Code portability applied the Wave-propagation kernel

Rodrigo C. Machado[1] **, Cristiano A. Künas**[1] **, Bruno Machado Morales**[1]
Thiago da Silva Araújo [1] **, Alexandre Sardinha** [2] **, Arthur F. Lorenzon** [1]
Philippe O. A. Navaux [1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

{rcm,cakunas,bmmorales,thiago.araujo,aflorenzon,navaux}@inf.ufrgs.br

[2]Petróleo Brasileiro S. A., Brazil

alexandre.sardinha@petrobras.com.br

Abstract. *Reverse Time Migration (RTM) codes are often written using GPU vendor APIs, tying applications to one hardware supplier. The Fletcher method models wave propagation used by RTM. We port a CUDA implementation of the Fletcher method to HIP, Kokkos, RAJA, and SYCL, measuring eight kernel variants (1-pass or 2-pass, float vs. packed float2, with or without shared memory) across five GPUs from AMD and NVIDIA. No single kernel variant fits all architectures best. Packed float2 achieves the highest performance gains on AMD, whereas shared memory on NVIDIA often degrades throughput. On AMD (MI300A), HIP peaks at 22.15 GS/s with Kokkos within 4%, while on NVIDIA (GH200), CUDA/HIP reach 20.35 GS/s.*

1. Introduction

Scientific computation involves the usage of complex physics and math. It requires heavy processing and often handling large files. To meet these requirements, scientific software must be tuned to the target hardware and must handle computation, communication, and I/O efficiently to achieve the best performance. This leads to big codebases, often using vendor-specific APIs which tie their development to their products.

Changing vendors means rewriting parts of production code or maintaining multiple versions for vendor-specific APIs. This problem keeps companies from using the product that best fits their needs at a given time, as the human and time resources needed to port or maintain multiple vendor-specific codebases are more expensive than the savings of switching technologies.

Companies in the Oil and Gas field use different software for each field related to their works. One of these is the Reverse Time Migration (RTM), which produces imaging of the subsurface of the earth or the water, assisting in the surveys of new oil and gas reservoirs. RTM software is usually written to work with GPUs, and each vendor has its own API and compiler, optimized and often exclusive to their own products. The Fletcher method [Fletcher et al. 2009] is an algorithm to simulate wave propagation in a medium.

Our contributions are showing the behavior of different kernel techniques on different architectures from AMD and NVIDIA and how well the code portability frameworks perform. We test the trade-offs of a combination of using shared memory or not,

using float against float2, and using a single kernel (1-pass) to run the wave propagation or two kernels to compute it (2-pass). The memory hierarchy, bandwidth and computation throughput vary between the architectures we use in our experiments, showing that there is not a single best universal optimization. We analyze how each version, optimization, and framework behaves and compare each solution using the Pennycook metric [Pennycook and Sewall 2021] to measure performance portability.

Our main findings show that no single kernel variant or programming model is best for all architectures. On NVIDIA GPUs, every HIP kernel variant runs within 0.3% of its native CUDA counterpart, since HIP maps directly to CUDA API calls. Comparing the best variant of each framework, HIP is 4.0% faster than Kokkos on the MI300A, while RAJA and SYCL are 31% and 38% slower than Kokkos. On the RX 7900 XT, all four frameworks are within 6% of each other. 1-pass execution is faster on GPUs with lower memory bandwidth per SM, such as the L40S, whereas higher-bandwidth GPUs favor 2-pass execution. Packed float2 yields the largest gains on AMD by reducing register spilling, while shared memory helps on AMD but gives mixed results on NVIDIA, slowing down most H100 and GH200 variants.

Section 2 reviews related work. Section 3 describes the Fletcher method and its 1-pass and 2-pass kernels, with and without shared memory and float2. Section 4 discusses the port to each framework. Section 5 presents the experimental setup, parameters, and metrics. Section 6 presents and discusses the results, and Section 7 concludes the paper.

2. Related work

Evaluating performance portability has gained attention as different HPC centers deploy GPUs from different vendors. The closest work to ours is that of [Lorenzon et al. 2024], who evaluated eleven implementations of the Fletcher model across CUDA, HIP, Kokkos, RAJA, and OpenMP Target on five GPU generations from AMD and NVIDIA. They showed that HIP achieves the highest performance portability and that the impact of the programming model grows as the L2 cache capacity decreases. A follow-up study [Lorenzon et al. 2025] extended the same set of implementations to energy consumption and carbon emissions. In both studies the implementations differ in programming model and in grid decomposition (2D or 3D), while the algorithm stays fixed: the time step is always computed in two passes, one kernel for the partial derivatives and one for the propagation. As a consequence, they show which framework ports best, but not whether an algorithmic choice that pays off on one GPU still pays off on another.

Outside RTM, Malenza et al. [Malenza et al. 2024] ported the LSQR solver of the Gaia AVU-GSR code to CUDA, C++ PSTL, SYCL, OpenMP, and HIP, and measured it on five GPUs from AMD and NVIDIA. HIP was the most portable model, closely followed by SYCL compiled with AdaptiveCpp, while the tuning-oblivious PSTL version lagged behind. Shared-memory staging [Micikevicius 2009] and kernel fusion [Wahib and Maruyama 2014] are established optimizations for GPU stencils and memory-bound kernels, and are the algorithmic choices we evaluate here.

