Master in High Performance Computing

Parallelization of the Pre-stack Depth Migration Code

Supervisor(s):
Umberta Tinivella,
Irina Davydenkova

Candidate:
Mohammad Enayati

10th edition
2023–2024

Summary

Pre-stack depth migration (PSDM) is a computationally intensive algorithm widely used in seismic imaging to accurately position subsurface reflectors. Its high computational cost, largely dominated by deeply nested loops and irregular memory access patterns, makes performance optimization essential for large-scale seismic processing. This thesis investigates performance improvements of a PSDM kernel through enhanced vectorization and pure OpenMP-based parallelization.

Initially, the code was analyzed to identify opportunities for improving compiler auto-vectorization within the most computationally demanding loops. Loop restructuring, reduction of data dependencies, and improved memory access patterns were applied to increase vectorization efficiency and better utilize modern CPU vector units. However, due to varying loop bounds across iterations, relying solely on SIMD optimization limits both workload balance and overall scalability. In such cases, combining outer-loop parallelism with inner-loop compiler auto-vectorization typically provides better control over workload distribution and vectorization efficiency.

Building on this observation, the optimized OpenMP implementation parallelizes higher-level loops while preserving the previously improved vectorized kernels. Special attention was given to workload distribution, scheduling strategies, and memory access patterns in order to minimize thread contention and maximize processor utilization. To evaluate the effectiveness of the proposed approach, both strong scaling and weak scaling studies were conducted to analyze performance behavior as the number of processing cores and problem size vary.

In addition to performance analysis, energy consumption considerations were incorporated by monitoring the energy usage of the OpenMP-parallelized implementation, allowing an assessment of the trade-offs between performance gains and energy efficiency on multi-core systems. Experimental results demonstrate that the optimized implementation achieves significant performance improvements over the baseline version while maintaining favorable energy characteristics. Overall, the findings highlight that combining improved vectorization with scalable thread-level parallelism and careful performance analysis can substantially accelerate PSDM workloads on modern high-performance computing architectures.

Acknowledgements

Foremost, I would like to express my heartfelt gratitude to my advisors, Dr. Umberta Tinivella and Dr. Irina Davydenkova, for their continuous support, guidance, and invaluable insights throughout this journey.

I am also sincerely grateful to Dr. Ivan Girotto, co-director of the MHPC program, for his dedication in organizing the training courses. I would also like to thank Dr. Michela Giustiniani for her valuable support throughout the thesis preparation process, and Dr. Ivan De La Cruz Vargas-Cordero for testing the optimized code using real-world data.

This work is funded by the National Recovery and Resilience Plan project TeRABIT (Terabit network for Research and Academic Big data in Italy - IR0000022 - PNRR Missione 4, Componente 2, Investimento 3.1 CUP I53C21000370006) in the frame of the European Union - NextGenerationEU funding.

Above all, I wish to thank my family for their constant love and unwavering spiritual support throughout my life.

Last but certainly not least, I dedicate my deepest gratitude to my beloved wife, Mandana, for her kindness, love, patience, and unconditional support. Thank you for standing by my side, especially during the most challenging times.

Contents

Chapter 1

Introduction

Seismic Unix (SU) is a comprehensive open-source seismic processing package developed in C at the Colorado School of Mines and maintained by the Center for Wave Phenomena and the broader geophysical community. The code is hosted on GitHub:

`https://github.com/JohnWStockwellJr/SeisUnix`

SU provides a large suite of utilities covering key steps of seismic data processing, including filtering, stacking, migration, velocity analysis, and visualization.

1.1 Why Parallelizing?

SU is a widely used open-source package for seismic data processing, but many of its tools were originally designed to run sequentially on a single CPU. As seismic datasets have grown dramatically in size due to advances in acquisition technology, sequential processing has become a serious bottleneck. Tasks like filtering, migration, stacking, and velocity analysis may involve terabytes of seismic traces, and executing them in a purely serial fashion can lead to long runtimes that slow down exploration and research workflows. Parallelization allows these workloads to be distributed across multiple processors or nodes, reducing execution time from hours or even days down to minutes.

At the same time, modern computing environments provide a wide range of parallel architectures, from multicore desktops to large high-performance computing clusters and GPUs. Without parallelism, SU cannot take advantage of these resources, leaving significant computing power underutilized. Many of SU's operations are also *embarrassingly parallel*, meaning they can be divided into independent tasks, such as processing different traces or frequency bands, with little need for inter-process communication. By parallelizing these workloads, seismic processing pipelines can achieve near-linear speedups, making it possible to run larger and more complex algorithms efficiently.

Parallelizing SU code is therefore not just about speed, but also about scalability and enabling new possibilities. High-performance frameworks such as MPI for distributed-memory clusters, OpenMP for shared-memory systems, or GPU acceleration with CUDA/OpenCL can be leveraged to handle the computational demands of modern seismic applications. This efficiency gain accelerates exploration projects, reduces costs by optimizing compute resources, and allows geophysicists to test advanced algorithms that would otherwise be computationally prohibitive. In short, parallelization ensures SU remains a powerful, practical, and future-ready tool for seismic research and industry.

1.2 Case Study

Among the many processing modules available in SU, this thesis focuses specifically on the 2D pre-stack depth migration (PSDM) code. While SU provides a wide range of utilities for seismic data filtering,

stacking, and imaging, PSDM represents one of the most computationally demanding and scientifically significant components (for more information, see Chapter 2). By concentrating on this module, the work aims to study its structure, evaluate its performance, and explore improvements through parallelization to better exploit modern high-performance computing architectures.

In this thesis, the PSDM workflow was implemented using several programs from the SU package. The core migration step was performed with `sukdmig2d`, which carries out 2D Kirchhoff depth migration based on a depth-velocity model, allowing accurate positioning of reflectors in complex geological settings. The velocity model required for migration was generated using `unif2`, which creates a 2D uniform velocity field as an initial approximation. Ray tracing was conducted with `rayt2d`, which computes two-dimensional seismic ray paths and travel times through the velocity model, providing essential input for the migration process. Finally, `sustack` was used to stack seismic traces, enhancing the signal-to-noise ratio by summing traces that share common reflection points. Together, these SU programs form the complete PSDM processing sequence adopted in this study.

1.3 HPC Architecture

The computations were carried out on the Leonardo booster partition, which provides access to GPU resources. The architecture of this partition is `x86_64`, supporting both 32-bit and 64-bit CPU execution modes with little-endian byte ordering. Each compute node is equipped with a single socket comprising 32 physical cores, organized into two NUMA domains, with one hardware thread per core. The processor is an Intel Xeon Platinum 8358 operating at a base frequency of 2.60 GHz, with a maximum turbo frequency of 2.60 GHz and a minimum frequency of 0.8 GHz. The cache hierarchy consists of 48 KB L1 data cache (L1d), 32 KB L1 instruction cache (L1i), 1280 KB L2 cache per core, and a shared 49,152 KB L3 cache.

The computations were also carried out on the Leonardo DCGP (Data-Centric General Purpose) partition, which provides CPU-only resources optimized for large-scale parallel workloads (see Appendix C). The architecture of this partition is `x86_64`, supporting both 32-bit and 64-bit CPU execution modes with little-endian byte ordering. Each compute node is equipped with two sockets, each comprising 56 physical cores, for a total of 112 cores per node. Hyperthreading is not enabled, providing one hardware thread per core. The processor is an Intel Xeon Platinum 8480+ operating at a nominal frequency of 3.00 GHz, with a minimum frequency of 0.8 GHz. The system exposes eight NUMA domains per node, reflecting the underlying memory hierarchy. The cache hierarchy consists of 48 KB L1 data cache (L1d) and 32 KB L1 instruction cache (L1i) per core, a 2048 KB L2 cache per core, and a shared 107,520 KB L3 cache per socket.

Unless otherwise stated, all results presented in this thesis were obtained on the Leonardo booster partition; computations performed on the Leonardo DCGP partition are explicitly indicated when discussed.

1.4 Performance Metrics

Using *performance* as a metric, defined simply as *work* divided by *time*, poses several challenges in practical and interpretative contexts. The primary difficulty lies in defining and quantifying work, which varies widely depending on the nature of the problem; whether it involves operations, memory transactions, or floating-point computations. Moreover, measuring work accurately can be complex, particularly in parallel systems, where synchronization, communication overhead, and load balancing must also be considered. These challenges make performance less intuitive and harder to compare across different systems or implementations. To address these issues, alternative metric such as speedup is commonly used. Speedup provides a relative measure of performance improvement by comparing the execution time of a parallel implementation to a baseline execution (often using a sequential baseline or, in some cases, another parallel configuration).

If $T(B)$ represents the compute time taken to execute the application on B nodes, and $T(N)$ represents the compute time when using N nodes, the speedup S_p with respect to B nodes as the baseline is defined as:

$$S_p = \frac{T(B)}{T(N)}. \tag{1.1}$$

Here, a 'node' is considered a computational unit, which can represent a physical node in a cluster, a CPU, a GPU, or any other processing element depending on the system's architecture and the context of the application. Note that, typically, speedup is measured relative to a single node, i.e., $B = 1$.

Moreover, when using only 'one core with one thread', where only in-core optimization is possible, the concept of speedup needs to be adapted as follows:

$$S_p = \frac{T_{baseline}}{T_{optimized}}, \tag{1.2}$$

where $T_{baseline}$ is the execution time of the baseline version and $T_{optimized}$ is the execution time of the optimized version.

1.5 Performance Tools

1.5.1 Intel Advisor

Intel Advisor, a performance-analysis and optimization tool provided within the Intel oneAPI framework, offers a comprehensive suite of features designed to support performance tuning across different hardware architectures. Its capabilities encompass single-core optimizations, such as vectorization and memory layout analysis, multi-core enhancements through threading and parallelism (e.g., OpenMP), and accelerator-oriented techniques, including GPU offload modeling. These tools collectively enable developers to identify performance bottlenecks and guide optimization strategies across a wide spectrum of computing resources. However, the scope of this thesis is deliberately limited to two of Intel Advisor's functionalities: vectorization analysis and roofline modeling.

Vectorization analysis is employed to detect and evaluate opportunities for exploiting SIMD (Single Instruction, Multiple Data) instructions on modern CPUs, thereby improving computational throughput. Roofline modeling, on the other hand, provides a performance visualization framework that correlates computational intensity with memory bandwidth, facilitating a systematic assessment of whether an application is bound by compute or memory resources. By concentrating on these two aspects, this thesis aims to provide a focused yet rigorous investigation into computational efficiency and the underlying performance-limiting factors. To perform the analyses, Intel Advisor was accessed on the Leonardo supercomputing cluster by loading the module:

```
module load intel-oneapi-advisor/2023.2.0
```

On the Leonardo booster partition, the CPU supports SIMD extensions with 128-bit, 256-bit, and 512-bit widths for floating-point computations. The 128-bit `SSE2`, `SSE4.1`, and `SSE4.2` extensions process 4 single-precision or 2 double-precision values per instruction and are useful as a baseline for comparison with wider SIMD units. The 256-bit `AVX` and `AVX2` handle 8 single-precision or 4 double-precision values and include fused multiply–add (FMA) support for improved performance, while 512-bit `AVX-512` processes 16 single-precision or 8 double-precision values at once, offering the highest parallelism at the cost of increased power consumption and potential frequency throttling.

To analyze the vectorization potential of the application, the survey analysis was collected using the command:

```
advixe-cl --collect=survey --project-dir=./advisor_proj --
    ./your_executable [args]
```

This pass identifies the most time-consuming loops and provides information about their vectorization status (scalar, partially vectorized, or fully vectorized), along with diagnostic reasons in cases where vectorization was not applied. The results stored in the project directory can then be combined with additional analyses such as `Trip Counts` and `FLOP` counting to construct a roofline model. To analyze the roofline model of the application, the survey analysis was collected using the command:

```
advixe-cl --collect=tripcounts --flop --project-dir=./advisor_proj
    -- ./your_executable [args]
```

1.5.2 Intel VTune

Intel VTune Profiler is a comprehensive performance analysis tool designed to help developers identify and resolve performance bottlenecks in CPU- and accelerator-based applications. It provides low-overhead sampling, event-based profiling, and microarchitecture-level insights to analyze hotspots, memory access patterns, vectorization efficiency, threading scalability, and power usage across modern Intel processors. VTune supports both system-wide and application-level analysis, integrates with common development environments, and offers intuitive visualizations that map performance issues directly to source code. By enabling data-driven optimization decisions, Intel VTune plays a critical role in improving application efficiency, scalability, and overall performance on Intel hardware platforms.

To perform the analyses, Intel VTune was used on the Leonardo supercomputing cluster by loading the appropriate module:

```
module load intel-oneapi-vtune/2023.2.0
```

Data were then collected using one of the following commands, depending on the desired analysis type. The `hotspots` collection identifies which functions and code regions consume the most execution time, helping to locate the main performance bottlenecks:

```
vtune -collect hotspots -result-dir vtune_proj
    ./your_executable [args]
```

Alternatively, the `hpc-performance` collection provides a broader analysis of computational efficiency, including CPU utilization, vectorization, memory behavior, and potential hardware limitations relevant to high-performance computing workloads:

```
vtune -collect hpc-performance -result-dir vtune_proj
    ./your_executable [args]
```

Chapter 2

Imaging Subsurface Structures Using 2D PSDM in SU

Within the broad suite of processing tools offered by SU, this thesis concentrates on the 2D PSDM module. Although SU includes numerous utilities for filtering, stacking, and imaging seismic data, PSDM stands out as one of the most computationally intensive and methodologically advanced components. Its complexity and central role in seismic imaging make it a key subject for detailed investigation.

Accurate imaging of subsurface structures in complex geological settings requires high-quality seismic reflection data, which can only be achieved if the velocity field is known with high precision. In this case, stacking velocities are not reliable because they are evaluated assuming flat horizons and constant lateral velocities at each common depth point [1, 2]. To avoid this problem, the velocity model building can be done only by using an iterative approach. In the case of complex geological settings, PSDM is essential to obtain the correct depths and geometries of structures [1] (see Appendix A). In fact, PSDM provides a powerful tool for performing velocity analysis because of its high sensitivity to the velocity error and its ability to handle both reflector dips and lateral velocity variations (e.g., [1]). In recent decades, thanks to the availability of high-performance computers, the iterative application of PSDM is the most commonly used method to determine the 2D and 3D seismic velocity fields (e.g., [3] and references therein).

The most powerful method was developed by Liu and Bleistein (1995; codes available in SU), in which the PSDM output consists of two migration sections characterised by the same phase, but different amplitudes. The algorithm performs the PSDM using the input and perturbed velocity models. The ratio of the amplitudes of these two PSDM sections is used to evaluate the residual velocity [4]. The result of the migration can be organized into common image gathers (CIGs): if the migrated reflections in the CIGs are flat, it means that a correct migration velocity was used to migrate the data [1]. By contrast, the slope of the reflections in the CIGs indicates an error in the migration velocity, which can be corrected by analyzing the residual energy and then updating the velocity.

The residual energy (called the r-parameter) is a measure of the flatness deviations of the reflections along the offset in each CIG [5]. A zero value for the r-parameter means that the velocity is corrected at the corresponding reflection. If the r-parameter has a negative/positive value, it means that the velocity must be increased/decreased. Then, using the theory of Liu and Bleistein (1995), the r-parameter is converted to a residual velocity used to update the input velocity. It is important to remember that, when the medium has strong lateral velocity variations, the algorithm works adequately even with small velocity corrections. All steps, such as PSDM, CIG analysis, r-parameter evaluation, velocity update, are performed iteratively until all reflections in the CIGs are reasonably flat, i.e. when the variation of the depth of the reflector versus offset is sufficiently small in each

CIG (see details in [4]). Note that this approach provides both sensitivity and error estimations for migration-based velocity analysis, which is helpful in assessing the reliability of the estimated velocity.

