# ACCFLAGS := -ta=multicore -Minfo=accel
LDFLAGS :=  -lnetcdf -ldl
NVTXLIB := -I/opt/nvidia/hpc_sdk/Linux_x86_64/20.9/cuda/11.0/include

miniWeather: miniWeather_openacc.cpp
	${CC} ${CFLAGS} ${ACCFLAGS} -o miniWeather miniWeather_openacc.cpp ${NVTXLIB} ${LDFLAGS}

clean:
	rm -f *.o miniWeather
```

OpenACC directives / optimization

Data Optimization directives to minimize data transfer between host and device when not using managed memory. CUDA kernels don’t need to wait for data transfer during their execution. Present clause using must of the time.

Parallel Loop optimizations and vector size adjusted to 32.

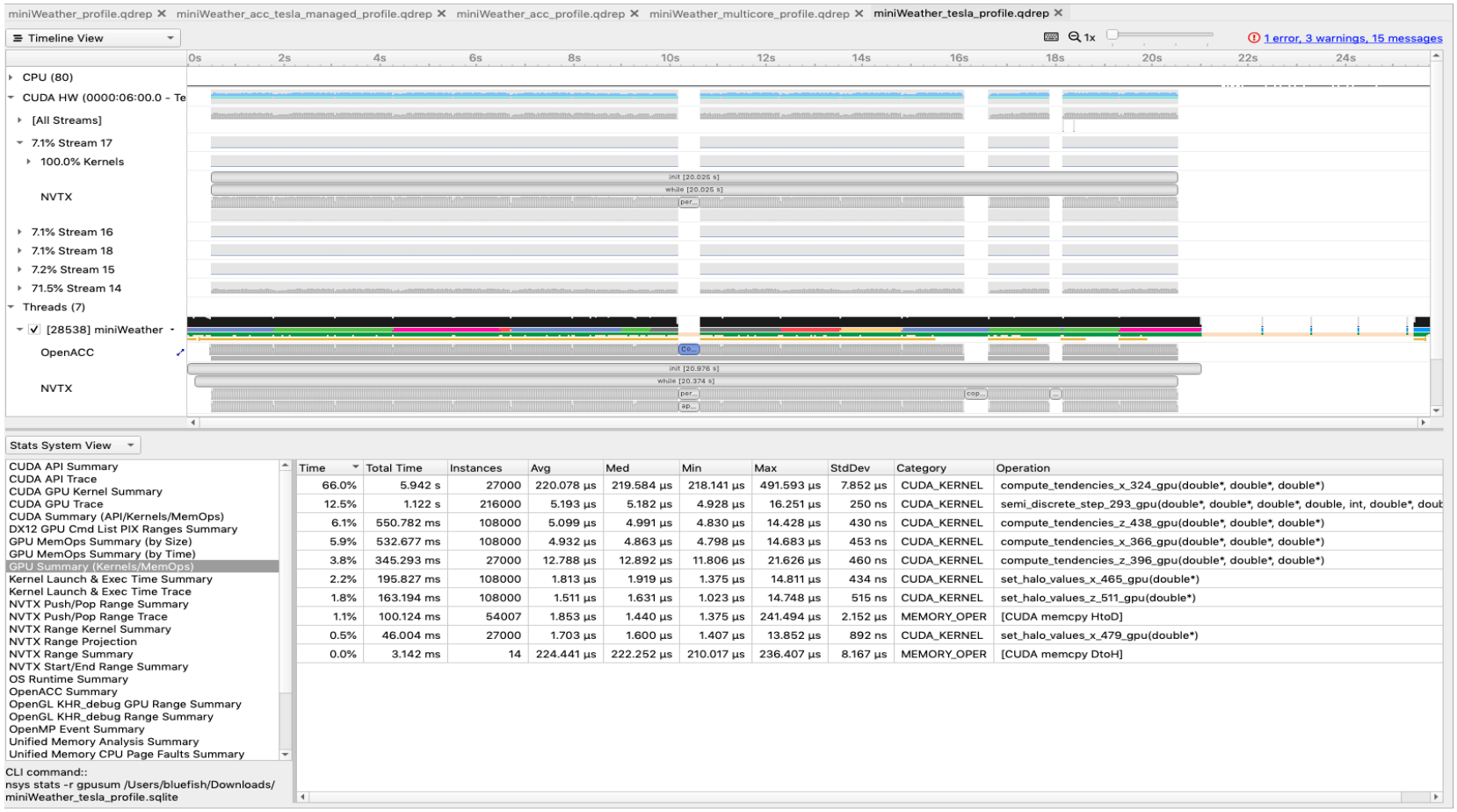
Variables that could potentially be shared among multiples threads are declared private and local to each thread in the gang.

Inner loops are collapsed in one loop and vectorized.

Independent Loop iterations are parallelized using asynchronous cuda streams.

OpenACC directives / optimization

Application Timeline and GPU summary (Kernels/Memory).



Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Category	Operation
66.0%	5.942 s	27000	220.078 µs	219.584 µs	218.141 µs	491.593 µs	7.852 µs	CUDA_KERNEL	compute_tendencies_x_324_gpu(double*, double*, double*)
