Integrating CUDA memory management mechanisms for domain decomposition of an acoustic wave kernel implemented in OpenMP

Yuri Nicolau Freire[1], **Hermes Senger** [1]

[1]Departamento de Computação – Universidade Federal de São Carlos (UFSCar)

yurinicolau@estudante.ufscar.br, hermes@ufscar.br

Abstract. *OpenMP is a well-known tool for parallelizing code in a directive-based programming model. While it has been extended to include support for offloading for devices such as GPUs, multi-gpu programming using data map directives requires redundant data allocation and non-intuitive data synchronization. This paper studies an alternative implementation of a CUDA-OpenMP hybrid kernel using native Unified Virtual Addressing memory pointers in an OpenMP target kernel.*

1. Introduction

OpenMP has been a staple in parallel programming for years thanks to its easy acceleration of existing algorithms in a directive-based programming model. This model was extended to encompass device offloading in version 4.0 with the introduction of the *target* directive, which aims to deliver seamless performance and simplicity seen in CPU-based parallelization for manycore devices such as GPUs, and is accompanied by the *target data map* pragma for easy data migration between host and Device. Although the *data map* directives show great results for single-gpu implementations, its use in multi-gpu domain decomposition lacks the ability to mapping multiple sections of an array into different devices. For this reason, more device memory than actually necessary is required, and handling synchronization and data consistency of halo regions between iterations of stencil calculations is error-prone. Alternatively, a developer may implement domain decomposition by explicitly allocating and copying memory via OpenMP's other functions. While this method may be more efficient, it demands more programming effort, requiring proper synchronization in order to produce correct results.

This paper shows how the use of Nvidia's Unified Virtual Addressing as a means of programming domain decomposition of stencil-style OpenMP GPU kernels, such as the acoustic wave simulation used in this paper, can make implementation easy and lower the required on-board device memory when taking advantage of a multi-gpu environment.

2. Acoustic wave simulation

Acoustic wave simulation models are commonly used in scientific applications such as seismic imaging, where its kernel is widely used in Full Waveform Inversions, a method that aims to reconstruct images of material layers underground by analyzing simulated data against real-world measured values. The wave propagation can be modeled by the elastic wave equation, a second-order differential equation that describes particle displacements [Virieux and Operto 2009]. This equation can be simplified by assuming an

isotropic medium, constant density, and by neglecting shear strains. This simplified version can be written as follows:

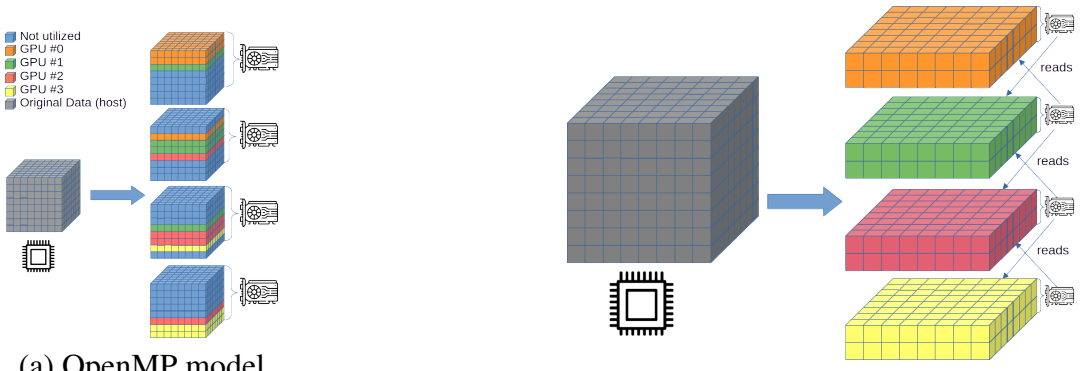
$$\frac{\partial^2 p}{\partial t^2}(x,t) - c^2(x)\nabla^2 p(x,t) = -\rho c^2(x)\nabla.b(x,t) \tag{1}$$

where ρ corresponds to the density, b are external body forces, p represents scalar pressure, $c = \sqrt{\frac{k}{\rho}}$ is the wave speed and $\nabla = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z})$ in cartesian coordinates. Equation 1 can be discretized using second-order in time and variable spatial order (even, varying from 2,...,20). Finally, the equation reduces to:

$$\begin{aligned} p^{n+1}_{i,j,k} &= 2p^n_{i,j,k} - (1-\eta\Delta t)p^{n-1}_{i,j,k} \\ &+ \frac{c^2\Delta t^2}{1+\eta\Delta t}\frac{1}{\Delta x^2}(v_0 p^n_{i,j,k} + \sum_{m=1}^{r} v_i(p^n_{i+m,j,k} + p^n_{i-m,j,k})) \\ &+ \frac{c^2\Delta t^2}{1+\eta\Delta t}\frac{1}{\Delta y^2}(v_0 p^n_{i,j,k} + \sum_{m=1}^{r} v_i(p^n_{i,j+m,k} + p^n_{i,j-m,k})) \\ &+ \frac{c^2\Delta t^2}{1+\eta\Delta t}\frac{1}{\Delta z^2}(v_0 p^n_{i,j,k} + \sum_{m=1}^{r} v_i(p^n_{i,j,k+m} + p^n_{i,j,k-m})). \end{aligned} \tag{2}$$

This equation can be implemented as a stencil computation in software, which is very computationally intensive and a strong candidate for execution in manycore environments, as the high number of relatively simple calculations suits devices such as GPUs.

3. Methods



(a) OpenMP model

(b) UVA model

Figure 1. Memory allocation for the two strategies.