To quantify performance portability we use the metric of Pennycook et al. [Pennycook and Sewall 2021, Pennycook et al. 2019], the harmonic mean Φ of the efficiencies of an application over a set of platforms, which is zero if the application does not run on one of them. Marowka [Marowka 2025] showed that when the efficiency

is taken relative to the best implementation observed on each platform, the score of an implementation can change only because another implementation was added to the comparison, and proposes alternatives that avoid this effect. Harrell et al. [Harrell et al. 2018] complement the performance view with metrics of development productivity, collected while porting production codes to portability frameworks. In prior Fletcher studies each implementation is one application, so Φ ranks framework and decomposition together.

Our work crosses the two dimensions that the studies above treat separately. On the algorithmic side, we vary the time-step structure (one pass or two passes), the data layout (`float` or packed `float2`), and shared-memory staging. On the framework side, we port each of the eight variants to HIP, Kokkos, RAJA, and SYCL, adding SYCL to the models evaluated for Fletcher. We compute Φ both per framework–variant pair and per framework, using the best variant of the framework on each GPU.

3. Fletcher Method

The Fletcher application models a seismic wave propagation over time in a medium. It is the core of applications such as RTM, which store snapshots of the forward propagation and reuse them in the backward propagation to image the subsurface. We study only the computational part, so snapshotting is disabled and only the kernels are timed. The application is a single-GPU C++ code that uses CUDA on NVIDIA GPUs.

The Fletcher method formulation [Fletcher et al. 2009] uses a tilted transversely isotropic (TTI) medium and models a pseudo-acoustic wave propagation. This is a stencil application of order 8, with radius R of 4, where each point to be updated needs the values of its four neighbors in each direction over the three axes. The application receives the grid dimensions nx, ny, nz, the absorbing border width $absorb$, the thread block dimensions $block_x, block_y, block_z$, and the kernel variant. The allocated grid adds the absorbing border and a pad of R on each side, so $sx = nx + 2(R + absorb)$, and likewise for sy and sz. We keep sx a multiple of 32 in every experiment so that a warp reads a contiguous, aligned segment.

The application allocates 3D arrays of these dimensions, ten of which describe the medium and are only read: six anisotropy coefficients and four squared velocities. The wave field is held by two pairs, pc and qc for the current state and pp and qp for the previous one. Four more arrays, pDx, pDy, qDx and qDy, hold the partial computations of the derivatives and are used only by the 2-pass version.

Algorithm 1 gives the structure shared by every version. Each time step injects the source at a fixed coordinate, propagates the wave, and swaps the current and previous arrays. The versions differ only in how line 9 is split into kernels and in how the intermediate values are stored.

3.1. 1-pass

The 1-pass version computes the whole update in a single kernel. Each thread handles one (X, Y) column and sweeps it along Z. At each Z position, it reads the stencil neighbors of pc and qc and updates pp and qp. The three cross derivatives (XY, XZ, YZ) are computed on the fly, reading neighbors on different levels of two axes at once, which makes the accesses non-contiguous and expensive.

Algorithm 1 Fletcher method: outer structure

function FLETCHER($nx, ny, nz, st, dt, bord, absorb, blk$)
  $sx, sy, sz \leftarrow padGrid(nx, ny, nz, bord, absorb)$
  $C \leftarrow initMedium(sx, sy, sz)$ ▷ $ch1dxx, ch1dyy, ch1dzz, ch1dxy, ch1dyz, ch1dxz$
  $V \leftarrow initVelocities(sx, sy, sz)$ ▷ $v2px, v2pz, v2sz, v2pn$
  $pp, pc, qp, qc \leftarrow allocZeroed(sx, sy, sz)$
  $sendToGPU(C, V, pp, pc, qp, qc)$
  for $it \leftarrow 1$ to st **do**
    $injectSource(pc, qc, it \cdot dt)$
    $wavePropagate(pp, pc, qp, qc, C, V, blk)$
    $swap(pc, pp)$; $swap(qc, qp)$
  end for
  $copyToMainMemory(pc, qc)$
end function

3.2. 2-pass

The 2-pass version splits the computation into two kernels. The first kernel computes the X and Y derivatives once and stores them in pDx, pDy, qDx and qDy. The second kernel uses them to compute the rest of the update. In the 1-pass version, some of these derivatives are recomputed for neighboring points.

This is the *Der1Der1* modification of [Serpa et al. 2021]. They measured the cross derivatives at 76% of the floating-point operations, of which 82% was repetition. The price is four full-sized arrays written and read through DRAM at every time step.

3.3. Shared memory

Both versions have a shared-memory variant. For every Z plane, each block loads a tile of $(block_x + 2R) \times (block_y + 2R)$ elements into shared memory, so the X and Y neighbors of the plane are read once from global memory and then reused. Each thread keeps the nine Z neighbors of its column in a local array, which is shifted as the loop advances one plane, so only one new value is loaded per plane and the remaining ones are reused. In the 1-pass version the tile holds pc and qc. In the 2-pass version the first kernel stages pc and qc, and the second one stages pc, qc, pDx and qDx.