12.5%	1.122 s	216000	5.193 µs	5.182 µs	4.928 µs	16.251 µs	250 ns	CUDA_KERNEL	semi_discrete_step_293_gpu(double*, double*, double*, double, int, double*, doub
6.1%	550.782 ms	108000	5.099 µs	4.991 µs	4.830 µs	14.428 µs	430 ns	CUDA_KERNEL	compute_tendencies_z_438_gpu(double*, double*, double*)
5.9%	532.677 ms	108000	4.932 µs	4.863 µs	4.798 µs	14.683 µs	453 ns	CUDA_KERNEL	compute_tendencies_x_366_gpu(double*, double*, double*)
3.8%	345.293 ms	27000	12.788 µs	12.892 µs	11.806 µs	21.626 µs	460 ns	CUDA_KERNEL	compute_tendencies_z_396_gpu(double*, double*, double*)
2.2%	195.827 ms	108000	1.813 µs	1.919 µs	1.375 µs	14.811 µs	434 ns	CUDA_KERNEL	set_halo_values_x_465_gpu(double*)
1.8%	163.194 ms	108000	1.511 µs	1.631 µs	1.023 µs	14.748 µs	515 ns	CUDA_KERNEL	set_halo_values_z_511_gpu(double*)
1.1%	100.124 ms	54007	1.853 µs	1.440 µs	1.375 µs	241.494 µs	2.152 µs	MEMORY_OPER	[CUDA memcpy HtoD]
0.5%	46.004 ms	27000	1.703 µs	1.600 µs	1.407 µs	13.852 µs	892 ns	CUDA_KERNEL	set_halo_values_x_479_gpu(double*)
0.0%	3.142 ms	14	224.441 µs	222.252 µs	210.017 µs	236.407 µs	8.167 µs	MEMORY_OPER	[CUDA memcpy DtoH]

OpenACC directives / optimization

Private variables definition per thread to avoid race conditions

Data already on device (present)

