Performance of OpenMP loop transformations for the acoustic wave stencil on GPUs

Jaime F. de Souza
Universidade Federal de São Carlos
Brazil
jaimefreire@estudante.ufscar.br

Letícia S.F. Machado
Universidade Federal de São Carlos
Brazil
suellenletici@gmail.com

Edson S. Gomi
Universidade de São Paulo
Brazil
gomi@usp.br

Claude Tadonki
Mines ParisTech / PSL
France
claude.tadonki@mines-paristech.fr

Simon McIntosh-Smith
University of Bristol
UK
S.McIntosh-Smith@bristol.ac.uk

Hermes Senger
Universidade Federal de São Carlos
Brazil
hermes@ufscar.br

In this work we evaluate the performance of unroll and tiling, two loop transformations introduced in OpenMP 5.1 and implemented in Clang 13 for GPUs. Experiments on a common seismic kernel demonstrate performance gains on three GPU architectures.

1 OPENMP OFFLOADING FOR GPUS

GPUs can dramatically accelerate a large variety of computer applications. With the support for heterogeneous architectures early introduced in OpenMP 4.0 and updated in OpenMP 4.5, OpenMP is becoming an interesting choice for supporting performance and productivity for the development of HPC applications. OpenMP 5.1 introduced *unroll* and *tiling* loop transformations [4], whose code offloading for such GPUs was first supported in Clang 13. In the present work, we evaluate the performance of code offloading for these loop transformations for a seismic application kernel running on representatives of three GPU architectures.

2 THE ACOUSTIC WAVE KERNEL

The simulation of acoustic waves in multi-material domains is the kernel of seismic applications such as in full-waveform inversion (FWI) and reverse-time migration (RTM), used by industry for the characterization of reservoirs of hydrocarbons, such as oil and gas. The propagation of acoustic waves in a constant density medium can be modeled as follows:

$$\frac{1}{v_p^2}\frac{\partial^2 p(\mathbf{x},\mathbf{y},\mathbf{z},t)}{\partial t^2} - \nabla^2 p(\mathbf{x},\mathbf{y},\mathbf{z},t) = f(\mathbf{x},\mathbf{y},\mathbf{z},t) \tag{1}$$

where v_p is the wave propagation velocity, $p(\mathbf{x},\mathbf{y},\mathbf{z},t)$ is the pressure field, and $f(x,y,z,t)$ is an external body force (i.e. source). To solve this PDE by the finite differences method, we divide the wave field into a regular 3D grid of points equally spaced from each other by distances Δx, Δy, and Δz. Δt is the time increment (ranging from $n = 0$ to a defined number of time steps). By using second-order central differences and isolating the term $p_{i,j,k}^{(n+1)}$, we get the following discretized equation (for 2nd-spatial order):

$$p_{i,j,k}^{(n+1)} = 2p_{i,j,k}^{(n)} - p_{i,j,k}^{(n-1)} + 2\Delta t^2 . v^2 \Big(\frac{p_{i+1,j,k}^{(n)} - 2p_{i,j,k}^{(n)} + p_{i-1,j,k}^{(n)}}{\Delta x^2} + \frac{p_{i,j+1,k}^{(n)} - 2p_{i,j,k}^{(n)} + p_{i,j-1,k}^{(n)}}{\Delta y^2} + \frac{p_{i,j,k+1}^{(n)} - 2p_{i,j,k}^{(n)} + p_{i,j,k-1}^{(n)}}{\Delta z^2} \Big) \tag{2}$$

Each point $p_{i,j,k}^{(n+1)}$ (Fig. 1) is calculated as a function of its neighbors (in red) and its own value (in blue) at the current instant n and at the previous timer step $n-1$. A major performance issue with stencils is their bad data locality, which leads to memory inefficiency.

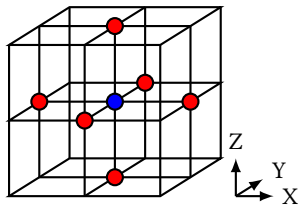

Figure 1: A 7-point (2nd spatial order) stencil.

3 PERFORMANCE EVALUATION

We tested representatives of three GPU architectures: RTX 2080 Super (Turing), V100 (Volta) and A100 (Ampere), described in Table 1. The PDE was numerically solved using the finite difference method discretized with 2nd-, 8th-, and 16th spatial order (using 7-point, 25-point, and 49-point stencils, respectively). Our baseline code is a simplified version of `simwave` [2] implemented in C, and optimized using loop unroll and tiling supported by Clang 13.0 (LLVM project). We used the flags `-O3 -fPIC -ffast-math -fopenmp -fopenmp-version=51 -fopenmp-targets=nvptx64 -Xopenmp-target`. The underlying CUDA 11.0 and the software environment were configured in a Singularity container executing on a Slurm cluster. We used grids of 256^3 , 512^3 , and 1024^3 points in size, and with 400, 800 and 1600 time steps, respectively (enough to propagate the signal from the center to the borders).

