What did you learn from Vtune profiling:

I've used Vtune Threading option which is a feature that allows you to analyze the performance of a multi-threaded application. When analyzing a multi-threaded application, VTune Threading enables you to gather performance data from each thread separately and then aggregate the results to get a more comprehensive view of the application's performance.

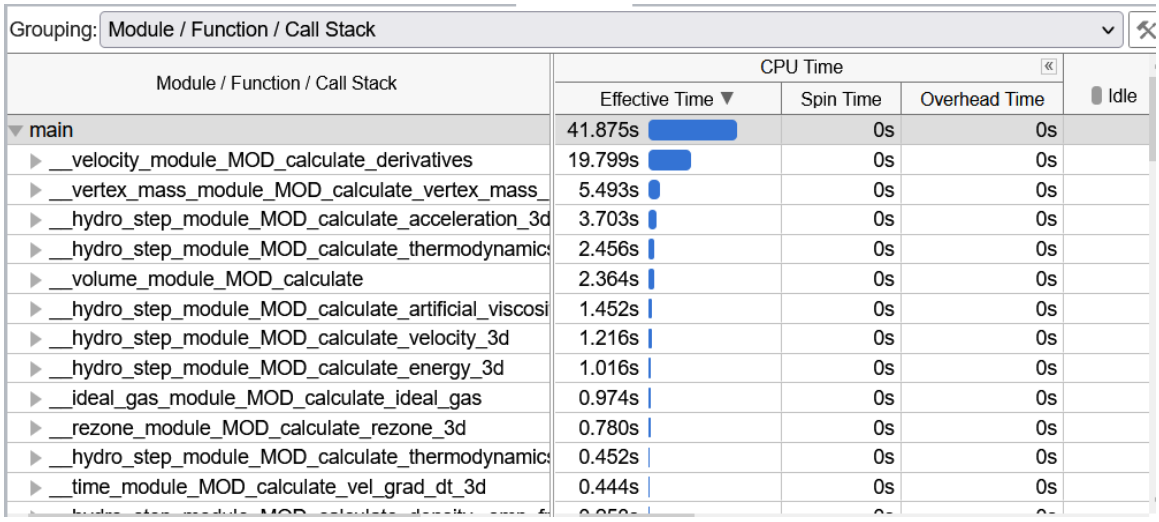
With this option, I've found places with hotspots, which are functions on the project, that were using the most time in the application and the exact duration spent in each function.

This information gave me the performance bottleneck in the code and where should I focus on.

From the profiler I've known that the best place to perform optimization was calculate_derivatives, and a couple more functions which ill mention later in the report.

Each one of these hotspots that were found, I dealt with her with the openmp pragmas of fortran.

Picture from the Vtune showing the functions and the time each one spent on the run: (We can see that calculate derivatives took the most time, so we started by handling it first)



Grouping: Module / Function / Call Stack

Module / Function / Call Stack	CPU Time			Idle
	Effective Time ▼	Spin Time	Overhead Time	
main	41.875s	0s	0s	
__velocity_module_MOD_calculate_derivatives	19.799s	0s	0s	
__vertex_mass_module_MOD_calculate_vertex_mass_	5.493s	0s	0s	
__hydro_step_module_MOD_calculate_acceleration_3d	3.703s	0s	0s	
__hydro_step_module_MOD_calculate_thermodynamic	2.456s	0s	0s	
__volume_module_MOD_calculate	2.364s	0s	0s	
__hydro_step_module_MOD_calculate_artificial_viscosi	1.452s	0s	0s	
__hydro_step_module_MOD_calculate_velocity_3d	1.216s	0s	0s	
__hydro_step_module_MOD_calculate_energy_3d	1.016s	0s	0s	
__ideal_gas_module_MOD_calculate_ideal_gas	0.974s	0s	0s	
__rezone_module_MOD_calculate_rezone_3d	0.780s	0s	0s	
__hydro_step_module_MOD_calculate_thermodynamic	0.452s	0s	0s	
__time_module_MOD_calculate_vel_grad_dt_3d	0.444s	0s	0s	

(I have to mention that the Vtune profiler didn't work as I expected since I have the problem of seeing the function as there location on the code, assembly location, I informed Yehonathan about the issue.)