3.4. Float2

Every access to a p array at a point comes with an access to a q array at the same point. The float2 variant therefore merges the pairs pc/qc and pp/qp, and, in the 2-pass version, also pDx/qDx and pDy/qDy. A single memory transaction then delivers both values.

4. Code portability frameworks

We ported our CUDA version to four code portability frameworks: HIP, Kokkos, RAJA, and SYCL. The physics is written once, in headers shared by the five backends, so each port only rewrites memory allocation, kernel launch, thread indexing, and the shared-memory tile. Kokkos, RAJA, and SYCL express each kernel as a lambda over an index space, while HIP keeps the CUDA kernel syntax.

Porting to HIP was basically mapping the CUDA API to HIP calls. The call prefix changes from `cuda` to `hip` and the arguments are the same.

On the Kokkos framework we replaced raw pointers with `Kokkos::View` in a device memory space, and the launch by a `parallel_for` over a `TeamPolicy`, where the team is the block and the team members are its threads. Memory traits on the Views take the role of the CUDA qualifiers: `const` with `RandomAccess` for read-only data and `Restrict` for outputs. The shared memory tile becomes an `Unmanaged View` in the scratch memory space, sized per team. Since every View must be released before Kokkos is finalized, we declare a `Kokkos::ScopeGuard` as the first member of the driver class, so that C++ destroys the Views before the guard finalizes Kokkos.

RAJA does not handle memory, so here we keep the raw pointers, but the allocations go through Umpire, a library suggested in the RAJA documentation that handles memory allocation on the host and the device. The launch becomes a `RAJA::launch` with one loop policy per block and thread dimension, and the shared tile becomes a `RAJA_TEAM_SHARED` array. These policies name the vendor (`cuda_thread_x_direct` or `hip_thread_x_direct`), so the backend selects them with the preprocessor.

SYCL allocates with unified shared memory, `sycl::malloc_device` and `sycl::malloc_host`, and launches by submitting a `parallel_for` over an `nd_range` to a queue, with the block given as the local range. Unlike CUDA, the global range counts work items instead of blocks and must be a multiple of the local range. Both ranges list their dimensions from the slowest to the fastest varying, so the contiguous dimension `x`, which comes first in CUDA, comes last in SYCL. The shared tile becomes a `sycl::local_accessor`. This backend contains no vendor-specific code.

Kokkos and SYCL fail silently when the device rejects a launch, and the run proceeds as if the kernel had executed. To detect these failures, the Kokkos backend queries the error state of the underlying CUDA or HIP runtime after each propagation step, and the SYCL backend attaches an exception handler to the queue that aborts the run.

5. Methodology

5.1. Experimental setup

Our experiments were conducted on the Santos Dumont supercomputer and on the GPPD facilities, on different nodes each comprising different CPU-GPU and memory architectures. Experiments were conducted on a single node and a single GPU. Table 1 shows the hardware configuration of each node used and Table 2 shows the software stack used in each node.

5.2. Parameters

All results use, on each machine, the largest cubic grid that fits in device memory: 888^3 points on the MI300A, 600^3 on the RX 7900 XT, 760^3 on the L40S, 984^3 on the H100, and 1048^3 on the GH200. For each backend and kernel type — that is, for each combination of (1-pass, 2-pass) $\times$ (float, float2) $\times$ (with, without shared memory) — we searched for the best thread-block size for the large grid on the device. This resulted in up to eight different thread-block configurations for each backend on each device. We used the following configurations in the search:

Table 1. Target hardware platform specifications

	grace1	h100	gh200	sirius	mi300a
GPU	L40S	H100 80GB HBM3	GH200 120GB	RX 7900 XT	MI300A
Architecture	sm_89	sm_90	sm_90	gfx1100	gfx942
SM / CU	142	132	132	42	228
Memory	44.4 GiB	79.2 GiB	95.0 GiB	20.0 GiB	128.0 GiB
Shared / LDS	100 KiB[c]	228 KiB[c]	228 KiB[c]	64 KiB	64 KiB
L1 / vL1D	*unified*	*unified*	*unified*	32 KiB[b]	32 KiB
L2	96.00 MiB	50.00 MiB	60.00 MiB	6.00 MiB	4.00 MiB[a]
LLC	—	—	—	80 MiB	256 MiB
Bandwidth (GB/s)	864	3352	4023	—	—

Table 2. Version, compiler, and optimization flags per backend

Backend	Version	Compiler	Flags
CUDA	CUDA 12.9	nvcc	-O3 –fmad=true -Xptxas -O3 –maxrregcount=127
HIP (AMD)	ROCm 7.0.2	hipcc (Clang 20)	-O3 -ffp-contract=fast
HIP (NVIDIA)	ROCm 7.0.2 / CUDA 12.9	hipcc ($\rightarrow$ nvcc)	-O3 –fmad=true -Xptxas -O3 – maxrregcount=127
RAJA	v2025.12.2 (Umpire v2025.12.0)	nvcc / hipcc	Same as CUDA / HIP, plus –extended-lambda
Kokkos	4.7.04	nvcc_wrapper / hipcc	-O3
SYCL	AdaptiveCpp 3536de0	acpp (LLVM 19/20)	-O3 (cuda:sm_XX / hip:gfxXXXX)