3.1 Loop transformation strategies

The baseline code (Listing 1) merges the three loops into a single parallel iteration space before the distribute directive splits up the iterations between a league of teams. This baseline is used by state-of-the-art DSLs for seismic applications, such as Devito [3]. Collapsing parallel loops for execution on GPUs has two performance effects: the number of parallel work items to be scheduled increases, and memory access pattern can be improved [1]. All implementations copy the data from the host's memory to the device's memory before the kernel is launched several times during

Table 1: GPUs architecture specifications.

	RTX 2080 Super	V100	A100
GPU Architecture	Turing	Volta	Ampere
SMs	48	80	108
CUDA cores / SM	64	64	64
CUDA cores / GPU	3072	5120	6912
Peak FP64 TFLOPS	0.35	7.8	9.7
Peak FP32 TFLOPS	11.2	15.7	19.5
Memory Size	8 GB	32 GB	40 GB
Memory Bandwidth	496 GB/s	900 GB/s	1555 GB/s
Shared Memory / SM	64 KB	96 KB	164 KB
L2 Cache Size	4 MB	6 MB	40 MB

time-stepping loop (line 1 of Listing 1), and then copy the data back to the host after this loop executes.

```
1 for(int t = 0; t < time_steps; t++) {
2   #pragma omp target teams distribute parallel for \
      collapse(3)
3   for(int i = radius; i < d1 - radius; i++){
4     for(int j = radius; j < d2 - radius; j++){
5       for(int k = radius; k < d3 - radius; k++){
6         ...
7         for(int ir = 1; ir <= radius; ir++){
8           // stencil point calculation
```

Listing 1: The parallel strategy used as baseline.

Next, we unrolled the innermost loop that iterates over the stencil radius to compute one grid point.

```
1 for(int t = 0; t < time_steps; t++) {
2   #pragma omp target teams distribute parallel for \
      collapse(3)
3   for(int i = radius; i < d1 - radius; i++){
4     for(int j = radius; j < d2 - radius; j++){
5       for(int k = radius; k < d3 - radius; k++){
6         ...
7         #pragma omp unroll full
8         for(int ir = 1; ir <= radius; ir++){
9           // stencil point calculation
```

Listing 2: Unrolls the loop that computes the spatial order.

The next optimization (Listing 3) applies the `collapse` and `tile` in the spatial loops. This later clause breaks the multidimensional loops into *tiles*, increasing data locality and limiting the iteration space to be collapsed, which reduces the GPU's scheduler overhead. We tested different combinations of tile sizes, and the best tile sizes are shown in Table 2 for the 2nd spatial order and FP64 scenario).

```
1 for(int t = 0; t < time_steps; t++) {
2   #pragma omp target teams distribute parallel for \
      collapse(3)
3   #pragma omp tile sizes(BLOCK1,BLOCK2,BLOCK3)
4   for(int i = radius; i < d1 - radius; i++){
5     for(int j = radius; j < d2 - radius; j++){
6       for(int k = radius; k < d3 - radius; k++){
7         ...
8         for(int ir = 1; ir <= radius; ir++){
9           // stencil point calculation
```

Listing 3: Baseline code with addition of tile clause.

Finally, in Listing 4 we combine all the above optimizations.

Table 2: Tile sizes applied to 2nd spatial order with FP64.

GPU	Grid size	Best tile sizes		
		Axis 1	Axis 2	Axis 3
RTX 2080	256^3	32	1	4
	512^3	16	1	4
V100	256^3	1	1	4
	512^3	8	1	2
	1024^3	8	1	2
A100	256^3	2	2	4
	512^3	2	1	4
	1024^3	8	1	4

```
1 for(int t = 0; t < time_steps; t++) {
2   #pragma omp target teams distribute parallel for \
      collapse(3)
3   #pragma omp tile sizes(BLOCK1,BLOCK2,BLOCK3)
4   for(int i = radius; i < d1 - radius; i++){
5     for(int j = radius; j < d2 - radius; j++){
6       for(int k = radius; k < d3 - radius; k++){
7         ...
8         #pragma omp unroll full
9         for(int ir = 1; ir <= radius; ir++){
10          // stencil point calculation
```

Listing 4: Using tiling and unroll.