My achievements:

First thing I did was running the project on serial mode without any optimization, to check the amount of time the running would take, and I've got this:

(Sorry for the print("hi") that were in the middle)

```
Application Output
  0
   making sedov-taylor
  done mesh
  Done diagnostics
  finished building problem
  hi
  Cycle time:    12.693916454911232
  hi
  Cycle time:    13.024112544953823
  hi
  Cycle time:    12.779446320608258
  hi
  Cycle time:    12.647634431719780
  hi
  Cycle time:    12.583921505138278
  hi
  Cycle time:    12.511623922735453
  hi
  Cycle time:    12.472357081249356
  hi
  Cycle time:    12.476540770381689
  hi
  Cycle time:    12.430432088673115
  hi
  Cycle time:    12.474860582500696
  Total Time:   126.13239316642284
  ncyc:          10
```

After that, I've decided to use -o3 flag on the cmakelist, which is a compiler optimization flag that enables the highest level of optimization in the compiler.

And I've got the result of **50 seconds** total time:

```
u187554@s001-n002:~/helperScal/ScalSALE-main/src/Scripts$ ./run.sh 1
 0
  making sedov-taylor
 done mesh
 Done diagnostics
 finished building problem
 Cycle time:     5.1485561244189739
 Cycle time:     5.0343927331268787
 Cycle time:     5.0332871153950691
 Cycle time:     5.0297751836478710
 Cycle time:     5.0369162596762180
 Cycle time:     5.0265680160373449
 Cycle time:     5.0231137685477734
 Cycle time:     5.0301657766103745
 Cycle time:     5.0321176201105118
 Cycle time:     5.0258645042777061
 Total Time:    50.449725050479174
 ncyc:            10
u187554@s001-n002:~/helperScal/ScalSALE-main/src/Scripts$
```

Last thing, was doing the optimizations using the open-mp pragmas of fortran while keeping -o3 flag, and I've reached the total time of 10 second approx.

```
u187554@s001-n061:~/ScalSALE-main/src/Scripts$ ./run.sh 1
 0
  making sedov-taylor
 done mesh
 Done diagnostics
 finished building problem
 Cycle time:    1.3682540468871593
 Cycle time:    1.0553217660635710
 Cycle time:    1.0775032900273800
 Cycle time:    1.1221211627125740
 Cycle time:   0.94899427518248558
 Cycle time:    1.0067895520478487
 Cycle time:   0.91284256055951118
 Cycle time:   0.91957117430865765
 Cycle time:   0.89623875729739666
 Cycle time:   0.94888567738234997
 Total Time:   10.332222618162632
 ncyc:           10
u187554@s001-n061:~/ScalSALE-main/src/Scripts$
```

The places where I used OpenMp pragmas:

To parallelize a specific part of my application using OpenMP, I first identified the section of code suitable for parallelization, which happened to be calculate_derivatives in velocity.f90, and so on, based on the previous identification using Vtune.

After parallelizing first function (in velocity.f90), I recompiled the code, and ran Vtune again to check the accomplishments so far, and checked again what's the next place to parallelize.
I kept parallelizing function and compiling and checking again, until I saw no further optimization comes out of parallelizing more function, and there I stopped and that's were the results:

We can see in the picture the places where the pragma "omp parallel do" was added, in each place of them there was a function that the vtune gave us, indicating that it ran a good amount of time and could be parallelized.

```
./Material/Equation_of_state/ideal_gas.f90:60:      !$omp parallel do private(k,j,i)
./Time_step/hydro_step.f90:527:        !$omp parallel do collapse(3) private(k,j,i,tmp_mat)
./Time_step/hydro_step.f90:561:        !$omp parallel do private(k,j,i,tmp_mat)
./Time_step/hydro_step.f90:583:        !$omp parallel do private(k,j,i)
./Time_step/hydro_step.f90:600:        !$omp parallel do private(k,j,i)
./Time_step/hydro_step.f90:675:        !$omp parallel do  private(k,j,i)
./Time_step/hydro_step.f90:684:        !$omp parallel do private(k,j,i)
./Time_step/hydro_step.f90:821:        !$omp parallel do  &
./Time_step/hydro_step.f90:1177:        !$omp parallel do &
./Time_step/hydro_step.f90:1826:        !$omp parallel do &
./Time_step/.ipynb_checkpoints/hydro_step-checkpoint.f90:563:        !$omp parallel do simd collapse(1) schedule(simd:static) private(j,i,tmp_mat)
./Time_step/.ipynb_checkpoints/hydro_step-checkpoint.f90:585:        !$omp parallel do simd collapse(1) schedule(simd:static) private(j,i)
./Time_step/.ipynb_checkpoints/hydro_step-checkpoint.f90:601:        !$omp parallel do simd collapse(1) schedule(simd:static) private(j,i)
./Time_step/.ipynb_checkpoints/hydro_step-checkpoint.f90:680:        !$omp parallel do simd collapse(1) schedule(simd:static) private(j,i)
./Time_step/.ipynb_checkpoints/hydro_step-checkpoint.f90:824:        !$omp parallel do simd collapse(1) schedule(simd:static)
./Time_step/time.f90:323:      !$omp parallel do private(k,j,i,vel_diff_coor_diff_i,vel_diff_coor_diff_j,vel_diff_coor_diff_k,vel_grad)
./Time_step/time.f90:534:      !$omp parallel do private(k,j,i,vel_diff_x,vel_diff_y,vel_diff_z,len_sq, &
./Rezone_and_Advect/rezone.f90:252:      !$omp parallel do private(k,j,i)
./Rezone_and_Advect/rezone.f90:270:      !$omp parallel do private(k,j,i)
./Rezone_and_Advect/rezone.f90:292:           !$omp parallel do private(k,j,i)
./Rezone_and_Advect/rezone.f90:304:           !$omp parallel do private(k,j,i)
./Quantities/Cell/volume.f90:122:      !$omp parallel do collapse(3) &
./Quantities/Vertex/vertex_mass.f90:191:        !$omp parallel do private(k,j,i,t,kk,jj,ii,i1,i2,i3,j1,j2,j3,k1,k2,k3)
./Quantities/Vertex/velocity.f90:233:      !$omp parallel do &
```

Describe what compilers you chose to use:

At first I searched google for what is better between the three compilers: ifort, gfortran and ifx, and I got this:

ifort is known for its high performance and optimization capabilities, particularly when running on Intel processors. This can make it a good choice for applications that require maximum performance, such as scientific or engineering simulations.

And ifort has good support for modern Fortran standards, which can make it easier to write and maintain high-quality Fortran code. It also has good support for OpenMP and MPI, which can be useful for parallelizing code on multi-core or distributed systems.

And after that I tested the three compilers on the second phase of parallelization (Where I had -O3 flag and no further openMP optimization), and I found that the three of them gave the same time, so given the explanation that I found on google, I've decided to go with iFort.

GPU offloading:

1. The program consists of a main loop (in problem.f90) which is dependent on previous iterations, making it difficult to offload to the GPU, To offload a function to the GPU, the longest function was chosen which was Calculate_derivatives (which was found in the vtune before).
2. Offloading the entire program to the GPU would be not feasible for this exercise, so I didn't do that, I only offloaded the calculate_derivative.
3. The solution is to map the relevant data structures before the main loop, update them on each iteration, and transfer arrays from CPU to GPU before Calculate_derivatives, and arrays from GPU to CPU after.

The implementation of this offloading was in two files, the problem.f90 and in velocity.f90, and all this was in the GPU folder.

I've added the relevant pragma and offloading function before the main loop in problem.f90, and add before and after the loop in velocity.f90 (calculate_derivatives) to send the data and get it back and update…