- NVIDIA: 32×4, 32×8, 32×16, 32×32, 64×4, 64×8, 16×8, 16×16
- AMD: 64×4, 64×8, 64×16, 32×8, 32×16, 32×32, 16×16

The search spaces differ to account for architectural SIMD widths: AMD CDNA architectures (such as the MI300A) utilize a 64-lane wavefront (Wave64), favoring block widths of 64 to maximize vector memory coalescing and SIMD occupancy, whereas NVIDIA GPUs and AMD RDNA architectures operate with 32-lane warps/wavefronts. Every reported result uses the optimal block configuration for its specific (machine, backend, kernel) triple.

To verify the numerical correctness of every port, we ran the native version (CUDA on NVIDIA, HIP on AMD) and used its output as reference. All frameworks and versions were executed with the same parameters for 50 time steps, and at each saved step every point of the wavefield was compared with the reference, dividing the absolute difference by the largest magnitude of the reference field. All differences were below 10^{-5} for every framework and version.

5.3. Metrics

Throughput is reported in GigaSamples per second, the number of grid points updated per unit of time:

$$\mathrm{GSamples/s} = \frac{10^{-9} \cdot N \cdot st}{t_{\mathrm{propagate}}}, \tag{1}$$

where $N = (sx - 2R)(sy - 2R)(sz - 2R)$ is the number of points updated at each time step, st is the number of time steps, and $t_{\mathrm{propagate}}$ is the accumulated time of the propagation kernels alone. Source injection and initialization are timed separately and are not included.

We used the Pennycook metric [Pennycook and Sewall 2021] to assess each framework on its performance portability. For an application a solving problem p on a set of platforms H, it is the harmonic mean of the efficiencies:

$$\Phi(a, p, H) = \begin{cases} \dfrac{|H|}{\sum_{i \in H} \dfrac{1}{e_i(a, p)}} & \text{if } a \text{ runs on every } i \in H \\ 0 & \text{otherwise,} \end{cases} \tag{2}$$

where $e_i(a, p)$ is the application efficiency of the framework on platform i: its median throughput divided by the best median obtained on that platform by any framework. This means that if on a set of architectures a framework cannot be used on one of them, it has 0 points.

We run 30 executions of each configuration, where one execution is one complete run of the program, and report the *median*, with the interquartile range as the dispersion. To state that one version is better than another we follow the three cases of Jain: if the confidence intervals of the two means do not overlap, the difference is significant; if they overlap and one mean falls inside the interval of the other, it is not; and only in the remaining case do we compute the confidence interval of the difference of the means. Intervals use Student's t, and we report the highest of the 0.99, 0.95 and 0.90 levels at which the verdict holds.

6. Results and discussion

Every figure and metric below is computed from 30 complete executions at the large grid size, using the best thread-block configuration for each (machine, framework, kernel) triple. Table 3 summarizes the peak median throughput (in GSamples/s) achieved by each framework on the five GPUs.

Figures 1, 2, and 3 illustrate the main effect ratio of each design axis (data layout, shared-memory staging, and kernel fusion). In all three charts, a bar extending to the right indicates that the right-hand variant achieves higher performance, while a bar extending to the left favors the left-hand variant, with bar length reflecting the speedup ratio. A missing bar signifies that Jain's three-case confidence interval test at the 90% level was statistically inconclusive (neither variant was significantly superior). An arrowhead at the tip of a bar indicates that the speedup ratio exceeds the plot axis boundaries.

Table 3. Peak median throughput (GSamples/s) per framework across target GPU platforms

Framework	MI300A	RX 7900 XT	L40S	H100	GH200
CUDA	—	—	9.21	17.30	20.34
HIP	22.15	4.50	9.22	17.26	20.35
Kokkos	21.30	4.55	8.43	13.22	13.93
RAJA	14.73	4.31	8.88	16.83	19.43
SYCL	13.11	4.55	8.41	13.17	14.83

Each value represents the highest median throughput (in GSamples/s) achieved by any kernel variant and block configuration of that framework on the large grid size. Dash (—) indicates unavailable target.

6.1. HIP and CUDA on NVIDIA

Across all 24 evaluated cells on NVIDIA GPUs, the HIP version runs at 0.9974 to 1.0012 of the CUDA performance. This happens because `hipcc` targeting an NVIDIA device maps HIP calls to CUDA calls, close to a reverse hipify. The code is then compiled with `nvcc`, as described in the HIP documentation. Both versions therefore share nearly the same source code and the same compiler. Divergences may occur when a CUDA API call has no HIP counterpart or when HIP uses a generic library instead of one tuned for NVIDIA. Our kernels do not fall into these cases. They only use common CUDA API calls that have a direct HIP translation.