```
    //Apply the tendencies to the fluid state
    nvtxRangePushA("apply_tendencies_fluidstate");

    for (ll = 0; ll < NUM_VARS; ll++)
    {
        #pragma acc parallel loop collapse(2) private(indt, inds) present( state_init[0:(nx + 2 * hs) * (nz + 2 * hs) * NUM_VARS],
state_out[0:(nx + 2 * hs) * (nz + 2 * hs) * NUM_VARS], tend[0:nx * nz * NUM_VARS ])  vector_length(32) async(ll)
        for (k = 0; k < nz; k++)
        {
            for (i = 0; i < nx; i++)
            {
                inds = ll * (nz + 2 * hs) * (nx + 2 * hs) + (k + hs) * (nx + 2 * hs) + i + hs;
                indt = ll * nz * nx + k * nx + i;
                state_out[inds] = state_init[inds] + dt * tend[indt];
            }
        }
    }
    #pragma acc wait

    nvtxRangePop();
}
```

Independent loop iterations are parallelized using asynchronous CUDA streams.

Parallel loop optimization collapse and vectorization

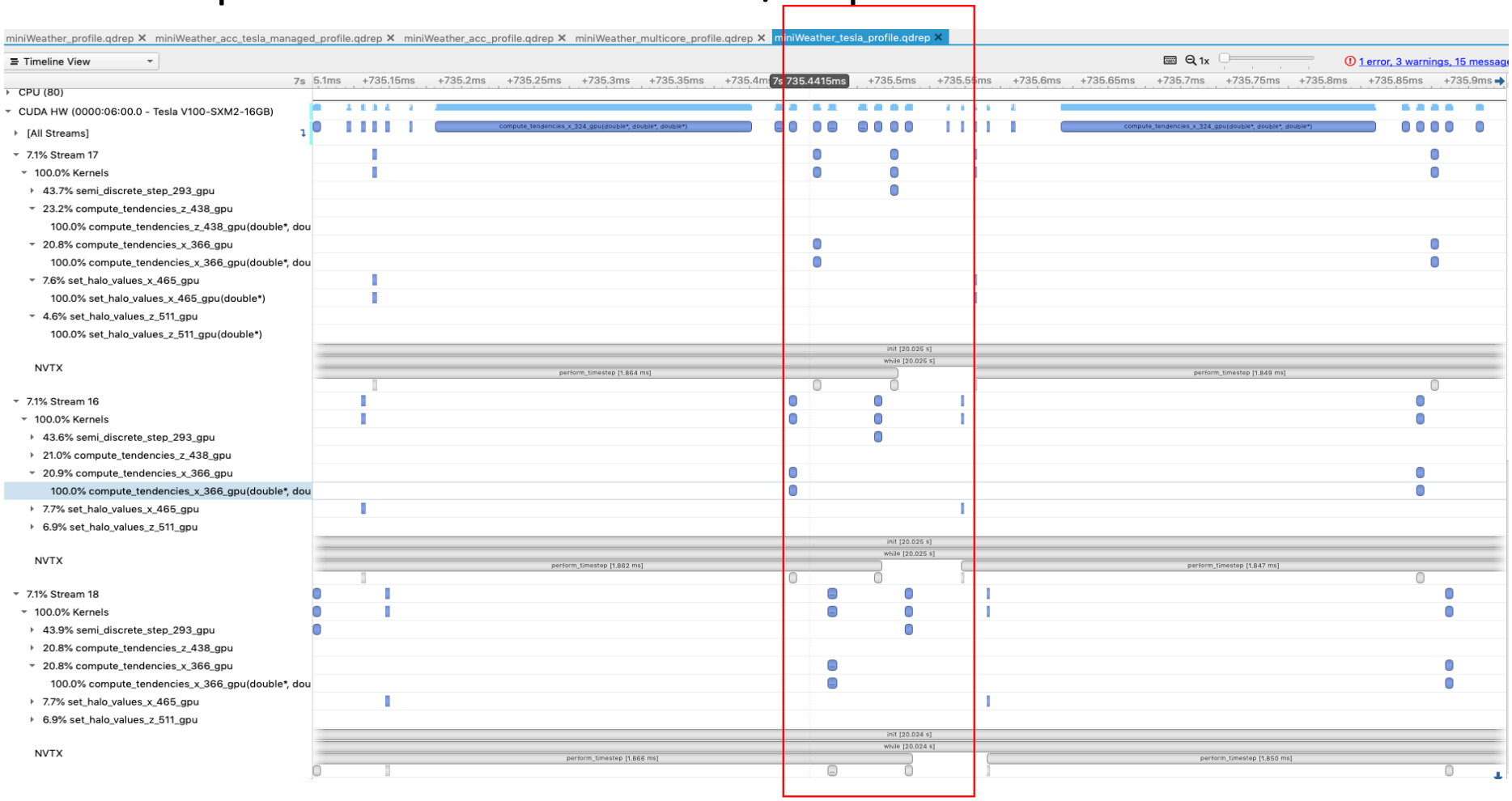
OpenACC directives / optimization