ISTRICI package is a tool for facilitating depth imaging of seismic data and velocity analysis by using PSDM iteratively. ISTRICI proposes three different ways to organize the CIGs to improve the seismic velocity analysis depending on different elements, such as targets and characteristics of the data, optimizing the result quality and reducing the time needed to analyze the r-parameters in the residual semblances. ISTRICI has been developed on the basis of the most common cases: (i) the continuity of some reflections along the seismic line, such as the seafloor, suggests that the picking procedure can be performed semi-automatically in the residual semblance (INTER workflow); (ii) the availability of the interpretation of the seismic section suggests that the velocity analysis can be performed along the interpreted reflectors (CIG workflow); (iii) in complex geological settings and/or in the presence of non-continuous reflectors, the velocity analysis can be performed as a function of depth rather than in a layer-stripping approach (TRAD workflow). A schematic representation of the three workflows is provided in Appendix B. The user can choose and move from one workflow to another in order to better resolve the targets and/or scientific questions on the basis of the characteristics of the data. Once the procedure is completed, ISTRICI can include a velocity gradient below the last inverted layer or a defined surface before performing the final PSDM to improve the seismic imaging.

A seismic line approximately 90 km long, acquired in the Tirrenian Sea (Italy), was analyzed to investigate deep subsurface structures. The geological environment is quite complex and only by using the PSDM, it is possible to obtain a reasonable imaging of the subsoil. So, the TRAD workflow of ISTRICI was used to extract information about the seismic velocities. The velocity model is created by using a sample in depth and distance equal to 15 m and 25 m respectively, while the number of samples in depth and distance is 601 and 3601 respectively. The seawater velocity was analysed after only two iterations (Figure 2.1-A), while the final velocity model for the first km below seafloor was obtained after 20 iterations (Figure 2.1-C). The results of the PSDM are reported in Figures 2.1-B and 2.1-D, in which the complexity of the geometry of the structure is clear.

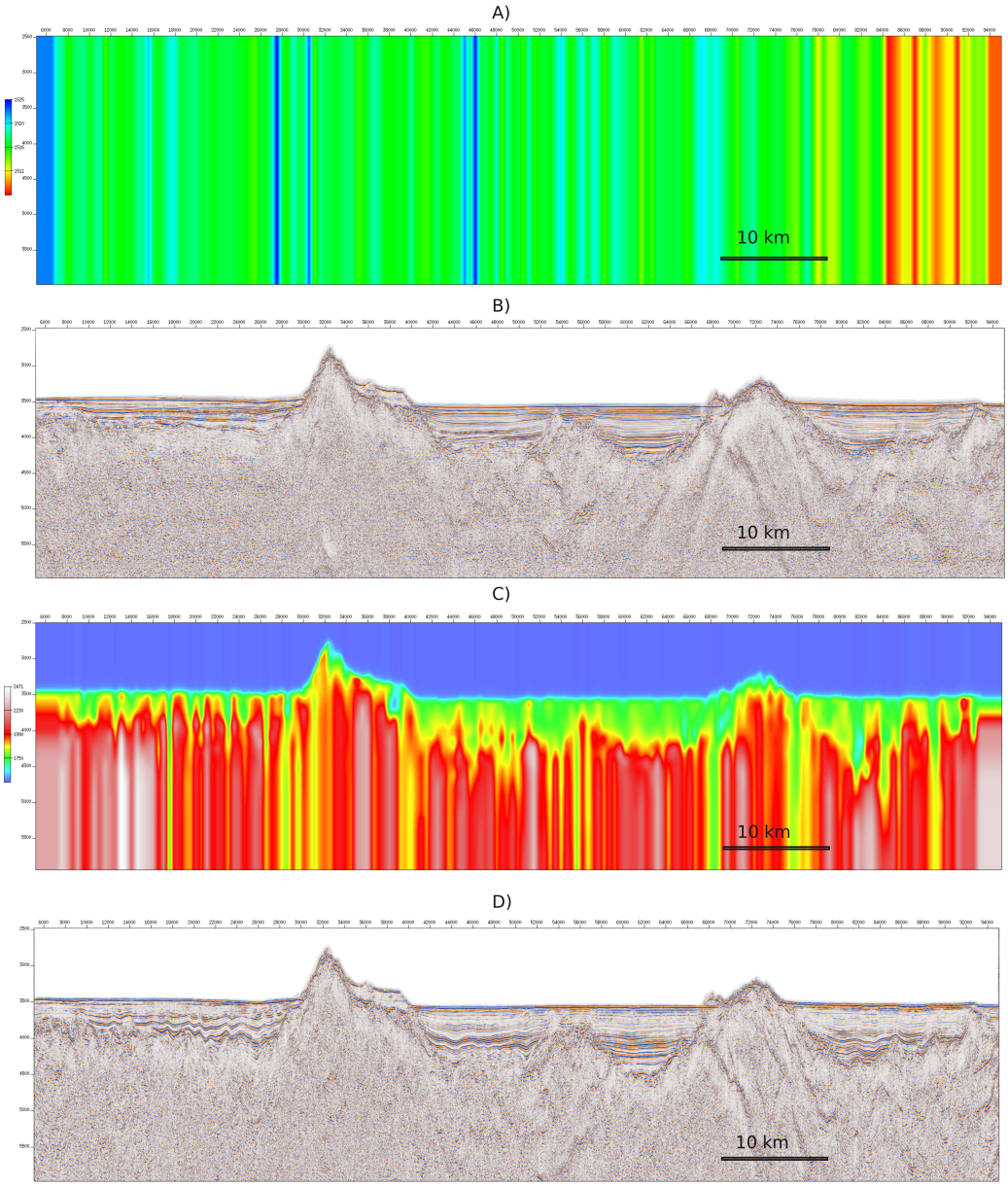

Figure 2.1: Velocity models and Pre-Stack Depth Migrated sections. In A) the initial water velocity model after 2 iterations, B) PSDM section by using initial water velocity model, C) Final velocity model after 20 iterations and D) PSDM section by using final velocity model.

Chapter 3

Single-Core Optimization

Before analyzing OpenMP parallelization, it is important to fully optimize single-thread performance. This includes enabling appropriate compiler optimizations, promoting vectorization, and addressing potential memory bandwidth limitations. Parallel speedup depends strongly on the efficiency of the serial execution; if the baseline performance is poor, adding threads will provide limited benefit and may introduce unnecessary overhead.

Optimizing the single-thread case first ensures that each core executes the workload efficiently, so that observed gains from OpenMP parallelization reflect effective scaling rather than compensation for serial inefficiencies.

All performance results reported in Chapters 3 and 4, including single-thread analyses and strong-scaling experiments, were obtained using a dataset with the lateral grid size $nx = 351$, the vertical grid size $nz = 401$, and 12120 traces, corresponding to a survey length of 8.75 km. An exception is made for the weak-scaling experiments, where the dataset configuration varies and is described separately in Section 4.9.

Finally, be cautious when applying optimizations, as certain optimizations can alter the outputs of the application. It is essential to verify that the results remain correct and consistent after optimization, particularly for computations sensitive to changes in precision, order of operations, or data dependencies.

3.1 Use a Better Compiler

Let us analyze the impact of different compilers, namely `GCC/8`, `GCC/12`, and `ICC/2023`, on the performance of the application. Table 3.1 reports the measured wall-clock runtimes for each compiler configuration. Among them, `GCC/12` shows the best overall performance, achieving the lowest elapsed time (or time-to-solution (TTS)).

The reference configuration is `ICC/2023`, which yields a runtime of 690 seconds and is therefore assigned a normalized speedup of 1.0. Compiling with `GCC/8` reduces the runtime to 567 seconds, corresponding to an improvement of about 20% over `ICC/2023` and a speedup of 1.2. Despite being an older release, `GCC/8` already provides more effective optimizations for this workload. The largest performance gain is obtained with `GCC/12`, where the runtime decreases to 342 seconds. This corresponds to a speedup of 2.01 relative to `ICC/2023` and about 1.66× compared to `GCC/8`, highlighting the optimization improvements introduced in newer `GCC` versions.

	ICC/2023	GCC/8	GCC/12
TTS (s)	690	567	342
Speedup	1.0	1.2	2.01

Table 3.1: Wall-clock runtime and normalized speedup obtained with different compilers.

Overall, these results highlight that `GCC/12` is the best-performing compiler for this workload. While `ICC/2023` is traditionally strong in HPC contexts, it underperforms here, and the nearly twofold performance gap compared to `GCC/12` is striking. Switching from `ICC/2023` to `GCC/12` cuts the runtime by about 50%, effectively doubling the performance. Based on this evidence, `GCC/12` (or newer) should be preferred as the default compiler choice for future runs.

Before proceeding, let us first provide a general overview of the code. Intel Advisor was used to identify performance hotspots and evaluate vectorization behavior. As shown in Figure 3.1, five hotspot loops were identified: four in the `mig2d` function and one in the `sum2` function. We refer to these kernels as `aliased depth-interp`, `lateral amp-filter`, `lateral-interp`, `amp-filter`, and `non-aliased depth-interp`, respectively. The report also indicates that only the `aliased depth-interp` loop is vectorized (marked by the orange circular arrow), while the other four loops execute in scalar mode (marked by the blue circular arrow).

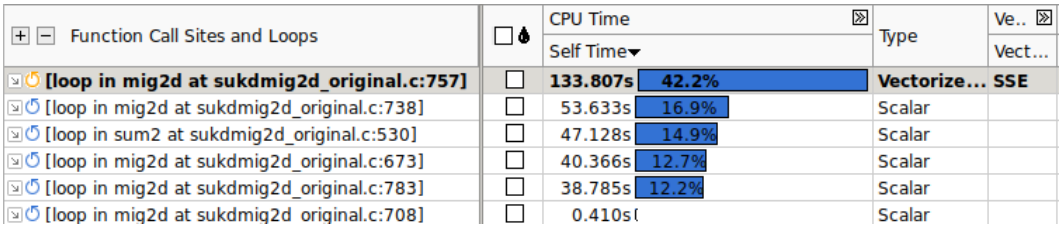

Figure 3.1: Hotspot loops identified in the baseline code compiled without optimization flags.

In the following, this thesis focuses on the identified hotspot loops and functions. The difficulty with optimizing at the statement level is that, aside from function calls, it is often ineffective because most statements correspond to only a few machine instructions. As a result, such small-scale optimizations usually produce limited improvement unless specific factors amplify the cost of a statement and make it hot enough to be worth optimizing. One of the most common amplifying factors is loops: the execution cost of statements inside a loop is multiplied by the number of iterations, making loops critical targets for optimization [1].

3.2 Use Your Compiler Better

Using a better compiler often means using the current compiler more effectively. For example, if an application runs slower than expected, it is important to verify that compiler optimizations are enabled. Although this may seem obvious, it is not uncommon to find cases where missing optimization flags significantly limit performance. In many situations, compiler optimizations alone can produce substantial speedups with no changes to the source code. After identifying performance bottlenecks, further gains can be achieved through targeted manual optimizations. This approach balances effort and effectiveness by combining automatic compiler optimizations with code-level improvements guided by performance analysis.

A practical starting point for performance evaluation is to enable the `-O3` optimization flag to establish a baseline, particularly when a quick improvement is desired. A comparison between `GCC/12`

[1] In practice, one can always find a slower portion of code, but trying to optimize everything can become a trap; therefore, the analysis is restricted to the measured hotspots.

compiled without optimization flags and with `-O3` shows a significant performance gain, see Table 3.2. The execution time decreases from 342 seconds to 138 seconds, corresponding to a speedup of about 2.48. This result clearly demonstrates the strong impact of compiler optimizations alone on runtime performance. In addition, the vectorization ratio[2] increases from 0.0% without optimization to 3.2% with `-O3`, indicating that the compiler is able to automatically exploit SIMD instructions when higher optimization levels are enabled.

Further improvements were observed when enabling `-march=native` together with `-O3`, leading to additional runtime reduction [3]. However, this configuration also produced slight changes in the numerical output, see Table 3.2 [4]. The output differences are likely due to architecture-specific optimizations introduced by `-march=native`, which allows the compiler to generate instructions tailored to the host CPU, potentially affecting floating-point evaluation and instruction selection. In particular, FMA instructions become available under this setting. It should be noted, however, that FMA provides limited benefit when the application is memory-bound rather than compute-bound (this aspect will be discussed in more detail in the following section).

Finally, enabling `-ffast-math` leads to a small increase in numerical error, see Table 3.2, although in this case the deviation remains negligible relative to the application requirements.

GCC/12	TTS (s)	Speedup	Error	Vectorization
gcc	342	1.0	--	0.0%
+ -O3	138	2.48	--	3.2%
+ -march=native	123	2.78	0.000104881	2.1%
+ -ffast-math	117	2.92	0.000114018	5.1%

Table 3.2: Performance of `GCC/12` with different compiler flags

Additional compiler flags were also tested in order to assess their potential impact on performance. In particular, options such as `-funroll-loops` and `-ftree-vectorize` were considered. However, these flags did not yield measurable performance improvements in the analyzed code. The main reason is that the relevant loops had been fully vectorized under the default optimization settings (e.g., `-O3`), leaving little room for further gains from manual unrolling or forcing vectorization. The `-fstrict-aliasing` flag was also evaluated and showed no observable effect on performance.

3.3 Data Types

Before focusing on increasing vectorization or in-core parallelization, it is important to first review the data types used throughout the code. A preliminary inspection suggests that different numerical

[2] Vectorization ratio is the percentage of floating-point operations executed as vector (packed) instructions, with 0% indicating fully scalar code and 100% indicating all operations are vectorized, regardless of the actual vector length used. In this work, the metric was measured using Intel VTune Profiler.

[3] Although the Leonardo system uses an Ice Lake CPU, the `-march=icelake-server` flag was not selected in order to keep the code more portable.

[4] The error is evaluated using the root mean square error (RMSE). It is computed by first calculating the squared difference between the reference and computed values at each of the N points. These squared differences are then summed and divided by N to obtain the mean square error. Finally, the square root of this quantity is taken to obtain the RMSE. Mathematically, it is defined as:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i^{ref} - x_i^{comp})^2}$$

where x_i^{ref} and x_i^{comp} denote the reference and computed values at point i, respectively.

Note that the RMSE values observed throughout the optimization process (maximum $\sim 1.5 \times 10^{-4}$) are considered acceptable for this application, as they remain well below the noise floor of the seismic data and therefore do not affect the interpretability of the migrated image.

precisions are used inconsistently, which may impact both performance and numerical stability. Standardizing and rationalizing precision choices is therefore a necessary preliminary step before applying low-level optimization strategies such as SIMD vectorization or thread-level parallelism.

Let's delve deeper into the above using Intel Advisor, which provides valuable insights and clarifications for future studies. For `GCC/12`, with the previously discussed flags enabled, Figure 3.2 shows that, for example, in the `aliased depth-interp` loop, the vector instruction set architecture (ISA) is `AVX512`, yet the vector length (VL) is limited 2 or 4 elements. Additional evidence is provided in Figure 3.3, where, despite `GCC/12` generating full-width 512-bit instructions for the `aliased depth-interp` loop, the loop remains predominantly memory-bound.