6.2. AMD: Hand-tuning vs. Framework abstractions

The HIP version was not developed for AMD. It is a port of the CUDA version, so it runs the same kernels on AMD GPUs. The only platform-specific tuning is the thread-block size. This explains why HIP does not always have the best performance on AMD.

6.3. Data layout axis: Float vs Float2

Figure 1 shows the impact using `float2` has on each kernel version. Packing the wavefields as `float2` gives the best results in general, with speedups between $3.98\times$ and $4.33\times$ on the RX 7900 XT in the 1-pass kernel with shared memory and $1.57\times$ to $2.99\times$ on the MI300A.

On NVIDIA GPUs, the same layout change gives at most $1.61\times$, and it slows the 1-pass CUDA and HIP kernels by about 10% on the H100 and GH200.

On the RX 7900 XT, the 1-pass kernel reserves 940 bytes of scratch per thread under Kokkos and 900 under SYCL with plain `float`, and none with `float2`. On the MI300A, `float2` reduces it from 924 to 288 bytes per thread for HIP in the 1-pass kernel with shared memory, and from 1988 to 524 for Kokkos in the 1-pass kernel without it. Packing the two fields also reduces the vector instruction count on the MI300A by 6% to 42%. On NVIDIA, `float2` also removes most of the local-memory spill of the 1-pass kernel, except under SYCL, but its median gain is only $1.01\times$ to $1.14\times$, against $1.44\times$ to $1.83\times$ on AMD. Our hypothesis for this behavior is the cache level that serves the spill. On AMD, the scratch traffic goes through small L2 caches (4 MiB per XCD on the MI300A, 6 MiB on the RX 7900 XT) before reaching the Infinity Cache, whereas on NVIDIA local memory is backed by 50–96 MB L2 caches, which likely keep the spill of a resident block on chip.

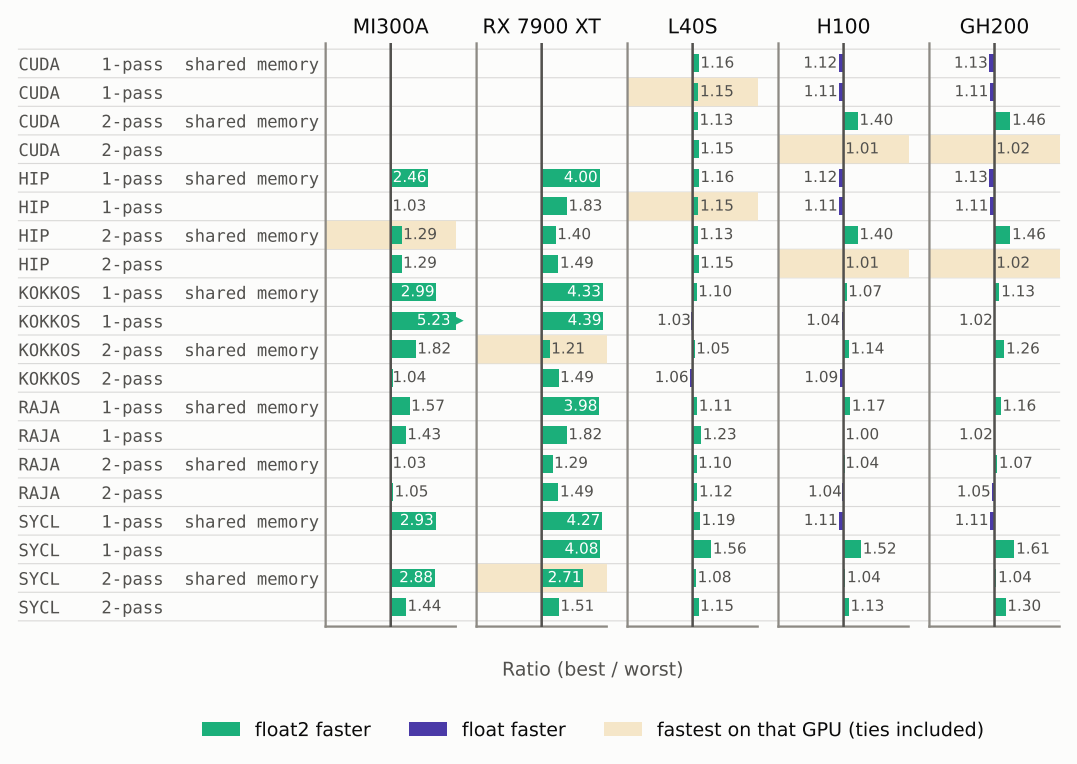

Figure 1. Performance effect ratio of data layout packing (plain `float` vs. packed `float2`). Bars extending to the right favor `float2`; missing bars denote inconclusive verdicts; arrowheads indicate ratios exceeding axis bounds.

6.4. Shared memory axis

In the 2-pass kernel, the shared memory axis (Figure 2) has opposite winners between vendors. On the MI300A, shared memory gives a $2.34\times$ speedup for HIP and $2.07\times$ for RAJA.

On the NVIDIA Hopper GPUs, performance degrades, reaching $0.64\times$ on the H100 and $0.58\times$ on the GH200 for CUDA, and down to $0.46\times$ for RAJA on the GH200.