The performance results are presented for 2nd space order (Figure 2) , expressed in terms of the average number of grid points computed per second (Gpoints/s) [1].

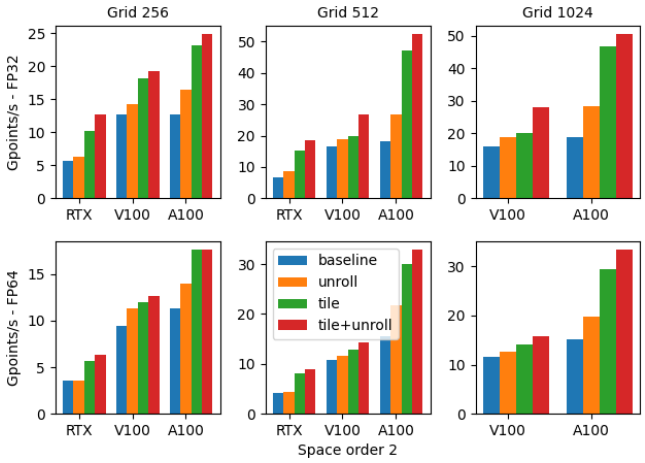

Figure 2: Gpoint/s for space order 2.

4 FINDINGS

As a general remark, both the unroll and tiling can yield significant improvements to the performance of the kernel evaluated on all GPUs evaluated. Performance gains ranged from 1.13x to 2.93x. In most scenarios, the best performance was achieved by combining unroll and tiling. As a last remark, the performance of tiling is highly sensitive to the choice of block size.

[1] Giga (10^9) grid points per second (Gpoint/s)

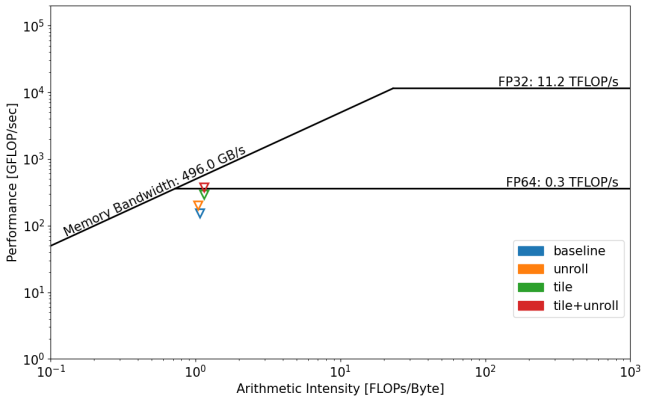

Figure 3: Roofline for FP32, space order 2 on RTX 2080 Super.

ACKNOWLEDGMENTS

The authors gratefully acknowledge sponsorship from Shell Brasil through the *ANP 20714-2 - Desenvolvimento de técnicas numéricas e software para problemas de inversão com aplicações em processamento sísmico* project at Universidade de São Paulo and the strategic importance of the support given by ANP through the R&D levy regulation. The authors also thank the support from São Paulo Research Foundation (FAPESP), grants #2022/00434-0 and #2019/26702-8.

REFERENCES

[1] Artem Chikin, Taylor Lloyd, José Nelson Amaral, Ettore Tiotto, and Muhammad Usman. 2019. Memory-Access-Aware Safety and Profitability Analysis for Transformation of Accelerator-Bound OpenMP Loops. 16, 3, Article 30 (jul 2019), 26 pages. https://doi.org/10.1145/3333060

[2] Jaime Freire de Souza, João Baptista Dias Moreira, Keith Jared Roberts, Roussian di Ramos Alves Gaioso, Edson Satoshi Gomi, Emílio Carlos Nelli Silva, and Hermes Senger. 2022. simwave-A Finite Difference Simulator for Acoustic Waves Propagation. *arXiv preprint arXiv:2201.05278* (2022).

[3] Fabio Luporini, Mathias Louboutin, Michael Lange, Navjot Kukreja, Philipp Witte, Jan Hückelheim, Charles Yount, Paul HJ Kelly, Felix J Herrmann, and Gerard J Gorman. 2020. Architecture and performance of Devito, a system for automated stencil computation. *ACM Transactions on Mathematical Software (TOMS)* 46, 1 (2020), 1–28.

[4] OpenMP Architecture Review Board. 2020. OpenMP API 5.1 Specification. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf. Accessed: 2022-08-02.