Function Call Sites and Loops		CPU Time	Type	Vectorized Loops		Instruction Set Analysis
		Self Time		Vector ISA	VL..	Data Types
[loop in mig2d at sukdmig2d_original.c:75		**45.979s**	**Vectorized (B...**	**AVX512**	**2; 4**	**Float32; Float64; Int32**
[loop in mig2d at sukdmig2d_original.c:673]		23.328s	Scalar			Float32; Float64; Int32
[loop in mig2d at sukdmig2d_original.c:783]		12.340s	Scalar			Float32
[loop in mig2d at sukdmig2d_original.c:738]		11.969s	Scalar			Float32
[loop in sum2 at sukdmig2d_original.c:530]		6.350s	Vectorized (Body)	AVX2	8	Float32

Figure 3.2: Hotspot loops identified in the baseline code compiled with optimization flags.

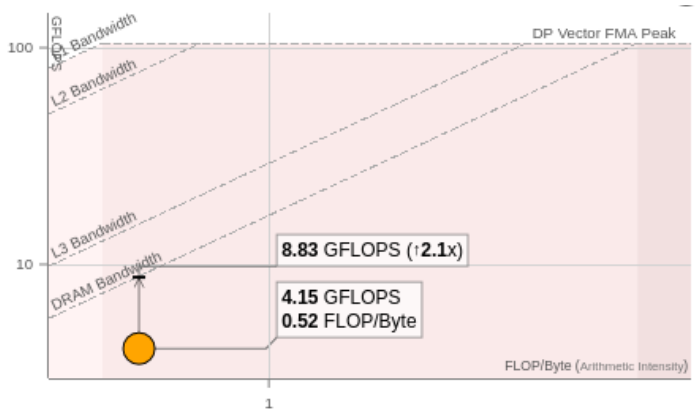

Figure 3.3: Roofline model of the `aliased depth-interp` loop, highlighting its performance regime (memory-bound vs. compute-bound).

In light of these findings, the next step is to reduce and standardize the data types used within the loop in order to improve vector efficiency and overall performance. As shown in Figure 3.2, the code appears to use different numerical precisions inconsistently, which may limit effective vector utilization. Therefore, we attempt to minimize the variety of data types within the loop to enhance uniformity and enable more efficient vectorization.

3.3.1 The `aliased depth-interp` Loop

Data type consistency can be improved, for example, by converting

```
sz0 = 1.0-sz;
```

to

```
sz0 = 1.0f-sz;
```

where `sz0` and `sz` are declared in single precision. In `C` and `C++`, a literal such as `1.0` is interpreted as a double-precision constant by default. Therefore, in a statement like `sz0 = 1.0 - sz`, even though both `sz0` and `sz` are declared as float, the constant `1.0` is a double. To carry out the subtraction, the compiler first promotes `sz` from float to double. The subtraction is then performed in double precision, and finally the result is converted back down to float before being stored in `sz0`.

If your intention is to work entirely in single precision, this implicit promotion to double is unnecessary and can reduce efficiency, especially inside performance-critical loops. A better approach is to make the constant explicitly single precision by writing `1.0f` instead of `1.0`, that is: `sz0 = 1.0f - sz`. In this case, both operands are of type `float`, the subtraction is executed in single precision, and no additional type conversions are required.

After applying this modification consistently to all relevant constants within the loop, the runtime decreased from 117 seconds to 112 seconds, indicating a clear performance improvement. The corresponding vectorization and roofline analyses are shown in Figure 3.4 and Figure 3.5, respectively. The updated roofline position suggests that the loop now behaves predominantly as compute-bound, which is reasonable, especially when considering its vectorization status.

Function Call Sites and Loops	CPU Time: Self Time	Type	Vectorized...		Instruction Set Anal...
			Vect...	VL (...	Data Types
[loop in mig2d at sukdmig2d_datatype.c:7	**44.229s**	**Scalar**			**Float32; UInt32**
[loop in mig2d at sukdmig2d_datatype.c:673]	20.989s	Scalar			Float32; Float64; Int32
[loop in mig2d at sukdmig2d_datatype.c:786]	12.480s	Scalar			Float32
[loop in mig2d at sukdmig2d_datatype.c:738]	12.360s	Scalar			Float32
[loop in sum2 at sukdmig2d_datatype.c:530]	5.781s	Vectorized (Body)	AVX2	8	Float32

Figure 3.4: Hotspot loops after standardizing data types within `aliased depth-interp` loop.

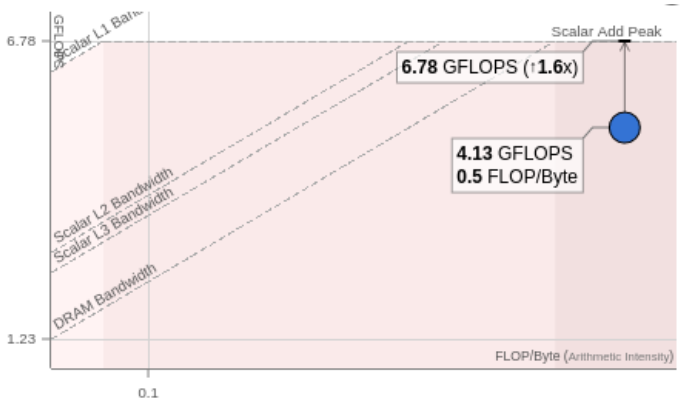

Figure 3.5: Roofline model of the `aliased depth-interp` loop after standardizing data types.

3.3.2 The `amp-filter` Loop

In this loop, simply converting constants such as `1.0` to `1.0f` is not sufficient to eliminate all double-precision operations. The presence of the mathematical function `cos()` in the following statement still introduces `float64` computations:

```
ampd = (cs0s+cs0g)*cos(0.5f*(angs-angg));
```

The reason is that `cos()` expects a `double` argument and returns a `double`. Even though the expression `0.5f * (angs - angg)` is evaluated in single precision, it is implicitly promoted to `double` when passed to `cos()`. Consequently, the trigonometric computation is performed in double precision, and the result

is then converted back to `float` when assigned, introducing unnecessary type conversions within the loop.

To ensure that the computation remains entirely in single precision, `cos()` should be replaced with `cosf()`, which operates on and returns `float`. After this modification, the loop uses only `float32` and `int32` data types. This change further reduces the runtime from 112 seconds to approximately 106 seconds, as expected. The corresponding vectorization analysis is shown in Figure 3.6.

Function Call Sites and Loops		CPU Time Self Time	Type	Vectorized... Vect...	Vectorized... VL (...	Instruction Set Analysis Data Types
[loop in mig2d at sukdmig2d_datatype.c:757]		44.529s	Scalar			Float32; UInt32
[loop in mig2d at sukdmig2d_datatype.c:673]		**18.269s**	**Scalar**			**Float32; Int32**
[loop in mig2d at sukdmig2d_datatype.c:738]		13.080s	Scalar			Float32
[loop in mig2d at sukdmig2d_datatype.c:786]		12.700s	Scalar			Float32
[loop in sum2 at sukdmig2d_datatype.c:530]		4.889s	Vectorized (Body)	AVX2	8	Float32

Figure 3.6: Hotspot loops after standardizing data types within `amp-filter` loop.

It is worth noting that certain compiler flags can help detect unintended type promotions between `float` (32-bit) and `double` (64-bit). In `GCC`, there is the `-Wdouble-promotion` option generates a warning whenever a `float` value is implicitly promoted to `double`. Conversely, the `-Wfloat-conversion` flag warns when a `double` value is implicitly converted to `float`. Enabling these warnings is useful for enforcing consistent numerical precision throughout the code and for identifying hidden type conversions that may introduce unnecessary performance overhead or affect numerical behavior.

3.4 Optimize Hot Statements

Before proceeding further, it is important to clarify that modern compilers are highly effective at detecting loop-invariant code and automatically moving such computations outside the loop to improve performance. An expression is considered loop-invariant if its value does not depend on the loop control variable. In most cases, developers do not need to manually rewrite these sections, as the compiler performs this optimization reliably when sufficient optimization levels (e.g., `-O3`) are enabled.

For this reason, `-O3` is used as a baseline configuration. It ensures that standard optimizations, including loop-invariant code motion, are automatically applied without manual intervention. This is particularly important in the present case, where most variables are defined in single precision (`float32`). Manual restructuring of expressions may alter the evaluation order of floating-point operations and therefore slightly change numerical results.

Floating-point arithmetic is not associative; for example, $(a+b)+c \neq a+(b+c)$ due to rounding effects. Since the parameters are stored in single precision, such differences may become more noticeable if expressions are manually reorganized. Therefore, it is preferable to allow the compiler to handle invariant code motion and related transformations.

Although using `double` precision could reduce rounding sensitivity, `float` is retained for two main reasons: (i) memory efficiency, since `float` requires half the storage of `double`, which is significant for large-scale datasets such as seismic traces; and (ii) computational efficiency, as single-precision operations are often faster and allow better vector utilization. Based on the numerical validation performed, the observed error remains negligible, justifying the use of single precision.

3.5 Unrolling

We rely on the compiler to perform loop unrolling because modern compilers are generally effective at this, producing code that is cleaner, more portable, and often achieves better performance. Moreover,

in our implementation, the bounds of the inner loop vary depending on the iteration of the outer loop, making manual unrolling impractical and potentially error-prone.

3.6 In-core Parallelization: (Auto)vectorization

Four most important in-core features that are relevant for speeding up code execution are: (*i*) Instruction-Level Parallelism (ILP). (*ii*) Simultaneous Multi-Threading (SMT). (*iii*) Pipelining, and (*iv*) Vectorization or SIMD. Techniques such as ILP, pipelining, and hyper-threading are managed automatically by the architecture without requiring programmer intervention, while techniques like vectorization depends on the programmer's effort to optimize code or provide explicit instructions. In this work, we focus on making modifications that improve auto-vectorization; no intrinsic-based vectorization techniques are used.

The Roofline model (Figure 3.7) shows that most of the hot loops still execute in scalar mode. Even though their raw performance may appear satisfactory, these loops do not exploit SIMD instructions, leaving significant headroom for improvement. Successfully vectorizing these loops would increase throughput and bring performance closer to the hardware's theoretical peak.

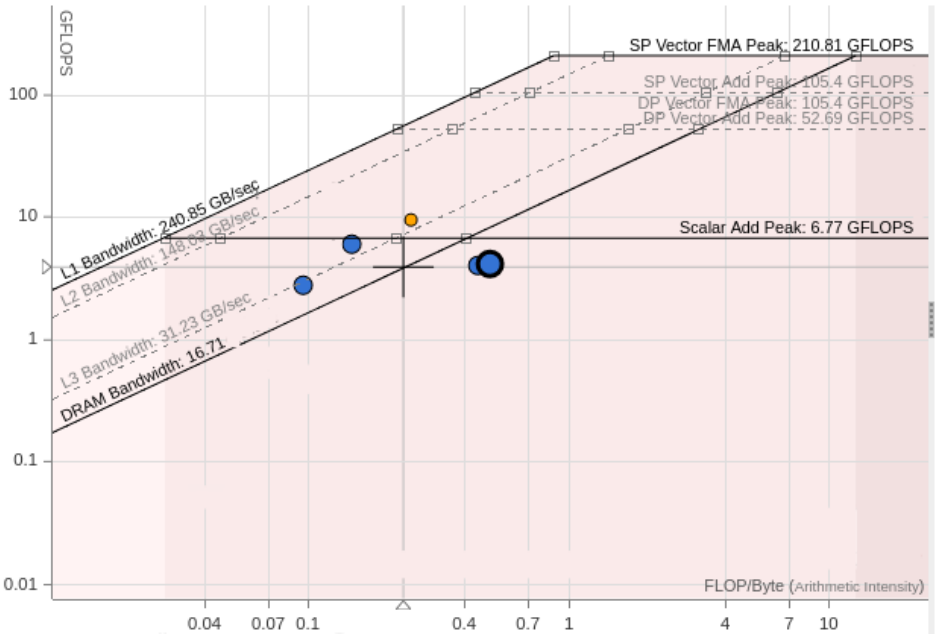

Figure 3.7: Roofline model of the analyzed loops, highlighting the scalar execution of hot loops due to control flow and other vectorization limitations.

Vectorization is a critical component of the `-O3` optimization level. However, as shown in Table 3.2, only 5.1% of loops are currently vectorized. Therefore, it is important to identify which loops are vectorized, which are not, and the reasons for missed vectorization opportunities.

To improve vectorization in the nested loops, one promising approach is inner-loop fission, which splits a large loop body into smaller loops with simpler operations. However, before applying loop fission, it is important to first eliminate branches and control flow within the loops. Removing conditional statements and other dynamic branching simplifies the loop structure, making it easier for the compiler to generate efficient SIMD instructions and fully exploit vector units. Once the control flow is minimized, inner-loop fission can be applied to further increase vectorization potential and overall performance. While here inner-loop fission is used, the approach is also applicable to

outer-loop fission as employed in OpenMP; see the next chapter for details.

3.6.1 Branch Elimination in Loops

To provide a solid ground for vectorization, we need to remove branches from loops. That means restructuring the loop body to avoid conditional statements like `if` or `switch` inside performance-critical iterations. Branches break the regular flow of the loop and make it harder for the compiler to generate SIMD instructions, since different iterations may follow different paths. By rewriting or eliminating these branches, the loop becomes more predictable and uniform, which gives the compiler a better chance to apply vectorization and deliver higher performance.

Using `MIN` and `MAX` Macros: Instead of writing explicit `if` statements, we can use simple macros like `MIN(a, b)` and `MAX(a, b)`, using the ternary operator, to keep a variable within a desired range (here, used only for integer data type). Considering these, for example,

```
if (nxtf < 0) nxtf = 0;
```

can be written more compactly as:

```
nxtf = MAX(nxtf, 0);
```

Note that modern compilers often translate `MIN/MAX` macros into efficient branch-free instructions.

By applying above, the RMSE decrease from `0.000114018` to `0.000109545`. This indicates that replacing `if` statements with branch-free macros (even for integers) can reduce mispredicted branches, which in turn can slightly alter the timing or execution order of nearby floating-point operations. Although the integer macros themselves don't introduce rounding errors, this more predictable execution can indirectly make floating-point computations more consistent, slightly reducing accumulated differences.

Using `fabsf()` Function: Instead of writing explicit `if` statements, we can use functions like `fabsf()` to replace manual checks such as `if (x < 0)`. That is,

```
if(ampd<0.0f) ampd = -ampd;
```

can be written more compactly as:

```
ampd = fabsf(ampd);
```

which is a branch-free operation. However, it should be noted that removing an `if` statement in this way means the `fabsf()` function is applied for all iterations of the loop, whereas with the explicit `if` it may only apply to a few iterations.

Using Masks: When a variable should only be updated under certain conditions, we can replace explicit `if` statements with mask-based expressions. For example, if `npv` is either `0` or `1` and we want to preserve the existing value when `npv == 0`, we can rewrite:

```
if(npv)
    amp1[iz] = ax0*ampt1[jx][iz]+ax*ampt1[jx+1][iz];
```

as

```
amp1[iz] = npv * (ax0 * ampt1[jx][iz] + ax * ampt1[jx+1][iz])
     + (1 - npv) * amp1[iz];
```

In this form, the assignment is effectively skipped when `npv == 0` (since the old value is multiplied by 1), and applied normally when `npv == 1`. This achieves the same behavior as an `if` statement, but without branching.

In other situations, we may want to accumulate the new contribution rather than overwrite the old value. In that case, the update can be expressed as:

```
amp1[iz] += npv * (ax0 * ampt1[jx][iz] + ax * ampt1[jx+1][iz]);
```

Here, when `npv == 0`, nothing is added and the original value of `amp1[iz]` is preserved. When `npv == 1`, the computed contribution is fully added to the existing value, producing the standard accumulated result.