Hardware counters explain this penalty. On the H100, shared memory drops the L1 hit rate of the 2-pass CUDA kernel from 74.4% to 7.2% and raises the DRAM read volume from 337.8 to 416.6 GB, while the instruction count grows by $1.23\times$. NVIDIA's L1 cache was already reusing the data, and the shared memory carved out of the same on-chip storage takes capacity from it.

On AMD, the L1 cache is separate from shared memory. Using shared memory therefore does not take capacity from the L1 of the CU, and it also reduces the instruction count ($0.85\times$ for HIP on the MI300A). Thus, shared memory helps where the hardware cache does not already capture the reuse, and hurts otherwise.

6.5. Kernel fusion: 1-pass vs 2-pass

The kernel fusion axis (Figure 3) shows that 2-pass execution wins in nearly all cases: $1.73\times$ on the H100, $2.00\times$ on the GH200, $1.69\times$ on the RX 7900 XT, and $4.86\times$ on the MI300A. The L40S is the single exception, where 1-pass wins by $1.61\times$.

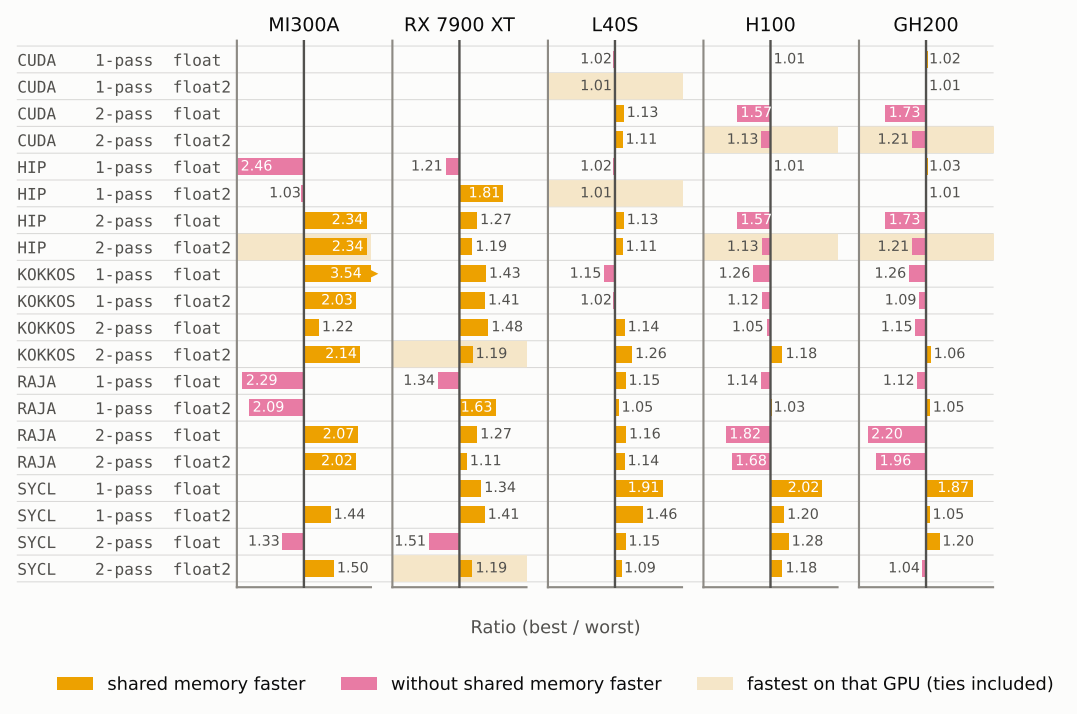

Figure 2. Performance effect ratio of shared-memory staging (2D vs. 2.5D shared-memory window). Bars extending to the right favor 2.5D shared memory; missing bars denote inconclusive verdicts.

On the L40S, fusing kernels cuts DRAM traffic from 162.9 to 95.4 bytes per grid point, but multiplies the executed instructions by approximately $3\times$ (from 20.7 to 61.4 billion, measured with the Nsight Compute counter `smsp__inst_executed`), because the 1-pass kernel recomputes derivatives shared by neighboring points.

Whether this trade-off pays off depends on the memory bandwidth available per Streaming Multiprocessor (SM). The L40S has 864 GB/s of DRAM bandwidth across 142 SMs (6.1 GB/s per SM), whereas the H100 provides 3352 GB/s across 132 SMs (25.4 GB/s per SM) and the GH200 provides 4023 GB/s across 132 SMs (30.5 GB/s per SM), four to five times more bandwidth per SM. On the H100 and GH200, memory bandwidth is abundant, so the extra instructions dominate. This also explains why the L40S breaks the NVIDIA pattern on shared memory: shared memory cuts the DRAM reads of the CUDA 2-pass kernel from 188.8 to 168.2 GB at a cost of only $1.04\times$ in instructions, for a gain of $1.11\times$ to $1.13\times$. Both L40S anomalies, 1-pass winning and shared memory winning, have the same cause: the L40S has the lowest bandwidth per SM of the three NVIDIA GPUs, so trading instructions for less DRAM traffic pays off.

