Optimizing ScalSALE

Yosef Goren

March 20, 2023

1 Summery

Introduction

In this project, we take the 3D Sedov-Taylor simulation, specifically the one found in the ScalSALE[1] paper implementation.
We start off from the original code written in Fortran 90, and go through a series of steps towrads optimizing the run on our specific hardware to get the shortest possible run time.
While the first part will be run fully on the CPU and the runtimes will be measured, the second part will be only theoretical - examining what should potentially be the best parts of the code to offload to a GPU, considering workloads, transfer times e.t.c.
The project with the source for it's final CPU implementation can be found at the github [3].

Tools

For the purposes of CPU parallalization - *openMP* was used. Additionally, *Intel VTune* was used to profile the code and find the most time consuming loops.

CPU Optimization

2 Hardware

In the following experiments, we are running on a NUMA system consisting of 2 NUMA nodes, each with 2 sockets, each with 6 cores. Each core has 2 hardware threads. So 24 hardware threads in total.
The CPU's themselves are *Intel Xeon Gold 6128*, running between 1.2 to 3.7 GHz.
This is hardware available on *Intel devcloud*[2].

3 Compilation

Objective

The objective of this part is to make the most out of the standard compiler technologies and achive a good serial CPU performance. To achive this task, different compilers with different optimization options were considered and trailed.

Compilers

Mutiple Fortran compilers have been trailed throught the development. Namely: 'GNU gfortran (7.4.0)', 'Intel ifort (2021.8.0)' and 'Intel ifx (2023.0.0)'.
In the table, the average runtime of a single simulation cycle is shown as a function of the compiler and the optimization level - The test is done on the original code. The calculation excludes the first cycle, since it often has a longer runtime - likely due to 'cold' caches.
This runtime for only one cycle should be negligable when many cycles are run.

	ifort	ifx	gfortran
O0 (none)	15.02	11.76	12.4
O2 (default)	5.08	4.93	12.41
Ofast	4.73	4.92	12.41

Running all three compilers shows that 'Intel ifort' is the fastest, while being marginally better than 'Intel ifx' and significantly better than 'GNU gfortran'. For the rest of the project, 'Intel ifort' will be used.

Flag Optimizations

After selecting the 'ifort' compiler with the 'Ofast' optimization level, a few specific flags were trailed. So before any of these took effect the runtime was 4.73 seconds.

- The '-ipo' flag:
  This flag enables interprocedural optimization between procedures defined in different files. This might be useful here since the main simulation loop takes place in many different sources.
  The result was a runtime of 3.39 seconds - equivalent to a $\times 1.39$ speedup.
- The '-xHost' flag:
  This flag requests the compiler to generate code which is best optimized for the current host machine.
  It enables the compiler to take into account the specific hardware which is available. The resulting runtime was 3.8 seconds - equivalent to a $\times 1.24$ speedup.

- The '-g' flag (check removed):
  The '-g' adds debug information and other compilation changes to make the output easier to debug, removing it did not make any noticable difference (but did make a difference when an explicit optimization level flag is not provided. In this case it changes the default optimization level from '-O2' to '-O0').

When combining the '-ipo' and '-xHost' flags, the we get a runtime of 2.44 seconds - equivalent to a $\times 1.94$ speedup.

By examining a run of this version with Vtune 'HPC' mode, we can see that the 'calculate_derivatives' procedure has seen vectorization [1] - likely due to the '-xHost' flag.

4 Parallel loops

Objective

The objective of this part is find the parts of the code that repeat the most and see if and how they can be run in parallel, which will hopefully lead to a good speedup with a simple few directives.

Parallelization

In order to quickly identify parallelization candidates in the code, Vtune was run on 'Hotspots' mode. It highlighted 6 procedures which together summed to 70% of the CPU time - the first of which being called 'calculate_derivatives' and being run during 37% of the CPU time.

Observing the 'calculate_derivatives' we can see it mostly consists of 3 nested loops: Within each nested iteration, a stencil at some (i, j, k) within the positions tensor is gathered, followed by gathering the same stencil from the velocities tensor.
This is then followed by a calculation which strictly depends on these stencils with the addition of the volume of the cell calculate elsewere. This calculate is intensive in float multiplication operations.

Inserting a simple parallel directive on the nested loop was able to achive the runtime of 1.891 seconds. This puts the global speedup at $\times 1.29$, assuming the runtime for the rest of the code remains static - we have a speedup of $\times 3.42$ for the 'calculate_derivatives' procedure [2]. Setting the scheduling to guided was

[1] In the procedue a small conditional branch was meant to decrease the workload by ignoring cells with negligable volume, however, this branch prevented vectorization - and was removed.

[2] This was calculated as $(2.44 - 0.63 * 2.44)/(1.801 - 0.63 * 2.44)$ since $0.63 * 2.44$ is assumed to be the runtime of the rest of the code.

able to get marginally better results at a cycle time of 1.841 seconds.

Moving on to the next most time consuming procedure, we have a procedure named 'calculate_vertex_mass_3d' taking 13% of the pre-parallelization CPU time. It also consists of the same nested loop structure. Adding the same parallel directive as before resulted in a runtime of 1.452 seconds per cycle.

Going forward with the next procedure named 'Calculate' in the 'volume' module - We get to 1.297 seconds per cycle.

Scheduling

Vtune analysis in 'Threading' mode shows how significat is the barrier time is as compared with the actual workloads in the parallel sections. Considering that the workloads are completely static (can be known before the iterations begin) - one might expect the static scheduling to perform the best, but this is far from the case.

A closer analysis shows how short the workload is - which would make is sensitive to noise preemption by the OS: if one of the threads is preempted - all other threads must wait for it to continue.
The fact that the system is NUMA might also play a role in exacerbating the phenomenon as scheduling a thread to varying NUMA nodes could be costly.
Perhaps it is not surprising that experimentation shows the best scheduling to be 'guided': it's low overhead complements the short workload, and it's dynamic nature enables it to adapt to the specific run.

GPU Offloading

References

[1] SCALSALE: Scalable SALE Benchmark Framework for Supercomputers. Re'em Harel ,Matan Rusanovsky ,Ron Wagner ,Harel Levin, Gal Oren. Implementation on github: `https://github.com/Scientific-Computing-Lab-NRCN/ScalSALE`.

[2] Intel Devcloud is a cloud-based platform for developers and researchers to experiment with and learn Intel hardware and software: `https://www.intel.com/content/www/us/en/developer/tools/devcloud/overview.html`

[3] A github repository for this project: `https://github.com/yosefgoren/ScalSALE`