Using `fminf` and `fmaxf` Functions: We can use `fminf()` and `fmaxf()` to keep a single-precision variable within a desired range. Considering these, for example,

```
if(temp<1.0f) temp = 1.0f;
if(temp>mtmax) temp = mtmax;
```

can be written more compactly as:

```
temp = fmaxf(1.0f, fminf(temp, mtmax));
```

3.6.2 Inner-Loop Fission

In numerical codes that involve deeply nested loops, the innermost loop is often a target for optimization. However, complex loop bodies that combine multiple computations and access several large arrays can introduce dependencies that prevent modern compilers from applying automatic vectorization. This means the compiler cannot easily map the loop iterations onto SIMD instructions, which are crucial for exploiting the full width of modern CPUs. Moreover, a large number of independent memory accesses in a single iteration can increase memory latency and reduce cache efficiency, since the processor must repeatedly fetch scattered data from main memory.

Loop fission is a technique where a single loop with multiple operations is transformed into several simpler loops, each performing only one part of the computation. By reducing the number of dependencies inside each loop, compilers are more likely to apply vectorization, and the memory access pattern becomes more regular. This technique trades off additional passes over the data for improved instruction-level parallelism, vectorization opportunities, and sometimes better cache utilization.

As an illustrative example, consider the following nested loop (the innermost loop is the `lateral amp-filter` loop):

```
for(ix=nxf; ix<=nxe; ++ix){
    ...
    for(iz=izt0; iz<nzt; ++iz){
        tzt[iz] = ax0*tsum[jx][iz]+ax*tsum[jx+1][iz]
                  +srs0*tb[jrs][iz]+srs*tb[jrs+1][iz]
                  +srg0*tb[jrg][iz]+srg*tb[jrg+1][iz];
        amp[iz] = ax0*ampt[jx][iz]+ax*ampt[jx+1][iz];
        ampi[iz] = ax0*ampti[jx][iz]+ax*ampti[jx+1][iz];
        tm[iz] = ax0*tmt[jx][iz]+ax*tmt[jx+1][iz];
        amp1[iz] = npv*(ax0*ampt1[jx][iz]+ax*ampt1[jx+1][iz])
                   + (1-npv)*amp1[iz];
    }
    ...
}
```

In this loop, the compiler encounters multiple independent computations that read from different input arrays (`tsum`, `tb`, `ampt`, `ampti`, `tmt`, `ampt1`) and write to separate output arrays (`tzt`, `amp`, `ampi`, `tm`, `amp1`). Although these operations are logically independent, they are interleaved within a single loop body. As a result, the compiler may conservatively assume potential memory aliasing between arrays, which can limit full vectorization. Moreover, each iteration performs several memory accesses to different locations, increasing pressure on the memory subsystem and further complicating efficient SIMD execution.

After applying inner-loop fission, the resulting loop structure is shown below:

```
for(ix=nxf; ix<=nxe; ++ix){
    ...
    for(iz=izt0; iz<nzt; ++iz){
        tzt[iz] = ax0*tsum[jx][iz]+ax*tsum[jx+1][iz]
                  +srs0*tb[jrs][iz]+srs*tb[jrs+1][iz]
                  +srg0*tb[jrg][iz]+srg*tb[jrg+1][iz];
    }
    for(iz=izt0; iz<nzt; ++iz)
        amp[iz] = ax0*ampt[jx][iz]+ax*ampt[jx+1][iz];
    for(iz=izt0; iz<nzt; ++iz)
        ampi[iz] = ax0*ampti[jx][iz]+ax*ampti[jx+1][iz];
    for(iz=izt0; iz<nzt; ++iz)
        tm[iz] = ax0*tmt[jx][iz]+ax*tmt[jx+1][iz];
    for(iz=izt0; iz<nzt; ++iz)
        amp1[iz] = npv*(ax0*ampt1[jx][iz]+ax*ampt1[jx+1][iz])
                    + (1-npv)*amp1[iz];
    ...
}
```

As an illustrative example, let us focus on the loop that computes the `amp` array. This loop has only two loads and one store per iteration, making it simple and well-structured for the compiler to vectorize across `iz`. Since all memory accesses are independent and stride-1 along the `iz` dimension, the compiler can safely generate SIMD instructions.

By splitting the loop, the runtime decreased to approximately 98 seconds. The corresponding roofline model was also analyzed before and after loop fission (see Figure 3.8 for the original loop and Figure 3.9 for the split version). In the original implementation, the loop was effectively limited to scalar execution, remaining well below the vector performance ceiling. After loop fission, the simplified loop bodies enabled successful vectorization, allowing the loop to move from the scalar region toward the vector performance regime. This shift is reflected in the roofline model by increased arithmetic throughput and improved utilization of the compute resources.

Further optimization of the `amp-filter` loop, followed by applying the same restructuring strategy to the `lateral amp-filter` loop, reduced the total runtime to approximately 86-88 seconds. In kernel `amp-filter`, loop splitting required an additional modification: per-iteration scalar temporaries such as `angs`, `angg`, `cs0s`, and `cs0g` were replaced with arrays. This change enabled the compiler to recognize data parallelism more clearly and vectorize the loop more effectively.

However, the same optimization strategy could not be directly applied to the `aliased depth-interp` and `non-aliased depth-interp` loops, due to structural or dependency constraints that prevented safe loop fission and effective vectorization.

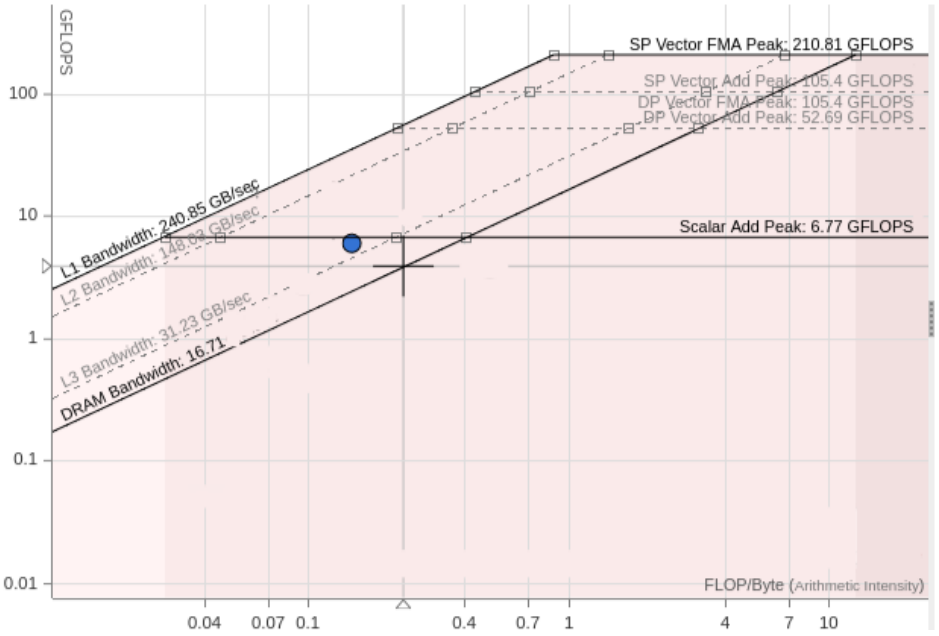

Figure 3.8: Roofline model of the original kernel before loop fission, showing scalar execution and limited vector utilization.

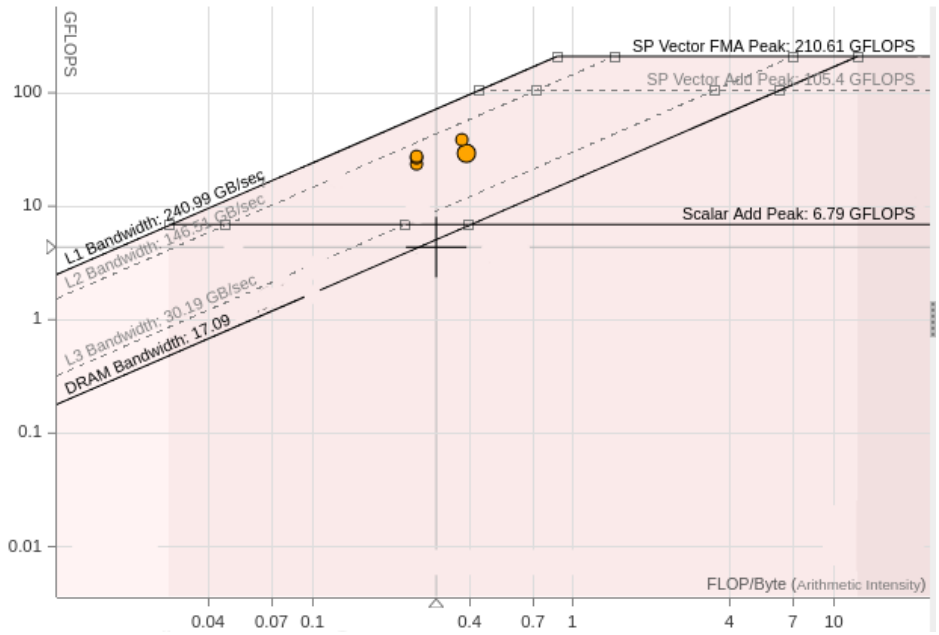

Figure 3.9: Roofline model of the kernel after loop fission, demonstrating improved vectorization and higher computational throughput.

Chapter 4

Multi-Core Parallelization: OpenMP

Parallelizing the outermost loop is often the preferred strategy in OpenMP programs because it minimizes synchronization overhead, especially the implicit barriers that occur at the end of parallel regions and parallel loops. When loop bounds vary across iterations, combining outer-loop parallelism with inner-loop SIMD typically provides better control over workload distribution and vectorization efficiency. In the previous chapter, we adopted this hybrid approach by exploiting inner-loop SIMD, whereas in this chapter we shift our focus to outer-loop parallelization to further reduce synchronization costs and improve scalability across multiple cores.

The analysis is restricted to the hot functions `mig2` and `sum`, with particular emphasis on their three outermost loops. In the `sum` function, the relevant loop is referred to as the `lateral-interpolation` loop. In `mig2`, the corresponding outer loops are identified in the code as the `compute-amplitude` loop and the `interpolate-amplitude` loop.

As noted earlier, all results presented here were obtained on the Leonardo booster partition, unless otherwise specified; any computations performed on the Leonardo DCGP partition are explicitly identified in the corresponding discussion.

4.1 Clarify the Variables' Scope

Declaring a loop variable directly inside the `for` loop is advantageous because it limits the variable's scope to the loop body, reducing the risk of unintended modifications or conflicts in other parts of the code. Once the loop completes, the variable is automatically destroyed, improving maintainability and preventing potential bugs. Additionally, this practice enhances readability by making it clear that the variable is only used within the loop. Following compilers like `GCC`, `Clang`, or Intel's `ICC`, it is considered best practice to declare variables in the narrowest scope possible, ensuring cleaner, safer, and more maintainable code by avoiding unnecessary *scope creep*.

When using OpenMP, this principle becomes even more important. Placing variables inside the loop body, rather than defining them outside, reduces the need for explicit `private` or `firstprivate` clauses, since such variables are inherently local to each iteration and therefore cannot be unintentionally shared across threads, effectively preventing race conditions. This approach also enables more aggressive compiler optimizations: loop-local variables can typically be allocated directly in CPU registers, whereas variables declared outside the loop and then marked with `private` or `firstprivate` require OpenMP to create separate memory-backed copies for each thread, introducing additional overhead. Moreover, forcing variables to be private at the OpenMP level often restricts the compiler's ability to reason about their lifetimes, limiting register allocation and other short-lived variable optimizations. By ensuring variables are created inside the loop rather than privatized

externally, we avoid unnecessary stack allocations, reduce OpenMP bookkeeping, and achieve cleaner, faster, and race-free parallel execution.

4.2 Avoiding Huge Private Arrays

We first focus on the `interpolate-amplitude` loop. In the original implementation, this loop contained several large per-iteration arrays (`tzt`, `amp`, `ampi`, `tm`, `amp1`). Privatizing such arrays within an OpenMP parallel region would require allocating large memory blocks per thread, thereby increasing memory pressure and potentially generating additional NUMA traffic.

To avoid this overhead, these arrays were replaced by global work arrays indexed as `[ix][iz]`. This approach eliminates the need for large thread-private allocations and promotes a more compact, cache-friendly memory layout shared across threads. For example, the original allocation for `tm`, that is, `alloc1float(nzt)` was replaced with `alloc2float(nzt, nx)` and correspondingly, `free1float(tm)` was replaced with `free2float(tm)`.

In the refactored design, each thread writes exclusively to its own slice of the global temporary arrays, indexed by `ix`. Since no two threads update the same memory region, race conditions are naturally avoided. This design eliminates the need for locks, atomics, or critical sections, which is essential for achieving good scalability at high core counts. A similar restructuring can be applied to the `compute-amplitude` loop.

4.3 OpenMP Thread Affinity

Thread affinity is a critical aspect of performance optimization on modern multicore systems. Contemporary processors feature complex topologies composed of sockets, cores, caches, and NUMA domains. Because of this hierarchical structure and the presence of shared resources, performance strongly depends on where threads are executed. Leaving thread placement decisions entirely to the operating system or runtime may result in unpredictable mappings, increased resource contention, and significant run-to-run variability. Therefore, explicit control of OpenMP thread affinity is essential to ensure stable and reproducible performance.

In this work, thread placement is controlled using the OpenMP standard affinity mechanism through environment variables. Specifically, the following configuration is employed:

```
export OMP_NUM_THREADS=<N>
export OMP_PLACES=cores
export OMP_PROC_BIND=close
```

The variable `OMP_NUM_THREADS` is set to `N`, the number of physical cores allocated to the job. By setting `OMP_PLACES=cores`, each OpenMP place corresponds to one physical core. Since the Leonardo Booster partition does not enable hyperthreading (SMT), each core provides a single hardware execution context. Consequently, defining places as cores enforces a one-thread-per-core mapping. The directive `OMP_PROC_BIND=close` ensures a compact placement strategy, meaning that threads are assigned to consecutive cores in a deterministic manner.

Although the Booster partition does not support hyperthreading, explicit affinity control remains important. It prevents unintended thread migration across cores, improves cache locality, and guarantees predictable utilization of the hardware topology. Moreover, compact binding helps maintain locality within sockets and minimizes unnecessary cross-domain communication. This approach eliminates performance variability due to uncontrolled scheduling and provides a consistent execution environment for benchmarking and scalability studies.

4.4 Outer-Loop Parallelization Without Fission

In this experiment, outer-loop parallelization without loop fission was applied using OpenMP (an example is the `interpolate-amplitude` loop shown in Listing 1). The execution time for 1, 2, 4, 8, 16, and 32 threads was measured as 125, 74, 46, 31, 26, and 30 seconds, respectively (see Table 4.1).[1] The results show good initial scaling as the number of threads increases up to 16 threads, indicating that the outer loop exposes a reasonable degree of parallelism. However, the performance improvement saturates beyond 16 threads, and a degradation is observed at 32 threads (see Figure 4.1, blue line).

```
// Interpolate amplitudes and filter length along lateral
#pragma omp parallel for
for ix = nxf ... nxe do
    compute izt0, iz0, ax, ax0, jx, ar, jrs, srs, srs0, jrg, srg, srg0

    for iz = izt0 ... nzt do
        tzt(ix,iz) <--- lateral interpolation of tsum and tb
    for iz = izt0 ... nzt do
        amp(ix, iz)  <-- lateral interpolation of ampt
    for iz = izt0 ... nzt do
        ampi(ix, iz)  <-- lateral interpolation of ampti
    for iz = izt0 ... nzt do
        tm(ix, iz)  <-- lateral interpolation of tmt
    for iz = izt0 ... nzt do
        amp1(ix, iz)  <-- lateral interpolation of ampt1 (if npv)

    compute nzp

    // Interpolate along depth if operater aliasing
    for iz = iz0 ... nzp do
        Compute aliasing depth interpolation
        Update mig(ix, iz), mig1(ix, iz)

    // Interpolate along depth if not operater aliasing
    for iz = nzp ... nz do
         Compute non-aliasing interpolation
         Update mig(ix, iz), mig1(ix, iz)
```

Listing 1: OpenMP implementation of the outer-loop parallelization without applying loop fission.

