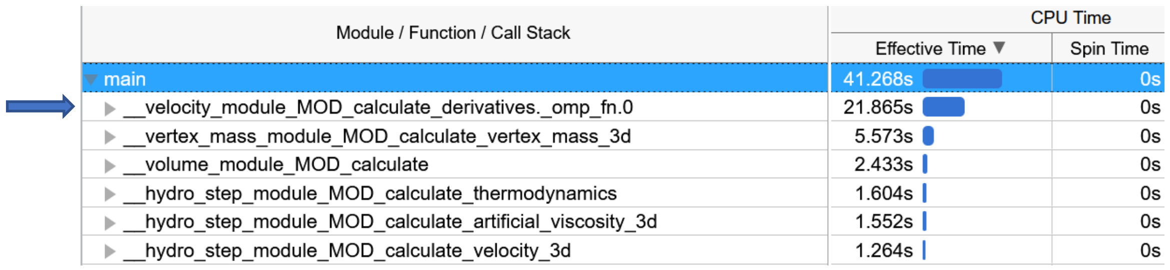
What did you learn from profiling?

Using function profiling, I was able to accurately identify the areas of the code that are consuming the most CPU time, and the precise time spent in each function. Running Vtune on Hotspot/Threading mode helped me identify the "time consuming" functions, since Vtune hotspot is a tool for identifying performance bottlenecks, it viewed and ranked the functions in the order they impact the performance.

In Vtune the function "calculate_derivatives" was on top of the list with almost 40% of the cpu time, with some other functions.
Therefore the "calculate_derivatives" function was my main focus, and I parallelized the three nested loops in this function (using openMp).
then moved on to the other functions, also parallelizing the main loops there.

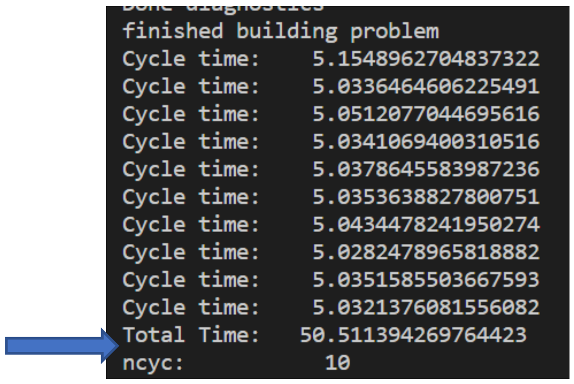


Module / Function / Call Stack	CPU Time	
	Effective Time ▼	Spin Time
main	41.268s	0s
__velocity_module_MOD_calculate_derivatives._omp_fn.0	21.865s	0s
__vertex_mass_module_MOD_calculate_vertex_mass_3d	5.573s	0s
__volume_module_MOD_calculate	2.433s	0s
__hydro_step_module_MOD_calculate_thermodynamics	1.604s	0s
__hydro_step_module_MOD_calculate_artificial_viscosity_3d	1.552s	0s
__hydro_step_module_MOD_calculate_velocity_3d	1.264s	0s

This is a screenshot in the threading mode

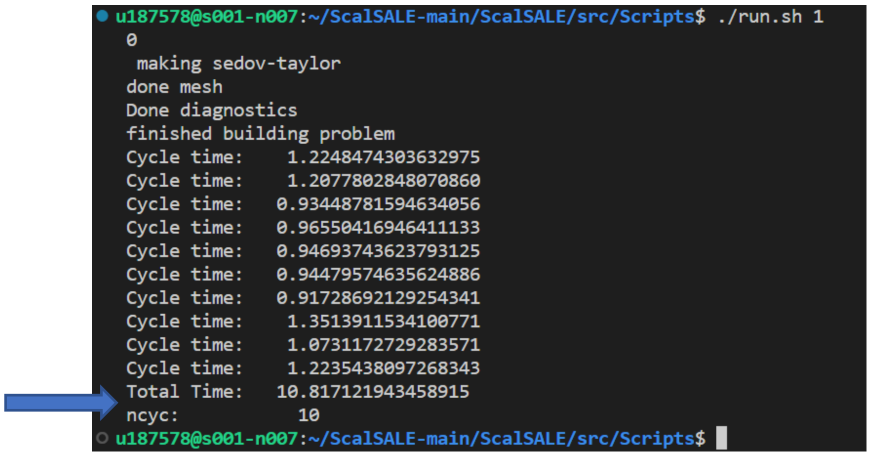
Achievements:

When trying to run the code for the first time, I achieved an average of 12s for one cycle.
Therefor the best option was to add the O3 flag, which is the highest level of optimization, in order to instruct the compiler to optimize code for performance.
in the screenshot below shows the cycle times after adding the O3 flag:



```
finished building problem
Cycle time:    5.1548962704837322
Cycle time:    5.0336464606225491
Cycle time:    5.0512077044695616
Cycle time:    5.0341069400310516
Cycle time:    5.0378645583987236
Cycle time:    5.0353638827800751
Cycle time:    5.0434478241950274
Cycle time:    5.0282478965818882
Cycle time:    5.0351585503667593
Cycle time:    5.0321376081556082
Total Time:   50.511394269764423
ncyc:            10
```

I used the OpenMp in order to parallelize the main loops in the "time-consuming" functions, and the result:

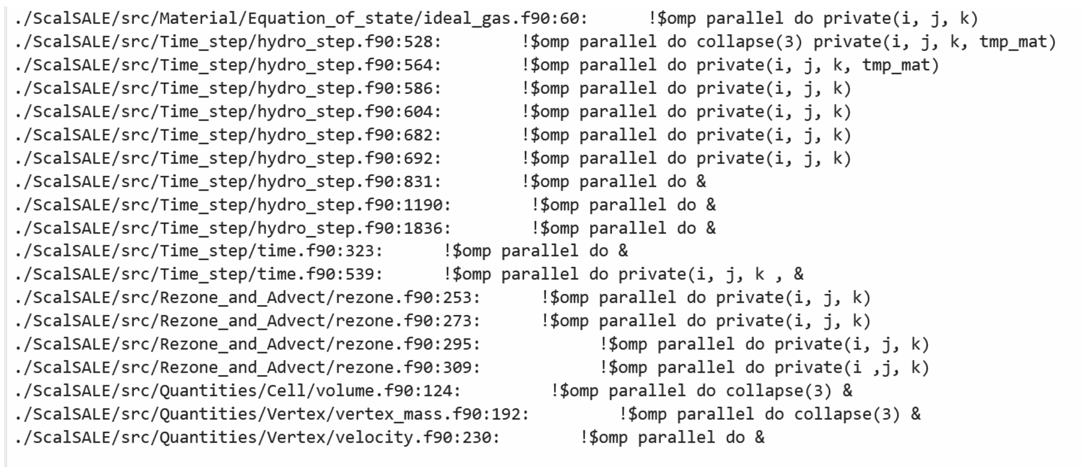


```
u187578@s001-n007:~/ScalSALE-main/ScalSALE/src/Scripts$ ./run.sh 1
 0
  making sedov-taylor
 done mesh
 Done diagnostics
 finished building problem
 Cycle time:    1.2248474303632975
 Cycle time:    1.2077802848070860
 Cycle time:   0.93448781594634056
 Cycle time:   0.96550416946411133
 Cycle time:   0.94693743623793125
 Cycle time:   0.94479574635624886
 Cycle time:   0.91728692129254341
 Cycle time:    1.3513911534100771
 Cycle time:    1.0731172729283571
 Cycle time:    1.2235438097268343
 Total Time:   10.817121943458915
 ncyc:           10
u187578@s001-n007:~/ScalSALE-main/ScalSALE/src/Scripts$
```

As I mentioned before in order to parallelize the code I first identified the section of code that was suitable for parallelization using Vtune. Which was in the “calculate_dirivatives” function(3 nested loops), and other functions that also had a high run time.