6.6. Framework performance portability analysis

Table 4 reports Φ over the AMD GPUs (MI300A, RX 7900 XT), the NVIDIA GPUs (L40S, H100, GH200), and all five. When each GPU uses the best variant of the framework, HIP has the highest Φ, followed by RAJA. The metric puts CUDA at zero because it does not run on AMD, while HIP, a direct byproduct of the CUDA code, runs on both vendors. The final results hide failures on some architectures, such as Kokkos, which has $\Phi = 0.847$ but is weakest on the GH200 (0.684 efficiency) and strongest on

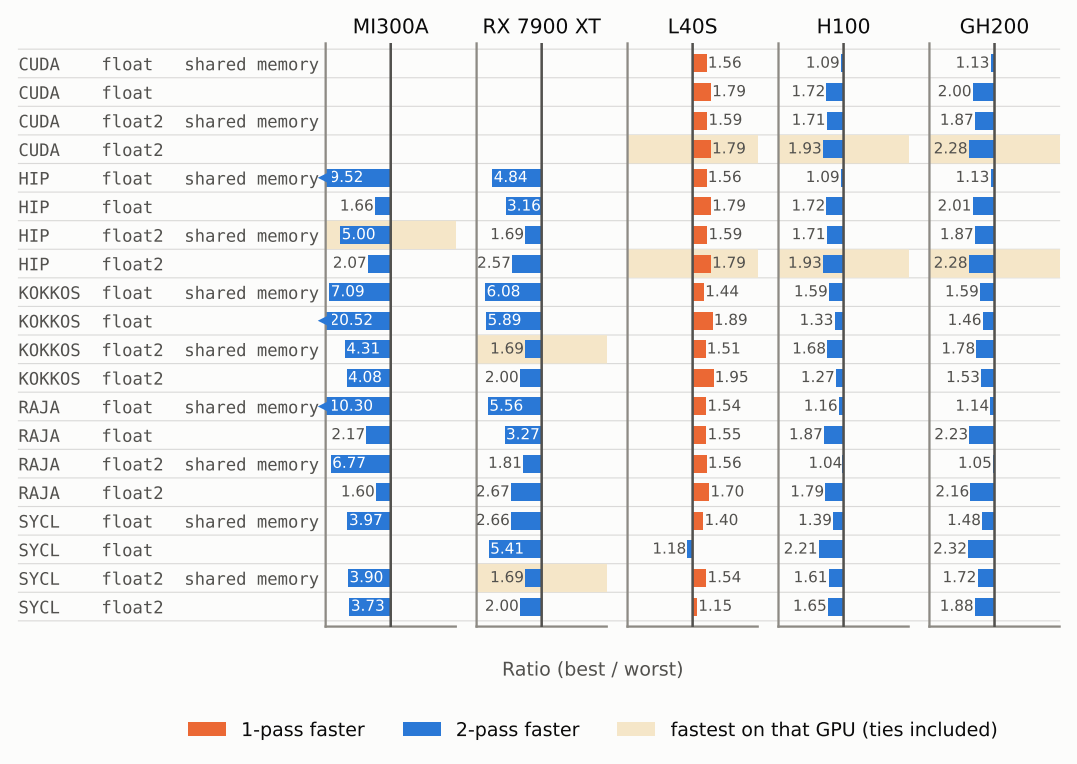

Figure 3. Performance effect ratio of time-step kernel fusion (1-pass vs. 2-pass). Bars extending to the right favor 2-pass execution; missing bars denote inconclusive verdicts; arrowheads indicate ratios exceeding axis bounds.

the RX 7900 XT (1.000). SYCL is weakest on the MI300A (0.592 efficiency) and also reaches 1.000 on the RX 7900 XT, for an overall $\Phi = 0.773$.

These scores take the best variant of each framework on each GPU, that is, they assume the developer selects the features that suit each architecture. When the variant is fixed and the same code runs on every GPU, as in prior Fletcher studies [Lorenzon et al. 2024], the scores drop and the ranking changes (last two columns of Table 4). The best fixed variant is the same for the four frameworks, 2-pass with float2 and shared memory, and with it HIP reaches $\Phi = 0.839$, followed by Kokkos (0.764), SYCL (0.702), and RAJA (0.614). The gap between the two settings is the cost of not adapting the algorithm to each architecture, and it is largest for RAJA.

Rather than representing uniform performance penalties, Kokkos and SYCL underperform on different devices likely due to specific compiler register allocation issues. At their optimal configurations, both portable frameworks match native code on the RX 7900 XT. The cost of portability is not a flat tax, but is concentrated in specific cells where generated code loses control of register allocation.