#threads	Elapsed Time (s)	Speedup
1	125	1.00
2	74	1.69
4	46	2.72
8	31	4.03
16	26	4.81
32	30	4.17

Table 4.1: Execution time and corresponding speedup of the OpenMP outer-loop parallelization without loop fission for different thread counts.

[1] Note that when analyzing parallel scaling, the baseline should be the same code executed with a single thread, rather than the original serial version.

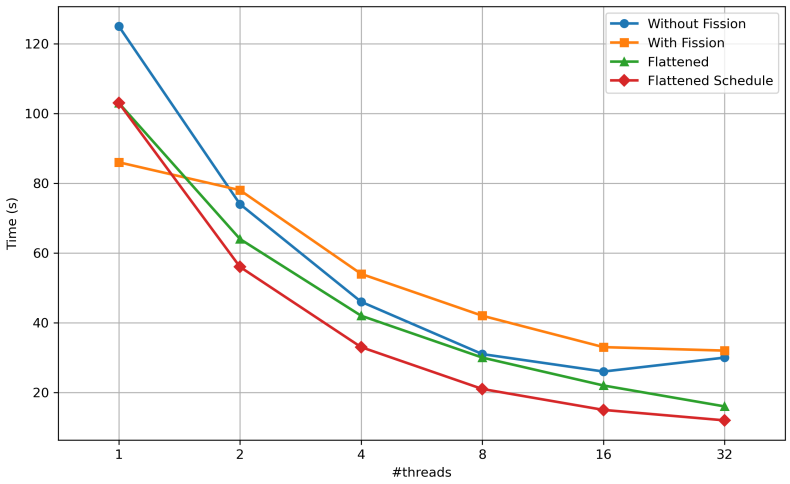

Figure 4.1: Execution time as a function of thread count for the various OpenMP parallelization approaches analyzed in this work.

The corresponding speedup[2] follows a similar trend. It increases steadily as the number of threads increases and reaches its maximum value at 16 threads. Beyond this point, adding more threads does not provide additional performance gains; instead, the speedup decreases at 32 threads (see Table 4.1 and Figures 4.2 and 4.3, blue lines). This confirms the presence of a scalability limit in the outer-loop parallelization without loop fission.

The observed saturation and performance degradation beyond 16 threads can be attributed to parallel overheads and limited exploitable parallelism. As the number of threads increases, the workload per thread decreases, making overheads such as synchronization, scheduling, and thread management more significant. In addition, potential load imbalance and shared-resource contention (e.g., memory bandwidth pressure) further limit scalability. The absence of loop fission restricts the available parallel granularity, preventing efficient utilization of higher thread counts.

[2]The reported speedup is relative to the single-thread execution time of this specific code version. Later tables correspond to different code variants with different single-thread baselines, so speedups across tables are not directly comparable.

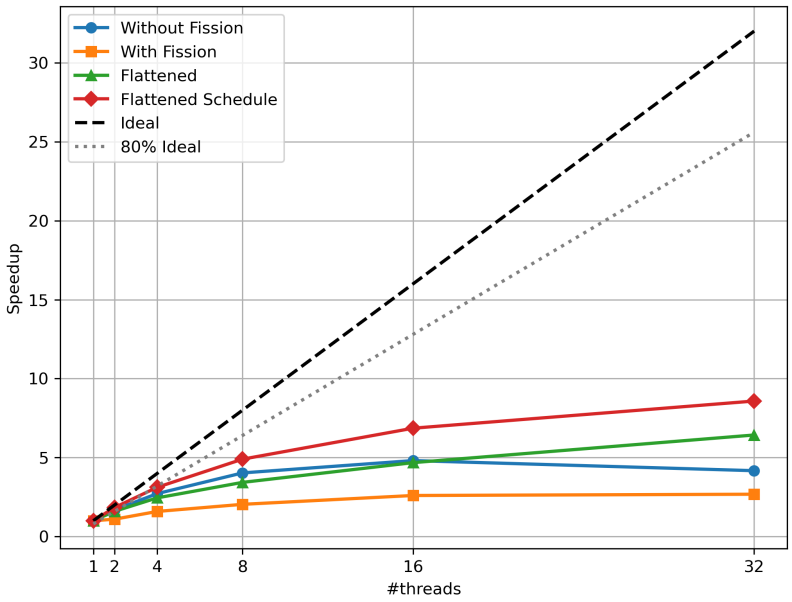

Figure 4.2: Speedup versus number of threads for the different OpenMP parallelization strategies (true numerical spacing on the x-axis).

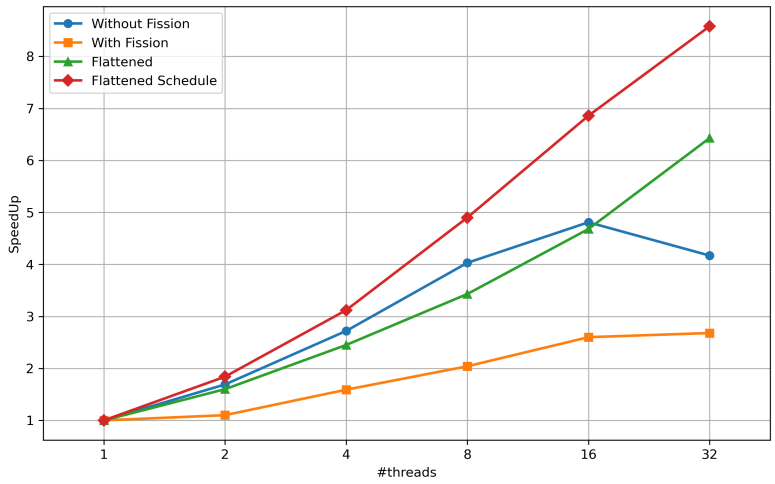

Figure 4.3: Speedup versus number of threads for the different OpenMP parallelization strategies (equal spacing on the x-axis for visual comparison).

4.5 Outer-Loop Parallelization With Fission

To efficiently parallelize the migration kernel, the original monolithic loop was decomposed into several OpenMP-parallel sub-loops. This restructuring was necessary because the original loop performed a large number of heterogeneous operations with varying data dependencies and memory-access patterns. Splitting the computation into distinct stages eliminates false dependencies, increases data parallelism, and allows both the compiler and the OpenMP runtime to optimize each computational phase independently.

Decomposing the loop also enables targeted optimization of each sub-loop according to its computational characteristics. In the original structure, the most time-consuming operations, contained in the `aliased depth-interp` and `non-aliased depth-interp` loops, were embedded within a large, heterogeneous loop body, making it difficult to apply specialized OpenMP clauses or compiler optimizations without affecting other computations. With the new structure, optimization efforts can focus on the parts that provide the greatest benefit. Different OpenMP strategies can be applied to different sub-loops, improving performance and scalability. Moreover, smaller, focused loop bodies allow variables initialized at the beginning of a loop to remain in registers, enhancing execution efficiency. Scheduling and task-based controls can also be applied more easily to balance load among threads.

As a potential direction for future work, selectively offloading only the most time-consuming sub-loop using OpenMP could avoid transferring all data associated with the original monolithic loop to the accelerator. In this case, only the data required by the offloaded sub-loop would be mapped to the device. Such selective data movement would reduce communication overhead and could further improve offloading efficiency.

Decomposing the outer loop into distinct stages also naturally enables a staged execution model. Each stage is parallel and, in our case, independent of the previous one, except for reading previously computed vectors (which prevents the use of `non-blocking` operations). In this implementation, the migration algorithm is structured as a eight-stage staged decomposition (see Listing 2), with each stage performing a specific part of the computation.

```
// Interpolate amplitudes and filter length along lateral
#pragma omp parallel
{
// ix = nxf ... nxe
#pragma omp for
for ix do compute dis_v[ix]

#pragma omp for
for ix do compute izt0_v[ix]

#pragma omp for
for ix do compute iz0_v[ix]

#pragma omp for
for ix do
    Compute ax, ax0, jx, nzp_v[ix]
    // iz = izt0_v[ix] ... nzt-1
    for iz do tzt(ix,iz)  <-- lateral interpolation of tsum
    for iz do amp(ix,iz)  <-- lateral interpolation of ampt
    for iz do ampi(ix,iz) <-- lateral interpolation of ampti
    for iz do tm(ix,iz)   <-- lateral interpolation of tmt
    for iz do amp1(ix,iz) <-- lateral interpolation of ampt1 (if npv)

#pragma omp for
for ix do
    Compute jrs, srs, srs0
    for iz do update tzt(ix,iz) <-- lateral interpolation of tb

#pragma omp for
for ix do
    Compute jrg, srg, srg0
    for iz do update tzt(ix,iz) <-- lateral interpolation of tb

// Interpolate along depth if operater aliasing
#pragma omp for
for ix do
    for iz = iz0_v[ix] ... nzp_v[ix]-1 do
        Compute aliasing depth interpolation
        Update mig(ix, iz), mig1(ix, iz)

// Interpolate along depth if not operater aliasing
#pragma omp for
for ix do
    for iz = nzp_v[ix] ... nz-1 do
         Compute non-aliasing interpolation
         Update mig(ix, iz), mig1(ix, iz)
} // end parallel region
```

Listing 2: OpenMP implementation of the migration kernel structured as a staged decomposition, where each stage performs a specific part of the computation.

In the above staged decomposition, a single parallel region was used for all stages instead of creating separate parallel regions for each stage. This approach reduces the overhead of repeatedly starting and stopping threads, preserves data locality, and improves overall performance. It also allows the use of the `nowait` clause when appropriate. Indeed, the `nowait` clause is applied at the end of stages that produce output consumed only by later non-adjacent stages, eliminating the implicit barrier and

allowing faster threads to proceed immediately to the next independent loop. It is not applied between stages where a data dependency exists.

Table 4.2 reports the execution time and corresponding speedup of the migration kernel for different thread counts. The results show that execution time decreases steadily as the number of threads increases, from 86 seconds with a single thread to 32 seconds with 32 threads (see also Figure 4.1, orange line). The speedup follows a similar trend, reaching a maximum of 2.68× at 32 threads (see also Figures 4.2 and 4.3, orange lines).

#threads	Elapsed Time (s)	Speedup
1	86	1.00
2	78	1.10
4	54	1.59
8	42	2.04
16	33	2.60
32	32	2.68

Table 4.2: Execution time and corresponding speedup of the OpenMP outer-loop parallelization with loop fission for different thread counts.

Although loop fission improves parallelism in the migration kernel, the observed speedup saturates at higher thread counts, with only a modest increase from 2.60× at 16 threads to 2.68× at 32 threads. This sub-linear scaling suggests that adding more threads does not translate into proportional performance gains. One possible explanation is that the kernel is memory-bound, meaning that the CPU cores spend a significant portion of time waiting for data to be loaded from memory rather than performing computations. The large working set of the migration kernel, combined with irregular memory-access patterns, can lead to memory bandwidth saturation, limiting the benefit of additional threads.

4.6 Flattened Memory Layout for Improved Performance

One effective approach to mitigate memory-bound limitations is to use flattened memory allocation for multi-dimensional arrays. For instance, instead of accessing the array as `[ix][iz]`, a flattened 1D representation such as `[ix*nzt + iz]` ensures that consecutive elements along the inner loop are stored contiguously in memory [3]. The most time-consuming loops, as identified using the Intel VTune profiler, are the `aliased depth-interp` and `non-aliased depth-interp` loops, corresponding to the last two loops in Listing 2; therefore, our analysis focuses on these loops.

Table 4.3 shows the effect of flattened memory allocation on the performance of the most time-consuming loops in the migration kernel. The execution time decreases significantly as the number of threads increases, from 103 seconds with a single thread to 16 seconds with 32 threads (see also Figure 4.1, green line). Correspondingly, the speedup improves from 1.00× to 6.43×, demonstrating a much more scalable behavior compared to the non-flattened implementation (see also Figures 4.2 and 4.3, green lines). This substantial improvement highlights that flattening arrays such as `tzt`, `amp`, and `amp1` enhances data locality and cache utilization, reducing memory-bound stalls and allowing threads to operate more efficiently. Even at higher thread counts, performance continues to scale well, indicating that memory bandwidth limitations are alleviated by this layout optimization.

[3] In this implementation, the inner loop iterates over `iz`, while the outermost loop iterates over `ix`.

#threads	Elapsed Time (s)	Speedup
1	103	1.00
2	64	1.60
4	42	2.45
8	30	3.43
16	22	4.68
32	16	6.43

Table 4.3: Execution time and corresponding speedup of the migration kernel using flattened memory allocation across different thread counts.

While flattening slightly alters the order of memory accesses and arithmetic operations, the resulting numerical error, that is, RMSE, increases only modestly, from `0.000109545` to `0.000154919`, which remains negligible relative to the overall computational results. Therefore, the performance benefits of this optimization outweigh the minimal impact on accuracy, making flattened memory allocation an effective strategy for improving parallel scalability of the migration kernel.

Before further analyzing the parallel performance of the optimized implementation, it is important to discuss a moderate single-thread slowdown observed in the flattened version compared to the original 2D layout (the runtime increases from 86 seconds in Table 4.2 to 103 seconds in Table 4.3). A likely explanation lies in the change in memory access behavior introduced by flattening the array from a pointer-based `[ix][iz]` structure to a contiguous indexing scheme of the form `[ix*nzt + iz]`. In the original layout, each lateral slice is allocated separately, which can confine the effective working set seen by a single thread to smaller memory regions that may fit more readily in lower cache levels and be more easily exploited by the hardware prefetcher despite the presence of pointer indirection. In contrast, the flattened representation removes this indirection but forces all slices into a single contiguous block, increasing cache pressure and slightly enlarging the working set footprint in the single-thread case, while also introducing a minor arithmetic overhead due to index computation. This trade-off is nevertheless beneficial for multi-thread execution, as the contiguous layout reduces NUMA-unfriendly pointer chasing and enables more predictable memory access patterns, ultimately improving parallel scalability.

4.7 Load Imbalance

Load imbalance occurs when computational work is unevenly distributed among processing units, causing some resources to remain idle while others are still executing. This imbalance leads to suboptimal parallel efficiency and increased overall execution time. In high-performance computing applications, load imbalance often arises from irregular data access patterns, conditional branches, or nonuniform problem decomposition. Mitigating load imbalance is therefore essential to fully exploit available parallelism and achieve scalable performance.

In our case (see Figure 4.4), Intel VTune analysis shows that when the number of threads is set to, for example, 16, in the flattened version, the CPU utilization histogram exhibits a bell-shaped distribution[4]. This indicates that the application does not sustain full concurrency throughout its execution. The reduced utilization can be attributed primarily to load imbalance and synchronization overhead within the OpenMP parallel regions. As threads complete their assigned work at different times, some become idle or enter spin states while waiting at synchronization points, such as implicit

[4] Although spin and overhead times first appear in the caption of Figure 4.4, we provide formal definitions here for all time metrics used throughout this section to clarify their meaning and relationships. CPU time is the total execution time summed across all threads. Elapsed time is the wall-clock time for program execution. Spin time refers to the time a thread spends actively waiting (consuming CPU cycles in a busy-wait loop) rather than performing useful work, typically at barriers or other synchronization points. Overhead time refers to the time spent by the system on activities that do not contribute directly to useful computation, such as thread scheduling, synchronization management, and other runtime bookkeeping.

or explicit barriers. This behavior is further illustrated in Figure 4.5 (CPU time is shown in brown, while spin and overhead time is shown in orange), where only four threads are shown for clarity, although the reported "CPU Utilization" metric represents the cumulative utilization across all 16 threads.