```
In [9]: !cd ../source_code && make clean && make
```

```
rm -f *.o miniWeather
pgc++ -O3 -w -ta=tesla -Minfo=accel  -o miniWeather miniWeather_openacc.cpp -I/opt/nvidia/hpc_sdk/Linux_x86_64/20.9/c
uda/11.0/include -lnetcdf -ldl
main:
    167, Generating copyin(hy_pressure_int[:nz+1],state[:((nz+4)*(nx+4))*4],tend[:(nz*nx)*4],state_tmp[:((nz+4)*(nx+
4))*4],hy_dens_cell[:nz+4],hy_dens_theta_cell[:nz+4],flux[:((nz+1)*(nx+1))*4]) [if not already present]
    196, Generating update self(state[:((nz+4)*(nx+4))*4])
semi_discrete_step(double *, double *, double *, double, int, double *, double *):
    293, Generating present(state_init[:((nz+4)*(nx+4))*4],tend[:(nz*nx)*4],state_out[:((nz+4)*(nx+4))*4])
         Generating Tesla code
        293, #pragma acc loop gang, vector(32) collapse(2) /* blockIdx.x threadIdx.x */
        295,   /* blockIdx.x threadIdx.x collapsed */
compute_tendencies_x(double *, double *, double *):
```

Example of Compiler messages.

OpenACC directives / optimization



Kernel compute_tendencies_x_366_gpu is launched across multiples streams

OpenACC directives / optimization

Data Optimization directives to minimize data transfer between host and device when not using managed memory. Creation of a data region

```
    nvtxRangePushA("while");

    #pragma acc data copyin(state[0:(nx + 2 * hs) * (nz + 2 * hs) * NUM_VARS ], state_tmp[0:(nx + 2 * hs) * (nz + 2 * hs) *
NUM_VARS ], flux[0:(nx + 1) * (nz + 1) * NUM_VARS], tend[0:nx * nz * NUM_VARS ], hy_dens_theta_cell[0:nz + 2 * hs],
hy_pressure_int[0:nz + 1], hy_dens_cell[0:nz + 2 * hs])
    {
        while (etime < sim_time)
        {
            //If the time step leads to exceeding the simulation time, shorten it for the last step
            if (etime + dt > sim_time)
            {
                dt = sim_time - etime;
            }

            //Perform a single time step
            nvtxRangePushA("perform_timestep");
            perform_timestep(state, state_tmp, flux, tend, dt);
            nvtxRangePop();

            //Inform the user

            printf("Elapsed Time: %lf / %lf\n", etime, sim_time);

            //Update the elapsed time and output counter
            etime = etime + dt;
            output_counter = output_counter + dt;
            //If it's time for output, reset the counter, and do output

            nvtxRangePushA("copy_state");
            if (output_counter >= output_freq)
            {
                output_counter = output_counter - output_freq;

                #pragma acc update self(state[: (nx+2*hs) * (nz+2*hs) * NUM_VARS ])
                output(state, etime);
            }
            nvtxRangePop();
        }
    }
```

Must of data is copied to the gpu only once

Dominant DtoH memory transfers

OpenACC directives / optimization

Data Optimization directives to minimize data transfer between host and device when not using managed memory.

Time	Total Time	Count	Avg	Med	Min	Max	StdDev	Operation
97.0%	100.124 ms	54007	1.853 µs	1.440 µs	1.375 µs	241.494 µs	2.152 µs	[CUDA memcpy HtoD]
3.0%	3.142 ms	14	224.441 µs	222.252 µs	210.017 µs	236.407 µs	8.167 µs	[CUDA memcpy DtoH]

Total	Count	Avg	Med	Min	Max	StdDev	Operation
92.75 MiB	54007	1.76 KiB	1.57 KiB	1.57 KiB	2.52 MiB	21.87 KiB	[CUDA memcpy HtoD]
35.21 MiB	14	2.52 MiB	2.52 MiB	2.52 MiB	2.52 MiB	0 B	[CUDA memcpy DtoH]

Must of the memory transfer occurred when data generated in the device was copied to variable state

Code available at:

https://github.com/mahsanchez/mw_openacc/blob/main/miniweather_openacc.cpp

https://github.com/mahsanchez/mw_openacc/blob/main/miniweather_multicore.cpp