In order to parallelize the functions, including the large loops, I included the OpenMp pargams we learned in class, using “#omp parallel do” before the loop and “#omp end parallel do”. Also, I had to set the private variables in the loops.

After each change, I compiled the code again and examined the results using vtune in order to achieve a better run time and to further optimize the code.



```
./ScalSALE/src/Material/Equation_of_state/ideal_gas.f90:60:      !$omp parallel do private(i, j, k)
./ScalSALE/src/Time_step/hydro_step.f90:528:        !$omp parallel do collapse(3) private(i, j, k, tmp_mat)
./ScalSALE/src/Time_step/hydro_step.f90:564:        !$omp parallel do private(i, j, k, tmp_mat)
./ScalSALE/src/Time_step/hydro_step.f90:586:        !$omp parallel do private(i, j, k)
./ScalSALE/src/Time_step/hydro_step.f90:604:        !$omp parallel do private(i, j, k)
./ScalSALE/src/Time_step/hydro_step.f90:682:        !$omp parallel do private(i, j, k)
./ScalSALE/src/Time_step/hydro_step.f90:692:        !$omp parallel do private(i, j, k)
./ScalSALE/src/Time_step/hydro_step.f90:831:        !$omp parallel do &
./ScalSALE/src/Time_step/hydro_step.f90:1190:        !$omp parallel do &
./ScalSALE/src/Time_step/hydro_step.f90:1836:        !$omp parallel do &
./ScalSALE/src/Time_step/time.f90:323:      !$omp parallel do &
./ScalSALE/src/Time_step/time.f90:539:      !$omp parallel do private(i, j, k , &
./ScalSALE/src/Rezone_and_Advect/rezone.f90:253:      !$omp parallel do private(i, j, k)
./ScalSALE/src/Rezone_and_Advect/rezone.f90:273:      !$omp parallel do private(i, j, k)
./ScalSALE/src/Rezone_and_Advect/rezone.f90:295:            !$omp parallel do private(i, j, k)
./ScalSALE/src/Rezone_and_Advect/rezone.f90:309:            !$omp parallel do private(i ,j, k)
./ScalSALE/src/Quantities/Cell/volume.f90:124:         !$omp parallel do collapse(3) &
./ScalSALE/src/Quantities/Vertex/vertex_mass.f90:192:         !$omp parallel do collapse(3) &
./ScalSALE/src/Quantities/Vertex/velocity.f90:230:        !$omp parallel do &
```

The compiler I used:

After a research I found that gfortran is a powerful and flexible option for parallelizing code, offering extensive compatibility.
I used the gfortran compiler and achieved a relatively good run time.

Note: I tried running the code using the ifort compiler (I changed the compiler using the way you showed in the last one minute in the video, also did the setvars command after changing), but each time I tried ./clean.sh it didn't complete, therefor I couldn't run the code using the ifort compiler and I couldn't compare the results between different compilers:(

GPU:

When examining the program I can see that it has a main loop with a lot of dependencies between iterations. So, if we want to offload data to the gpu we have to offload other dependent data too, which is not logical.

To resolve the issue, I tried the option of approaching the primary loop, I did map the relevant data structures before the primary loop – problem.f90, and updated them in every iteration, and offload the data from CPU to GPU before the "Calculate_derivatives" function and update data from GPU to CPU afterward.
The offloading implementation for this was divided into two files, namely problem.f90(it has the primary loop) and velocity.f90(Calculate derivatives), and was contained within the ScalSALE-gpu directory. In problem.f90, I added the relevant pragma and offloading function before the primary loop, and in velocity.f90, before and after the loop for Calculate_derivatives, I added the necessary functions to send the data, get it back, and update it.