To mitigate load imbalance in parallel executions, several strategies can be employed, including optimized scheduling policies, randomized iteration ordering, and task-based parallelism. These techniques aim to distribute the computational workload more evenly across threads, thereby improving resource utilization and overall parallel efficiency. In this work, we focus specifically on evaluating different OpenMP scheduling policies.

Note that, in this work, the `collapse(2)` clause cannot be applied because the trip count of the inner loop depends on the index of the outer loop (see Listing 2). Consequently, the loop nest is not perfectly rectangular, preventing safe collapse of the two loops. As a result, the amount of work associated with each outer-loop iteration varies significantly. When such a loop nest is parallelized, this irregular workload distribution can lead to load imbalance, as some threads are assigned iterations with short inner loops, while others process substantially longer ones.

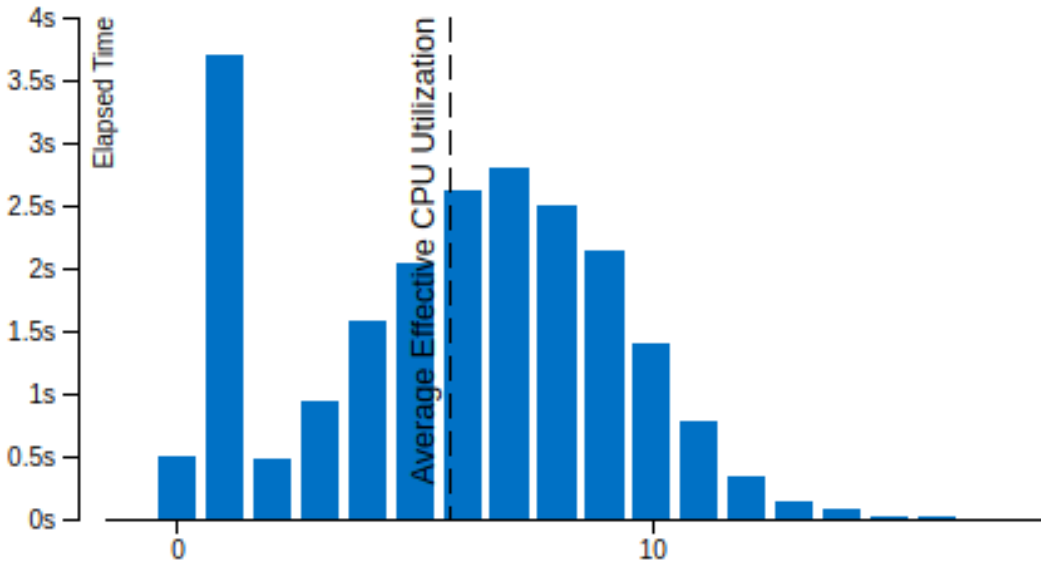

Figure 4.4: This histogram shows the percentage of wall time during which a given number of CPUs were running simultaneously. The bar shown at 0 on the x-axis represents the Idle CPU Utilization, which includes Spin and Overhead times.

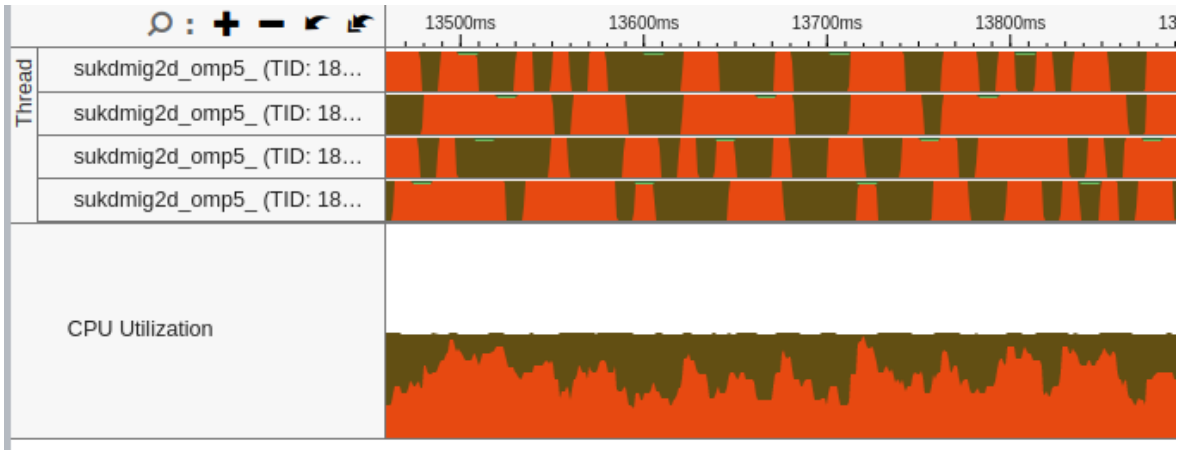

Figure 4.5: Intel VTune thread-level performance analysis.

4.7.1 Scheduling Policies

To further investigate potential load imbalance and synchronization effects, different OpenMP scheduling policies were evaluated. In particular, we analyzed the impact of `static` and `dynamic` scheduling with varying chunk sizes on the performance of the most time-consuming loops.

Different chunk sizes N were evaluated using `static` scheduling, as shown in Figure 4.6. For 32 threads, the best speedup is obtained with a chunk size of $N = 3$, while larger chunk sizes do not provide additional performance benefits. This suggests that moderate granularity offers a good balance between load distribution and scheduling overhead. Note that, for all tested chunk sizes: (*i*) The `nowait` clause was applied to eliminate the implicit barrier at the end of the parallel loop. (*ii*) The measured runtimes and speedups are essentially identical when using either `static` or `dynamic` scheduling (the scheduling choice is not a bottleneck).

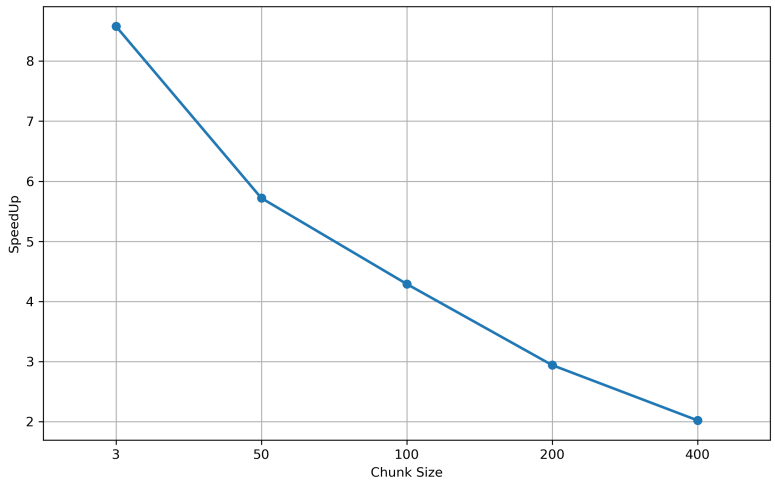

Figure 4.6: Speedup obtained with `schedule(static, N)` for different chunk sizes `N` using 32 threads.

To further analyze synchronization effects, spin time was measured for the unscheduled baseline

configuration, as well as for the scheduled versions, while varying the number of threads (see Tables 4.4, 4.5, and 4.6). As shown in Figure 4.7, both scheduling strategies significantly reduce spin time compared to the baseline configuration. However, the relative spin time remains nearly identical for `schedule(static,3)` and `schedule(dynamic,3)` across all tested thread counts. Note that, in Figure 4.7, since the total runtime differs across scheduling policies, spin time is reported as a percentage of total CPU time to enable a fair comparison.

#threads	CPU Time (s)	Spin Time (s)
1		
2	120.44	18.359
4	160.82	56.124
8	217.07	105.208
16	288.4	159.258
32	418.8	253.086

Table 4.4: CPU utilization without explicit scheduling optimization (baseline case).

#threads	CPU Time (s)	Spin Time (s)
1		
2	106.82	5.07
4	116.19	11.84
8	137.88	26.769
16	193.05	64.907
32	302.12	139.471

Table 4.5: CPU utilization using OpenMP scheduling for `(static,3) nowait`

#threads	CPU Time (s)	Spin Time (s)
1		
2	107.5	4.65
4	116.75	10.76
8	141.37	28.138
16	191.88	61.578
32	311.81	137.84

Table 4.6: CPU utilization using OpenMP scheduling for `(dynamic,3), nowait`

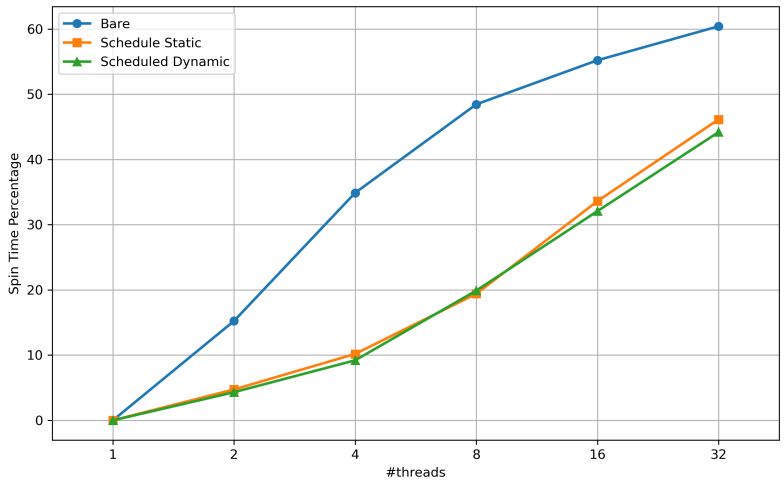

Figure 4.7: Spin time as a function of thread count for `schedule(static,3)` and `schedule(dynamic,3)`.

To further assess the impact of scheduling on workload distribution, the CPU utilization histogram was analyzed again after applying explicit OpenMP scheduling. After introducing explicit scheduling, the CPU utilization histogram changes markedly (see Figure 4.8). The distribution becomes almost flat, indicating a much more uniform utilization across all threads. This flattening of the histogram demonstrates that the workload is now distributed more evenly among the available cores, reducing idle time and improving overall parallel efficiency.

Comparing the histograms before and after scheduling clearly highlights the effect of the scheduling strategy. The transition from a bell-shaped to a nearly uniform distribution provides visual and quantitative evidence that explicit scheduling improves load balance and leads to better resource utilization.

As a final performance assessment of this section, Tables 4.7 summarizes the execution time, speedup, and corresponding parallel efficiency obtained for the flattened implementation using `schedule(static,3) nowait` as the number of threads increases. The runtime decreases steadily from 103 seconds in the single-thread configuration to 12 seconds with 32 threads (see Figure 4.1, red line), yielding a maximum speedup of 8.58×. Although performance improves consistently with increasing concurrency (see Figures 4.2 and 4.3, red lines), the scaling remains sublinear. In particular, the parallel efficiency, defined as the ratio between speedup and thread count, shows a progressive decline as more threads are employed: while high efficiency is achieved at low concurrency levels (92% with 2 threads and 78% with 4 threads), the marginal benefit of additional threads decreases beyond 8 threads, with efficiency dropping to 61%, 42%, and 26% at 8, 16, and 32 threads, respectively. This trend indicates that the scalability of the application becomes increasingly constrained in the many-thread regime; nevertheless, the overall performance gains justify focusing the remainder of the thesis on the analysis of this final optimized variant.

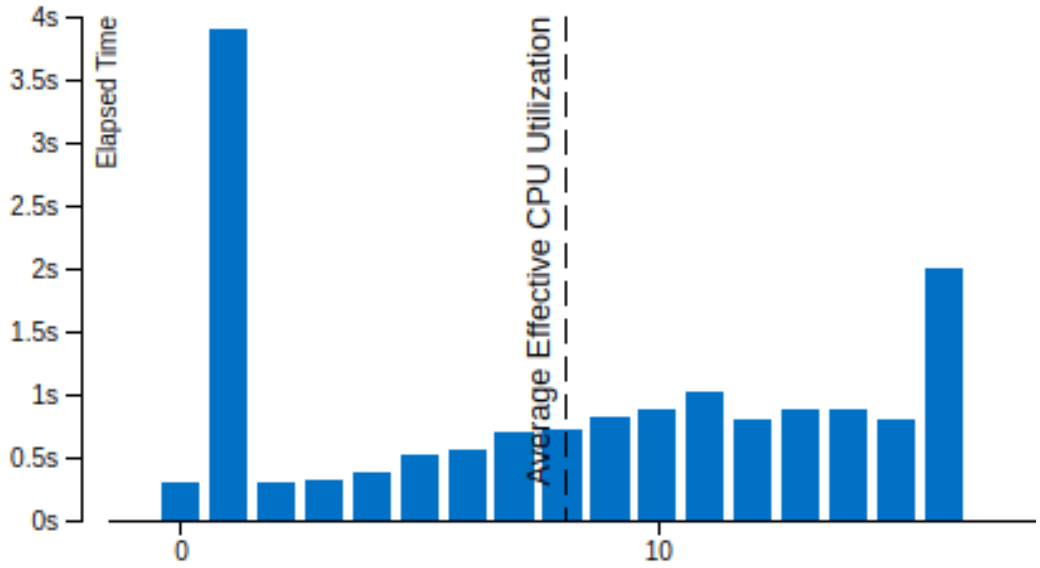

Figure 4.8: Histogram of CPU utilization for 16 threads using `schedule(static,3) nowait`

#threads	Elapsed Time (s)	SpeedUp	Efficiency [%]
1	103	1.00	100
2	56	1.84	92
4	33	3.12	78
8	21	4.90	61
16	15	6.86	42
32	12	8.58	27

Table 4.7: Speedup and parallel efficiency for the final code variant at increasing thread counts.

The observed sublinear behavior and the associated reduction in parallel efficiency at higher thread counts can be explained by several factors discussed throughout this work. First, as shown by the roofline analysis in Chapter 3, the most time-consuming kernels are memory-bandwidth bound even in single-thread execution; as more threads compete for the same memory bandwidth, the marginal gain per thread decreases. Second, the load imbalance discussed in this Section, evidenced by the bell-shaped CPU utilization histogram and the growing spin time in Table 4.4, causes some threads to idle at synchronization points while others are still computing. Third, the implicit barriers between the staged `omp for` constructs introduce synchronization overhead that grows with thread count, as reflected in the increasing spin time across Tables 4.4-4.6. Finally, a small residual serial fraction in the setup computations (`izt0`, `iz0`, `nzp` per iteration) places an upper bound on achievable speedup consistent with Amdahl's law.

4.8 Energy Considerations

Energy consumption has become a critical metric in modern computing, alongside traditional performance indicators such as execution time and scalability. As high-performance and large-scale computing systems grow in complexity and power density, optimizing code only for speed is no longer sufficient; it is equally important to understand how efficiently computational resources convert

electrical power into useful work. Energy measurement at the application level provides insight into the runtime behavior of a program, revealing how algorithmic choices, parallelization strategies, memory access patterns, and hardware utilization affect total energy usage. By analyzing energy together with performance metrics, it is possible to identify trade-offs between time-to-solution and power draw, and to guide optimizations toward solutions that are not only faster but also more sustainable and cost-effective.

