Optimizing ScalSALE

Yosef Goren

March 20, 2023

1 Summery

Administrative

The project with the source for it's final CPU and GPU implementations can be found at the github [3] repository. The CPU implementation is found in the 'yosef-parallel-impl' branch, while the GPU implementation can be found at the 'yosef-offloading' branch.
The implementation for the bonus part, i.e. the one which also optimizes the implementation for Euler rezoning is found in the 'yosef-bonus' branch.

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.

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.

[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.

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 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.

Results

To summerize the results a table of speedups is presented below. The baseline is taken as the non-optimized program, compiles with 'ifort'.

[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.

	cycle time (sec)	speedup
No Optimization	15.02	1
Standard Optimization	5.08	2.96
Aggerssive Optimization	4.73	3.17
Aggerssive+IPO	3.39	4.43
Aggerssive+IPO+Host Specific	2.44	6.15
All Opt+Easy Parallelization	1.841	8.16
All Opt+Full Parallelization	1.297	11.58

GPU Offloading

Objective

In this subsection - we examine our program and examine what are the parts of it which are most suitable for offloading to a GPU, which would hopefully be able to yield a speedup.

First Look

Upon inspection, we can see the program is built around a 'main loop' with strong circular dependency between it's iterations.
Ultimately - we would find a slice of the data which would be transfered to the GPU and processed there exclusively. As stated, this solution would require any data which interacts with our slice to be transfered as well.

The problem with this approach is that we quickly find that this mehodology requires doing the whole calculation on the GPU, which would indeed provide a great speedup, but realistically - the complex and branched nature of the implementation makes it infeasible; at-least for the scope of this project.

The other extreme oppisite approach would find the biggest 'simd'ish loops within the code, and offload them, while updating their data from and to the CPU with each iteration of the main loop.
The problem with the approach is the immense overhead of the data transfer; while it's hard to know without running the code - it's likely that this overhead will negate any speedup.

A more realistic approach would be to find a slice of the calculation which is significat enough to yield a speedup, while also have relativly little dependencies to on data which is processed on the CPU; to put it simply - we are interested in the parts of the program with the highest proportion between calculations and data dependencies.

Fundemental Memory Bound

Armed with our huristic - we go back to the code and attempt to find such a cross-section of the program. But upon further inspection we find that the program is structured in a way which makes the data dependencies very large and somewhat uniform:
The data of the program is mostly structrued in a set of 3D tensors which represent physical quantities, and the program consists of a series of cell-wise calculations: each taking in some of the tensors as input, and producing some of the tensors as output.

This circle of tensor-wide dependencies is very bad news for our huristic - since taking any subset of these calculations (which is not the full circle) requires transfering data on the scale of the tensors to and from the GPU.

Solution

With the understanding of the fundemental limitations which the structure of the program imposes, we can need to ask 'what is the best we can get under these circumstances?'.
Well - we can still attempt to approach our huristic from the other side - by searching for the calculations which have the highest amount of operations per cell, we might be able to find a slice of the program with a good proportion between calculations and data dependencies.
If any viable offload is exists - it would be in this slice.

Indeed we know a portion of the calculation that conforms with this idea; consider 'calculate_derivatives' procedure - which as we saw before - consists of more than a third of the serial CPU time - while not being more memory intensive than the other cell-wise calculations.
Our offloading solution consists of putting the input and output dependencies of 'calculate_derivatives' on the GPU, executing it on the GPU, while also updating it's inputs to the GPU and its outputs from the GPU at each iteration of the main loop.

Additional offloading ideas (not included from our implementation) are to chain more of the cell-wise calculations together, which could keep the data transfers the same, while increasing the amount of calculations done on the GPU.

Euler Rezoning (Bonus)

Continuing the code from the CPU Optimization part, we run our simulation on with Euler Rezoning and get a runtime of 3.79 seconds per cycle.

Vtune analysis on 'HPC' mode shows a function named 'calculate_advect_3d' as the biggest sequential bottleneck in the code - taking up 46.2% of the serial runtime.

The procedure consists of a few nested loops. Starting by parallelizing the largest one is simple enough by adding the appropriate directives and identifying the private variables.
A noteworthy addition to this is a few places where the procedure accesses tensors in neighboring cells, requires use the secure this access [3] . This has been done using the 'atomic' directive, and similarly with the tensor.

Parallelizing this this loop gets us to a runtime of 1.901 seconds per cycle. This putting us at a speedup of $\times 1.993$ (and even more considering 'Calculate_vertex_mass_3d' was already parallelized from the original CPU Optimizations part).

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`

[3] i.e. writes to 'vof_adv','mat_vof_adv','mat_cell_mass_adv','sie_vof_adv' and 'init_mat_layers_adv'