7. Conclusion and Future Works

We evaluated the Fletcher method on five GPUs from four architectures (NVIDIA Ada and Hopper, AMD CDNA3 and RDNA3), with five programming models (CUDA, HIP, Kokkos, RAJA, SYCL) and eight kernel variants. Our results show that no single algorithmic optimization is best on every GPU. Kernel fusion pays off only on the L40S, the GPU

Table 4. Performance portability scores (Φ) per framework

	Best per GPU			Best overall	
Framework	Φ_{AMD}	Φ_{NVIDIA}	Φ_{ALL}	**Variant**	Φ_{ALL}
CUDA	0.000	0.999	0.000	—	0.000
HIP	0.995	0.999	0.997	2P-F2-S	0.839
Kokkos	0.980	0.777	0.847	2P-F2-S	0.764
RAJA	0.782	0.963	0.882	2P-F2-S	0.614
SYCL	0.744	0.793	0.773	2P-F2-S	0.702

Φ is Pennycook's performance portability metric. Best per GPU: each GPU uses the best variant of the framework. Best overall: the same variant on every GPU, the one with the highest Φ_{ALL} (2P-F2-S: 2-pass, float2, and shared memory).

with the lowest bandwidth per SM, by trading DRAM traffic for instructions, whereas the other GPUs favor 2-pass execution. In the 2-pass kernel, shared-memory staging gives up to $2.34\times$ on AMD, where LDS and vector L1 are separate, but slows the NVIDIA Hopper GPUs (down to $0.58\times$ on the GH200), where shared memory takes capacity from the L1 cache. Packed `float2` gives the largest gains on AMD (up to $5.23\times$) by reducing register spilling, while on NVIDIA its gains are at most $1.61\times$. On performance portability, HIP reaches $\Phi = 0.997$ over all five GPUs, while the abstractions (RAJA $\Phi = 0.882$, Kokkos $\Phi = 0.847$, SYCL $\Phi = 0.773$) lose performance in specific cells.

Among the frameworks, HIP required the lowest porting effort and offers the best balance between portability and performance. For applications similar to this one, such as TTI wave propagation, the 2-pass kernel is the best choice on every GPU except the L40S, which has the lowest bandwidth per SM. Packed `float2` is part of the best fixed variant of every framework, since coupling values that are read together reduces the number of memory accesses. The downside of HIP is that it targets only GPUs, whereas Kokkos, RAJA, and SYCL also offer CPU backends. As future work, we intend to evaluate 3D thread-block mappings and decompositions, multi-GPU setups with halo exchange, and the frameworks on CPUs and in executions that combine CPU and GPU computation.

8. Acknowledgement

Some experiments in this work used the PCAD infrastructure (`http://pcad.inf.ufrgs.br`) at INF/UFRGS. The machine *grace* was funded by FAPERGS (22/2551-0000390-7) and *sirius* by Petrobras (TC 5900.0111175.19.9, 2020/00182-5).

The authors acknowledge the National Laboratory for Scientific Computing (LNCC/MCTI, Brazil) for providing HPC resources of the SDumont Supercomputer (`http://sdumont.lncc.br`), which have contributed to the research results reported within this paper.

This work was also supported by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001. Additionally, it was partially funded by the project CNPq/MCTI/FNDCT - Universal 18/2021 under grant 406182/2021-3.

References

Fletcher, R. P., Du, X., and Fowler, P. J. (2009). Reverse time migration in tilted transversely isotropic (TTI) media. *Geophysics*, 74(6):WCA179–WCA187.

Harrell, S. L., Kitson, J., Bird, R., Pennycook, S. J., Sewall, J., Jacobsen, D., Asanza, D. N., Hsu, A., Carrillo, H. C., Kim, H., and Robey, R. (2018). Effective performance portability. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 24–36.

Lorenzon, A. F., Navaux, P. O. A., Sardinha, A., and Messer, B. (2024). Towards performance portability of an oil and gas application on heterogeneous architectures. In *2024 IEEE 36th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 127–137.

Lorenzon, A. F., Navaux, P. O. A., Sardinha, A., and Messer, B. (2025). Evaluating code portability for carbon-efficient RTM computing. In *2025 IEEE/SBC 37th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 91–101.

Malenza, G., Cesare, V., Santimaria, M. E., Birke, R., Vecchiato, A., Becciani, U., and Aldinucci, M. (2024). Performance portability via C++ PSTL, SYCL, OpenMP, and HIP: the Gaia AVU-GSR case study. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1152–1163.

Marowka, A. (2025). Portability efficiency approach for calculating performance portability. *Future Generation Computer Systems*, 170:107826.

Micikevicius, P. (2009). 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, page 79–84, New York, NY, USA. Association for Computing Machinery.

Pennycook, S., Sewall, J., and Lee, V. (2019). Implications of a metric for performance portability. *Future Generation Computer Systems*, 92:947–958.

Pennycook, S. J. and Sewall, J. D. (2021). Revisiting a metric for performance portability. In *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 1–9.

Serpa, M. S., Pavan, P. J., Cruz, E. H. M., Machado, R. L., Panetta, J., Azambuja, A., Carissimi, A. S., and Navaux, P. O. A. (2021). Energy efficiency and portability of oil and gas simulations on multicore and graphics processing unit architectures. *Concurrency and Computation: Practice and Experience*, 33(18):e6212.

Wahib, M. and Maruyama, N. (2014). Scalable kernel fusion for memory-bound GPU applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, page 191–202. IEEE Press.