In addition to raw performance metrics, this work evaluates computational efficiency through both time-to-solution (TTS) and energy-to-solution (ETS). While TTS measures how quickly a given implementation completes its workload, ETS captures the total energy required to reach the same result. These two metrics are not always aligned: a faster execution may consume more power, while a slightly slower configuration can sometimes be more energy efficient. Therefore, a combined analysis of TTS and ETS provides a more comprehensive view of code behavior and system utilization. The goal of this study is to compare different implementations and configurations using both metrics in order to identify the most effective trade-off and determine the overall better solution.

To perform a combined evaluation of performance and energy efficiency, we use the Energy–Delay Product (EDP), defined as:

$$\mathrm{EDP} = \mathrm{ETS} \times \mathrm{TTS}. \tag{4.1}$$

This metric balances runtime and energy consumption and is commonly used to compare overall computational efficiency across different configurations.

The energy analysis reported in Table 4.8 presents the ETS, the TTS, and their product (ETS×TTS), corresponding to the EDP, for different thread counts. As the number of threads increases, both execution time and total energy consumption decrease significantly compared to the single-thread case. It is worth noting that while ETS stabilizes at 3.05 kJ from 16 to 32 threads, the continued reduction in execution time leads to a further improvement in the EDP. Overall, the results indicate that increasing parallelism not only improves performance but also enhances energy efficiency, with 32 threads providing the best combined energy–performance trade-off among the tested configurations.

#threads	ETS (KJ)	TTS (s)	ETSxTTS (KJxs)
1	14.86	103	1530.58
2	8.50	56	476.00
4	5.21	33	171.93
8	3.64	21	76.44
16	3.05	15	45.75
32	3.05	12	36.60

Table 4.8: Energy consumption and EDP values for the `schedule(static, 3) nowait` version.

The ETS reported in this work corresponds to the sum of CPU and DRAM energy consumption, averaged over five runs, and was measured using the Cineca energy monitoring tool, `Cinemon`. `Cinemon` leverages Intel RAPL (Running Average Power Limit) counters to track energy usage of CPUs and memory, providing fine-grained measurements with minimal overhead. In this study, the CPU sampling period was set to 100 ms by configuring `RAPL_PERIOD=0.1`, ensuring accurate and frequent energy readings during program execution. These measurements allow for a reliable comparison of energy efficiency across different computational setups and optimization strategies.

In the present study, GPU acceleration was not required, and all analyses were therefore performed using CPU resources only. The Booster partition was selected primarily to ensure compatibility with potential future developments involving GPU accelerators. However, experiments focused on pure OpenMP execution, including strong-scaling and energy-consumption analyses, were conducted on the DCGP partition, whose CPU-only configuration is better suited for such evaluations (see Appendix C).

4.9 Weak Scaling

In parallel computing, weak scaling refers to the study of how the runtime of a parallel algorithm changes as the problem size and the number of processing units are increased proportionally, such that the work per processor remains roughly constant. The main purpose of weak scaling is to evaluate how effectively a parallel implementation can handle larger problem sizes without being limited by communication overhead, memory bandwidth, or other hardware bottlenecks. This is particularly important for large-scale scientific computations, where increasing the dataset size is often necessary to improve resolution or coverage, and the goal is to ensure that the parallel algorithm can accommodate these larger datasets efficiently.

In a shared-memory parallel model such as OpenMP, weak scaling is achieved by keeping the work per thread constant while increasing the total problem size proportionally to the number of threads. In other words, if one thread processes a problem of size N, then two threads should process a problem of size $2N$, four threads a problem of size $4N$, eight threads a problem of size $8N$, and so on. Under ideal weak scaling conditions, because each thread performs the same amount of work regardless of the total number of threads, the overall runtime should remain constant as the number of threads increases.

In this work, weak scaling experiments were conducted for the 2D Kirchhoff migration algorithm, implemented using OpenMP. The vertical grid size was fixed at $nz = 601$, while the lateral grid nx and the number of traces increased with the survey length. The number of OpenMP threads was doubled each time the survey length was doubled to keep the work per thread approximately constant. In particular, the survey length, measured in kilometers (km), was doubled at each step because the data size scales proportionally with it, resulting in approximately twice the amount of data to be processed. The measured runtimes are shown in Table 4.9.

L (km)	nx	#threads	#traces	TTS (s)	TTS/L
2.5	100	1	2162	23	9.2
5	200	2	9270	67	13.4
10	400	4	27270	193	19.3
20	800	8	63270	371	18.55
40	1600	16	135270	1032	25.8
80	3200	32	279270	2205	27.5

Table 4.9: Weak scaling setup and measured runtime for the 2D Kirchhoff migration code.

The weak scaling results reported in Table 4.9 may at first glance suggest poor weak scaling, as the total TTS increases significantly with survey length, even though the number of threads is doubled at each step. However, this behavior is primarily due to the algorithmic complexity of the 2D Kirchhoff migration method, since the computational cost grows with both the lateral grid size and the number of traces, which increase with survey length. To better assess the weak scaling behavior independently of this inherent complexity, the runtime was normalized by the survey length (TTS/L). This normalization provides a more meaningful metric for evaluating scalability. The results indicate that, although the runtime per kilometer is not perfectly constant, the observed growth is largely attributable to the intrinsic computational characteristics of the algorithm rather than to limitations of the OpenMP parallelization. Overall, the implementation remains suitable for processing larger datasets with increasing thread counts while maintaining acceptable efficiency (see Figure 4.9).

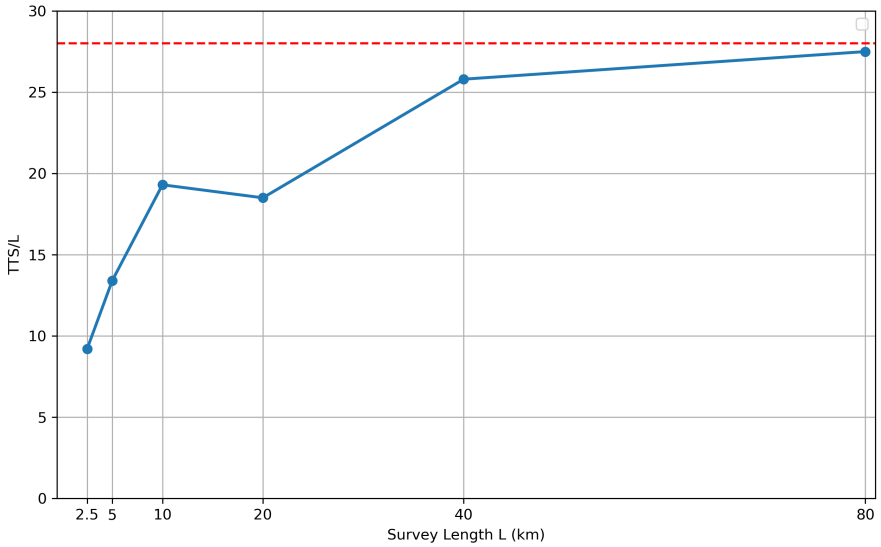

Figure 4.9: Normalized runtime (TTS/L) versus survey length for the weak scaling study. The trend highlights that the increase in total runtime is largely driven by the algorithmic complexity of the migration method rather than by limitations in OpenMP scalability.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This thesis investigated performance optimization of a PSDM kernel, a computationally intensive algorithm widely used in seismic imaging to accurately position subsurface reflectors. Due to its deeply nested loops and irregular memory access patterns, PSDM presents significant computational challenges that make performance optimization essential for large-scale seismic processing. The work focused on improving the efficiency of the kernel through enhanced SIMD vectorization and OpenMP-based shared-memory parallelization.

The study first analyzed the most computationally demanding loops to identify opportunities for improving vectorization. Loop restructuring, reduction of data dependencies, and improved memory access patterns were applied to increase vectorization efficiency and better utilize modern CPU vector units. However, because loop bounds vary across iterations, relying solely on SIMD optimization limits both workload balance and scalability. In such situations, combining outer-loop parallelism with inner-loop SIMD provides better control over workload distribution and vectorization efficiency. Based on these considerations, an optimized OpenMP implementation was introduced to exploit parallelism in the outer loops while preserving the benefits of the enhanced SIMD-vectorized kernels. Particular attention was given to workload distribution, scheduling strategies, and memory access behavior to reduce thread contention and maximize processor utilization.

The effectiveness of the optimized implementation was evaluated through strong and weak scaling experiments to analyze performance as the number of processing cores and problem size varied. In addition, energy consumption was monitored to assess the trade-offs between performance improvements and energy efficiency on multi-core systems. The experimental results show that the optimized implementation achieves significant performance improvements over the baseline version while maintaining favorable energy characteristics. Overall, this work demonstrates that combining improved vectorization with scalable thread-level parallelism and systematic performance analysis can substantially accelerate PSDM workloads on modern high-performance computing architectures.

5.2 Future Work

Although the optimizations presented in this thesis significantly improved the performance of the PSDM kernel through enhanced vectorization and pure OpenMP parallelization, several opportunities remain for further improvements and extensions.

One potential direction is the use of more advanced vectorization techniques. While this work relied mainly on compiler auto-vectorization, future work could explore explicit SIMD programming using intrinsic functions to gain finer control over vector operations and memory access patterns. This may further improve SIMD efficiency, particularly in complex loops where automatic vectorization is

limited. Additionally, since the PSDM algorithm involves irregular loop bounds and varying workloads across iterations, investigating dynamic load balancing strategies, such as more advanced OpenMP scheduling policies or task-based parallelism, could improve workload distribution and reduce thread idle time.

Another natural direction is the exploration of hybrid parallel programming models that combine distributed-memory and shared-memory paradigms. Integrating MPI with OpenMP would allow the PSDM implementation to scale beyond a single compute node and efficiently utilize large high-performance computing clusters, enabling the processing of significantly larger seismic datasets while maintaining high computational efficiency.

Finally, heterogeneous computing represents a promising avenue for further acceleration. Offloading computationally intensive kernels to hardware accelerators such as GPUs using programming models like CUDA, OpenACC, or OpenMP target offloading could provide substantial performance gains, particularly for the highly parallel loops present in PSDM algorithms.

Appendix A

PSDM Fundamentals Using the Kirchhoff Equation

In this Appendix, we recall the main concepts of the PSDM by using the Kirchhoff equation (for details, see [1]).

The Kirchhoff PSDM is based on diffraction summation method. The method search for the input data in the $x-t$ plane for energy (seismic data survey) that would have resulted if a diffracting source (the so-called Huygen's secondary source) were located at a particular point in the output $x-z$ plane. This search is conducted by summing all the amplitudes in the $x-t$ plane along the diffraction curve corresponding to Huygen's secondary source. This summation is mapped onto the point in the $x-z$ plane [1].

The migration scheme is based on the 2D shot pre-stack migration. The migration methodology begins with an initial weighted sum of all amplitudes on shot point by using Kirchhoff integral, and a second summation corresponding to the multiple coverage redundancy (from CIGs), in order to obtain a depth migrated stack (Figure A.1). After each new migration, the velocity model is updated and a new traveltime table is computed that will be used for the next migration (see Appendix B).

The Kirchhoff migration is a method of seismic migration that uses the integral form (Kirchhoff equation) of the wave equation. All methods of seismic migration involve the back propagation (or continuation) of the seismic wave field from the region where it was measured (Earth's surface or along a borehole) into the region to be imaged. In Kirchhoff migration, this is done by using the Kirchhoff integral representation of a field at a given point as a (weighted) superposition of waves propagating from adjacent points and times. Continuation of the wave field requires a background model of seismic velocity, which is usually a model of constant or smoothly varying velocity. Because of the integral form of Kirchhoff migration, its implementation reduces to stacking the data along curves that trace the arrival time of energy scattered by image points in the earth.

The method assumes that every point in the subsurface is a scatter point which scatters waves coming from a source point S. The wavefield collected at the Earth's surface is a superposition of waves coming from every scatter point. In migration, we propagate the surface wavefield back to subsurface scatter points and collect the wavefield at $t = 0$ to get the image of subsurface structure.

Let $P(x_s, x_r, z = 0)$ be the wavefield at the Earth's surface. The wavefield at every scatter point is :

$$P(x, z, t) = \int A \left(\frac{\partial}{\partial t}\right)^{1/2} P\left(x_s, x_r, z = 0, t + \frac{r_s}{v_d} + \frac{r_r}{v_u}\right) dx_s dx_r, \tag{A.1}$$

where, x_s and x_r are the x coordinates of source and receiver, respectively; r_s and r_r are the travel distances from source to scatter point and from scatter point to receiver, respectively; v_d and v_u are

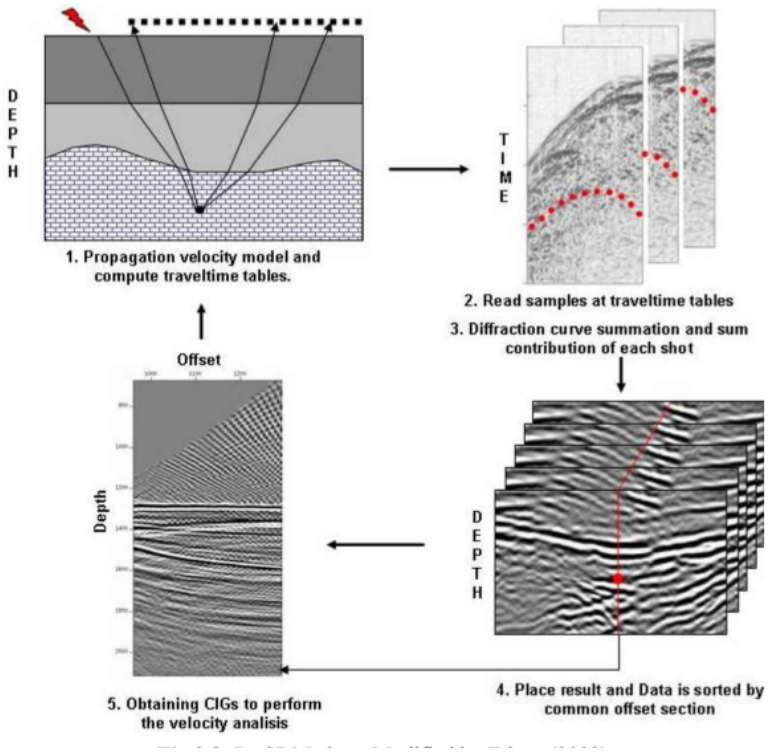

Figure A.1: PSDM chart (Vargas-Cordero, 2009).

the velocities of downgoing and upgoing raypaths, respectively. And,

$$A = \frac{\cos(\theta_s)\cos(\theta_r)}{\sqrt{v_d v_u r_s r_r}}, \tag{A.2}$$

is the amplitude scale factor. So the image of the subsurface is represented by:

$$P(x,z) = \int A \left(\frac{\partial}{\partial t}\right)^{1/2} P\left(x_s, x_r, z=0, \frac{r_s}{v_d} + \frac{r_r}{v_u}\right) dx_s dx_r. \tag{A.3}$$

Appendix B

ISTRICI Processing Flowcharts

ISTRICI is a software package designed to support seismic depth imaging and velocity model building through an iterative PSDM approach. The framework introduces three alternative strategies for organizing CIGs, enabling tailored velocity analysis according to survey objectives, structural targets, and data characteristics. This adaptive organization enhances the quality of the final imaging results while reducing the time required to evaluate the r-parameters of the residual semblance.