Our experiment is built upon a wave propagation simulator called Miniwave, a subset of a larger project named Simwave [Freire de Souza et al. 2022], which implements the forward equation in self-contained external kernel functions. These functions are implemented in C using many different techniques and are incorporated by a python front-end. One such technique is an OpenMP *target* single-gpu implementation, which is used as a basis for our tests.

Our goal was to study the use of CUDA's native memory functions in an OpenMP target kernel. The chosen one for our testing was Unified Virtual Addressing (UVA), a

technology in which memory pointers can be indiscriminately accessed either by the host or by any device present in the system. With UVA, data location is handled by the runtime driver, and prefetching and preferred locations may be hinted at via CUDA functions by the developer to improve performance [Nvidia 2023].

To benchmark the strategies, we first implemented domain decomposition using OpenMP's *data map* directives, which creates entire copies of the arrays onto every device, following the API specification [OpenMP 2021]. Synchronization was necessary in for exchanging halo data, which was implemented using the *data update* pragmas between iterations. One host thread was created for each device, being responsible for defining the correct parameters for each device. Our UVA implementation was made using *cudaMallocManaged* function calls following the Runtime API guidelines [Nvidia 2023]. This function allocates memory and creates virtual pointers accessible from any device. Data present on the host was copied onto these newly-allocated arrays and passed on to the OpenMP kernel by the *is_device_ptr()* clause. Implementing this strategy was simple, as accessing data from neighboring devices is handled by the runtime driver, being transparent to the developers. Much like in the OpenMP implementation, a host thread is responsible for defining starting and ending array positions for each device.

As exemplified in figure 1, in the OpenMP *data map* model, each GPU must have enough memory available for a copy of the entire grid, requiring more overall available memory than is actually needed, while in the UVA model, each GPU only stores its own local sub-domain, but can access halo data from its neighbors during execution.

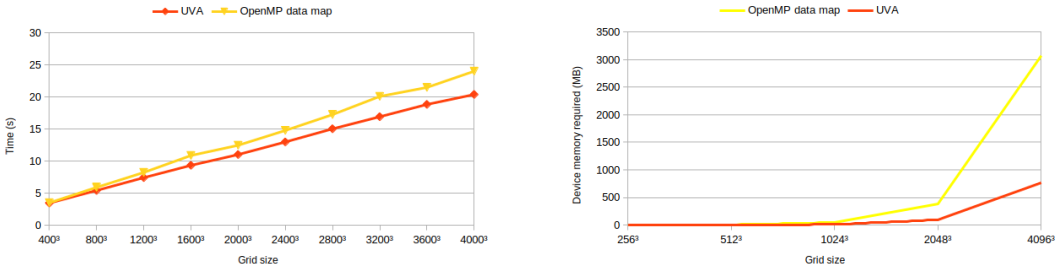
4. Results

Space Order	Size	UVA	OpenMP
SO = 2	Size = 256	4,66	4,31
	Size = 512	15,01	17,45
	Size = 1024	92,99	96,54
SO = 4	Size = 256	5,58	6,31
	Size = 512	20,73	24,65
	Size = 1024	144,67	135,6
SO = 8	Size = 256	8,31	10,89
	Size = 512	31,93	37,07
	Size = 1024	261,45	212,68

Table 1. Execution time for different grid sizes and space orders

The tests were conducted on a computing node equipped with 4 NVidia v100 GPUs connected via NVLink. As shown in Table 1, both implementations produce similar execution times and are proportionately affected by increasing grid sizes, while the OpenMP-only version is more negatively affected by bigger stencil radii. This is due to the slow synchronous transfer among devices, which halts execution after each iteration while devices wait for data to be updated.

Figure 2 shows how the UVA implementation performs consistently better than its *data map* counterpart, with lower execution times. Performance differences increase with bigger number of iterations, while the amount of GPU memory required per device increase much faster on the OpenMP-only version, which may limit grid sizes on systems where the GPUs are equipped with less memory.



(a) Execution time (b) Device memory required

Figure 2. Execution time for varying grid sizes with stencil radius = 2 (a) and amount of device memory required (b) in both strategies

Implementing both versions require attention in different areas. The *data map* method demands care when synchronizing data between iterations, as reading the wrong values due to improper synchronization can produce wrong results; while the UVA version requires a great deal of optimization with data prefetching and memory location advising to perform well, but is much less error-prone.

5. Conclusion

Integrating Nvidia's UVA memory pointers into an OpenMP *target* GPU kernel offers advantages when it comes to performance and ease of implementation. Utilizing native device memory functions can offer many tools that allow for easy implementation of domain decomposition. Although execution times were similar between both methods shown, future work should be conducted exploring other methods to further improve on performance, as many different native GPU data management tools are available and may be able to produce better results for stencil calculations in multi-gpu environments.

Acknowledgements

Y.N.F. thanks Programa Institucional de Bolsas de Iniciação Científica - PIBIC/CNPq/UFSCar for the scholarship. The authors thank FAPESP (Process # 2019/26702-8).

References

Freire de Souza, J., Moreira, J. B. D., Roberts, K. J., di Ramos Alves Gaioso, R., Gomi, E. S., Silva, E. C. N., and Senger, H. (2022). `simwave` – a finite difference simulator for acoustic waves propagation. *arXiv.org*.

Nvidia (2023). Cuda runtime api :: Cuda toolkit documentation. https://docs.nvidia.com/cuda/cuda-runtime-api/index.html. Last accessed on 22nd April, 2023.

OpenMP (2021). Openmp application programming interface specification version 5.2. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf. Last accessed on 3rd April, 2023.

Virieux, J. and Operto, S. (2009). An overview of full-waveform inversion in exploration geophysics. *Geophysics*, 74(6):WCC1–WCC26.