The methodology has been structured around three representative scenarios: (*i*) *INTER workflow*: When laterally continuous reflectors, e.g., the seafloor, are present, their coherence along the seismic profile enables semi-automated picking within residual semblance panels (see Figure B.1). (*ii*) *CIG workflow*: When a reliable geological interpretation of the seismic section is available, velocity updating can be constrained along the interpreted reflectors, linking model refinement directly to structural horizons (see Figure B.2). (*iii*) *TRAD workflow*: In structurally complex environments or where reflectors lack lateral continuity, velocity analysis is performed as a function of depth rather than following a layer-stripping strategy (see Figure B.3).

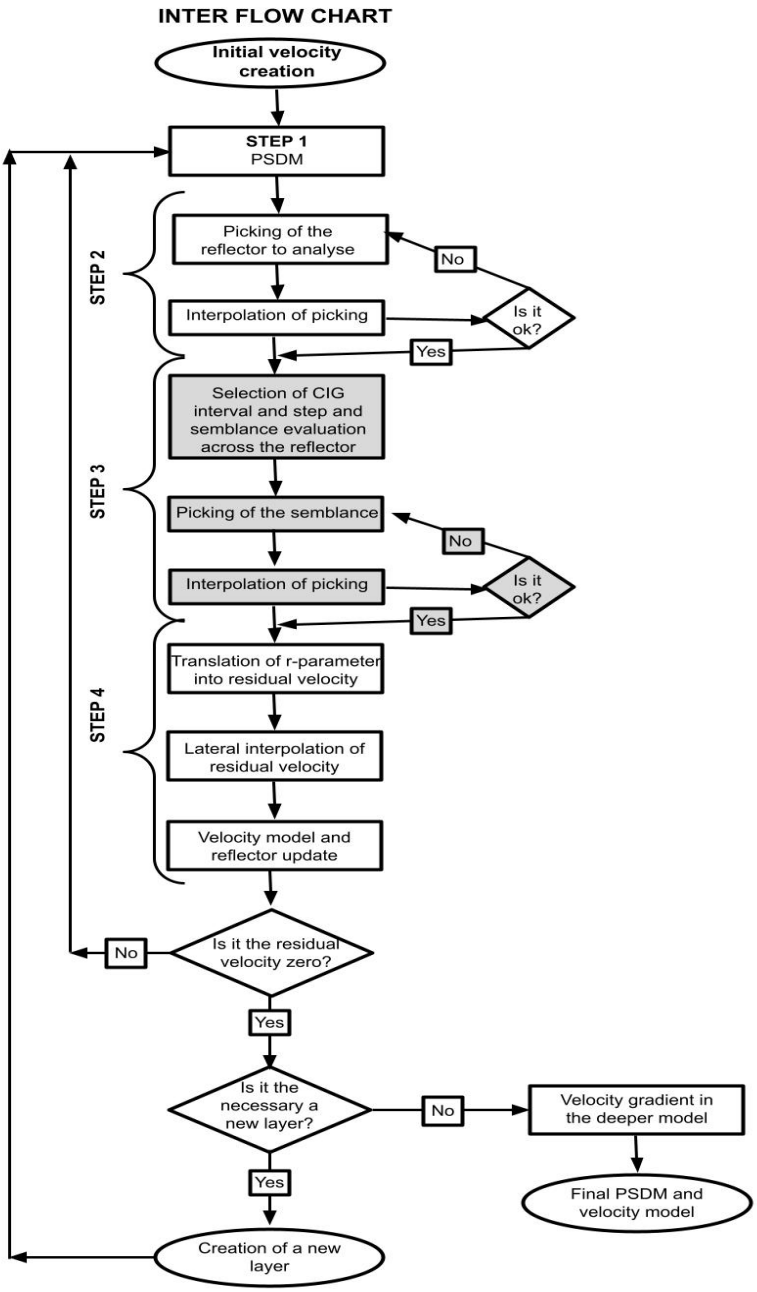

Figure B.1: Flowchart describing the INTER workflow.

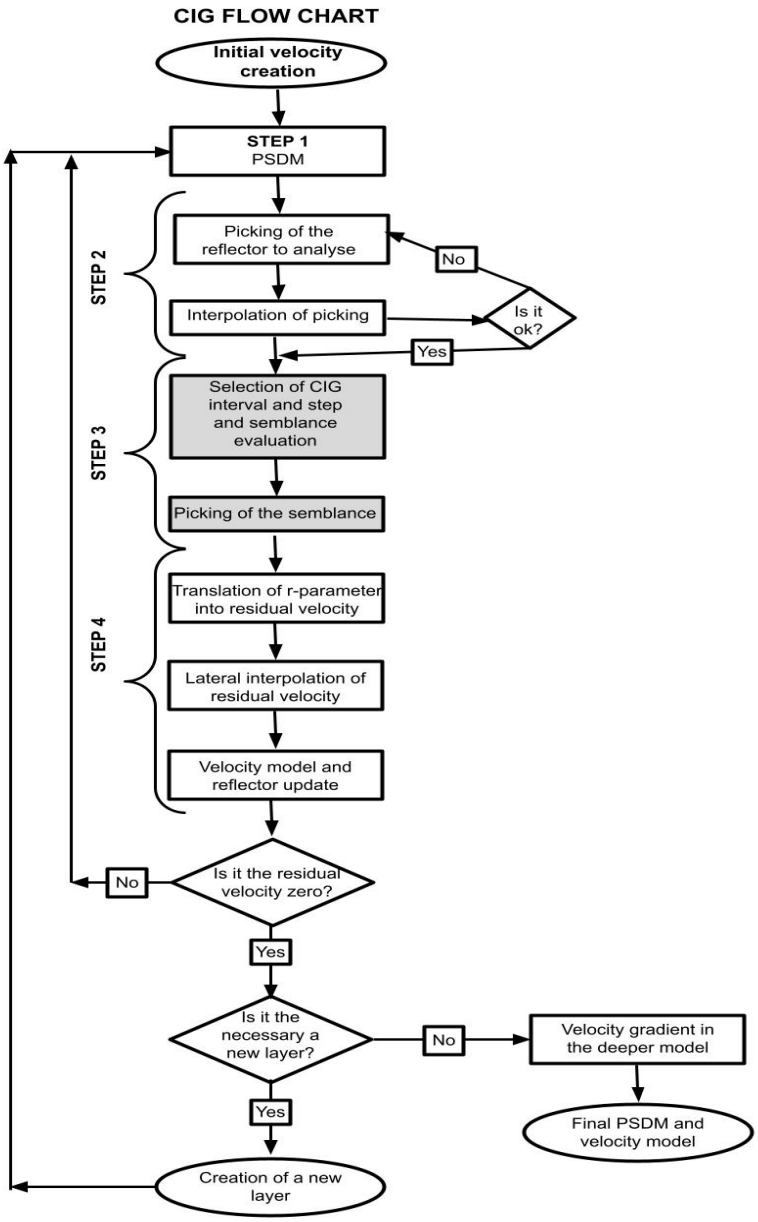

Figure B.2: Flowchart describing CIG workflow.

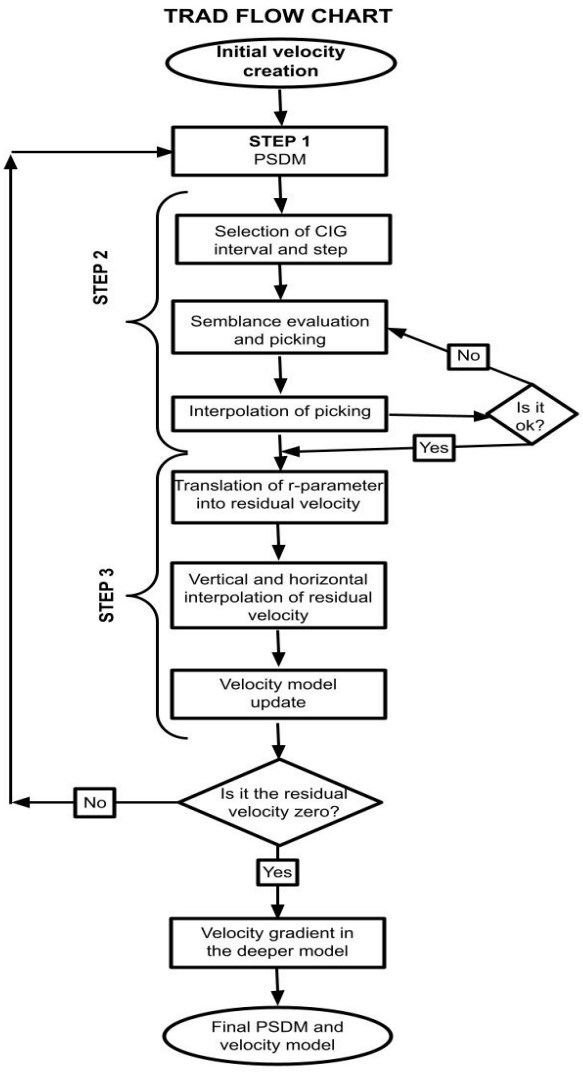

Figure B.3: Flowchart describing TRAD workflow.

Appendix C

DCGP's Results

The strong-scaling behavior of the pure OpenMP implementation was evaluated on the Leonardo DCGP partition. The results, reported in Table C.1, show a substantial reduction in TTS as the number of OpenMP threads increases from 1 to 16, with a speedup of approximately 6×. Beyond this point, the scaling efficiency progressively decreases, and only marginal performance gains are observed when increasing the thread count to 32. When using 64 threads, the execution time increases, indicating a clear loss of scalability.

The increase in TTS observed when moving from 32 to 64 threads can be attributed not only to NUMA effects and increased memory contention, but also to possible CPU frequency scaling. On modern Intel Xeon processors, the simultaneous activation of a large number of cores, particularly when executing `AVX-512` vector instructions, may lead to a reduction in operating frequency in order to remain within the thermal design power limits. At high thread counts on the DCGP partition, this effect may therefore contribute to the observed performance degradation.

Threads	TTS	ETS_cin	ETS_beo	ETS_cinxTTS	ETS_beoxTTS	SpeedUp
1	73	29.57	40.26	2158.61	2938.98	1.00
2	40	16.05	24.86	642.00	994.40	1.82
4	24	9.85	17.57	236.40	421.68	3.04
8	15	6.64	14.00	99.60	210.00	4.86
16	12	5.38	12.51	64.56	150.12	6.08
32	11	5.12	12.00	56.32	132.00	6.63
64	14	7.00	14.47	98.00	202.58	5.21

Table C.1: Energy and performance results for the `schedule(static,3) nowait` version on the DCGP partition. Energy is reported in kJ, time in seconds, and ETS values are averaged over five runs.

A similar trend is observed for the ETS. The energy consumption decreases significantly up to 16–32 threads, reaching a minimum around 32 threads, after which it increases when further threads are employed (see Figure C.1)[1]. This behavior suggests that additional parallelism no longer compensates for the increased power consumption and runtime overhead. This indicates that more cores do not necessarily lead to better energy efficiency. In general, the point of minimum energy consumption does not coincide with the point of best performance; however, in this case, they align at approximately 32 threads (see Figure C.2, for EDP).

[1]Energy measurements were obtained using two complementary approaches. First, the `Cinemon` tool was used to monitor energy consumption of individual devices on a node, including the CPU, GPU, and DRAM. Second, the Bull Energy Optimizer (BEO) estimates the energy consumption of the entire node. Using both `Cinemon` and BEO allows us to capture different perspectives on energy usage: `Cinemon` highlights the contribution of individual hardware components, while BEO provides a holistic, node-level estimate. In our experiments, BEO values are generally higher than `Cinemon` measurements because they include the energy of the full system, whereas `Cinemon` reports only the monitored devices.

Overall, these results indicate that, on the Leonardo DCGP partition, the optimal configuration for the analyzed OpenMP application lies between 16 and 32 threads, where both performance and energy efficiency are maximized.

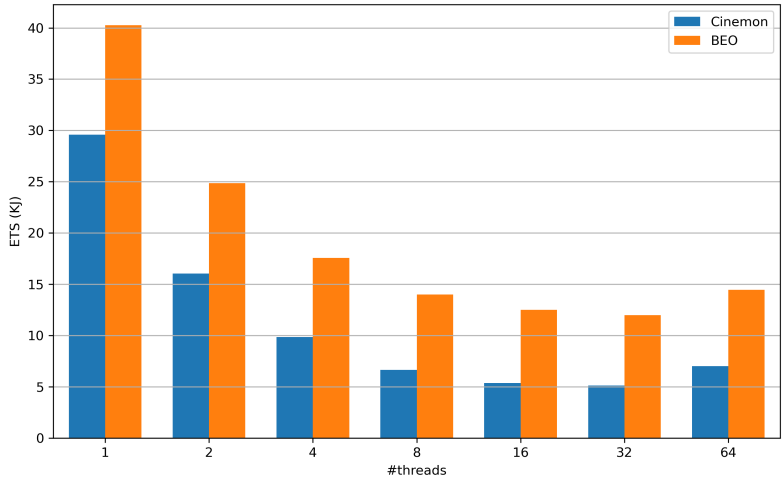

Figure C.1: Energy consumption (ETS) as a function of the number of threads, showing both `Cinemon` and BEO measurements. The systematic offset between them reflects the difference in measurement scope: BEO accounts for the full node, while `Cinemon` reports only the monitored devices.

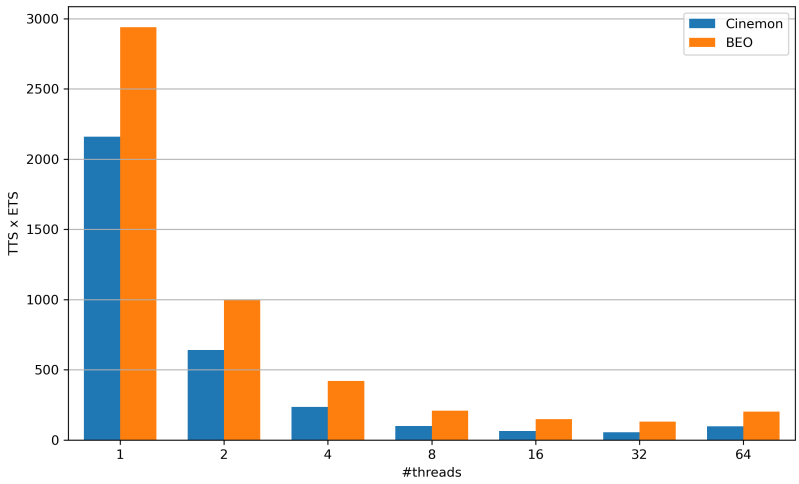

Figure C.2: ETS × TTS as a function of thread count, showing both `Cinemon` and BEO measurements.

Bibliography

[1] Yilmaz, O., 2001. *Seismic Data Analysis: Processing, Inversion and Interpretation of Seismic Data*, second ed. Society of Exploration Geophysicists, Tulsa, Oklahoma.

[2] Giustiniani, M., Tinivella, U., Nicolich, N., 2022. *Imaging subsurface structures using wave equation datuming advanced seismic techniques*. In: Bell, Rebecca, Iacopini, David, Vardy, Mark (Eds.), Interpreting Subsurface Seismic Data, vol. 2022. Elsevier.

[3] Tinivella, U., Giustiniani, M., 2023. *ISTRICI – Tools for facilitating seismic depth imaging and velocity analysis with seismic unix*, Computers & Geosciences.

[4] Vargas-Cordero, I.C., 2009. *Gas hydrate occurrence and morpho-structures along Chilean margin*. Dissertation, University of Trieste, Italy, 138 pp.

[5] Tinivella, U., Loreto, M.F., Accaino, F., 2009. *Regional versus Detailed Velocity Analysis to Quantify Hydrate and Free Gas in Marine Sediments: the South Shetland Margin Case Study*, vol. 319. Geological Society Special Publication.