Hardware-Agnostic Compact Difference Schemes in C++/Kokkos for High-Order CFD

Sana Nazir* and Jonathan Poggie†
Purdue University School of Aeronautics & Astronautics , West Lafayette, IN, 47901

A hardware-agnostic implementation of compact finite difference schemes is presented using the Kokkos programming model, targeting both CPUs (OpenMP) and GPUs (CUDA, HIP) without code duplication. The parallel cyclic reduction (PCR) algorithm achieves significant speedup over the sequential Thomas algorithm on GPUs for large three dimensional grids. For time dependent problems, naive plane by plane execution incurs substantial kernel launch and memory initialization overhead. This is mitigated through a fully batched execution that solves transverse tridiagonal systems simultaneously, yielding up to two orders of magnitude speedup in GPU execution time. The results highlight the necessity of algorithm level redesign when porting legacy Fortran solvers to accelerator architectures. Cross platform numerical solutions agree within machine precision across all CPU and GPU backends, indicating deterministic solver behavior. Comprehensive validation through method of manufactured solutions confirms design order spatial convergence (rates approaching 6.0 for sixth order schemes) and fourth order temporal accuracy with RK4 integration across heat equation, Burgers equation, and isentropic vortex test cases. Long time stability analysis with the isentropic Shu vortex reveals 4th order schemes survive 2 to 5 additional convection periods compared to 6th order schemes before aliasing induced failure. This confirms known filtering requirements for extended simulations. The resulting framework provides verified, performance-portable building blocks for CFD applications requiring high-order accuracy and sustained numerical integrity across evolving compiler and hardware architectures. All development and debugging were performed locally using the OpenMP backend, with total GPU usage limited to fewer than 100 node-hours.

I. Nomenclature

a_i, b_i, c_i	=	Tridiagonal matrix coefficients	d_i	=	Right-hand side of tridiagonal system
α	=	Thermal diffusivity	ν	=	Kinematic viscosity
Δt	=	Time step size	$\Delta x, \Delta y, \Delta z$	=	Grid spacings
d	=	Number of spatial dimensions	u	=	Scalar solution variable
$\nabla^2 u$	=	Laplacian of u	u_{num}	=	Numerical solution
u_{exact}	=	Exact (manufactured) solution	L_∞	=	Maximum absolute error norm
L_2	=	Discrete L_2 error norm	$L_{2,\text{rel}}$	=	Relative L_2 error norm
E	=	Discrete total energy	N_x, N_y, N_z	=	Grid points in each direction
$C4, C6$	=	Fourth- and sixth-order compact schemes	$AC4, AC6$	=	Asymmetric compact schemes
CR	=	Cyclic Reduction solver	PCR	=	Parallel Cyclic Reduction solver
$Thomas$	=	Sequential tridiagonal solver	$RK4$	=	Fourth-order Runge–Kutta scheme
CPU	=	Central Processing Unit	GPU	=	Graphics Processing Unit
$CUDA$	=	NVIDIA GPU programming model	HIP	=	AMD GPU programming model
$OpenMP$	=	Shared-memory CPU parallelism	$Kokkos$	=	Performance-portable C++ framework
x, y, z	=	Physical coordinates	ξ, η, ζ	=	Computational coordinates
$OMP-64$	=	OpenMP with 64 threads			

*PhD Student, School of Aeronautics & Astronautics, AIAA Student Member
†Professor, School of Aeronautics and Astronautics, AIAA Associate Fellow

II. Introduction

Compact finite difference schemes achieve high spectral-like accuracy using narrower stencils compared to explicit schemes of equivalent order [1]. This property makes them particularly attractive for turbulence resolving computational fluid dynamics (CFD) on structured grids, where capturing fine scale structures with minimal numerical dissipation is critical [2]. The pioneering work by Lele [1] established the theoretical foundations of compact schemes, demonstrating their spectral-like resolution characteristics. Subsequently, Gaitonde and Visbal [3] extended these schemes for practical CFD applications, including specialized boundary treatments. It has been demonstrated that compact finite difference methods offer superior spatial accuracy on coarser grids compared to conventional schemes [4], potentially leading to significant reductions in computational cost while maintaining high-order accuracy.

Achieving performance portability in computational fluid dynamics is a nontrivial task that requires algorithmic redesign and rigorous cross platform numerical verification. Traditional development workflows require separate code paths for multiple programming models and hardware targets, such as CUDA for NVIDIA GPUs, HIP for AMD GPUs, OpenMP for CPUs, etc. The Kokkos programming model [5] provides a unified abstraction layer that decouples numerical algorithms from hardware specific implementation details. As a result, the majority of development and debugging can be performed locally on CPU systems using the OpenMP backend, where correctness, unit testing, and accuracy verification can be established. Once numerical integrity is ensured, and a test framework is established, performance tuning and optimization for GPU architectures can be conducted using Kokkos profiling tools [6], enabling efficient performance portability without duplicating solver implementations or requiring extensive GPU compute time.

This work presents a Kokkos based implementation of high-order compact finite difference schemes (4th, 5th, and 6th order) demonstrating that true GPU acceleration necessitates parallel algorithms rather than syntactic translation of sequential methods. Two tridiagonal solution strategies are implemented: the sequential Thomas algorithm for CPU efficiency and Parallel Cyclic Reduction (PCR) for GPU parallelism. The entire development and debugging cycle was carried out locally on a laptop using the OpenMP backend, with only sporadic verification on HIP and CUDA backends, totaling fewer than 100 GPU node hours.

The Kokkos programming model provides a mature, production-ready foundation for implementing performance portable compact finite difference schemes. As demonstrated in recent studies, Kokkos achieves superior performance portability compared to other programming models, with performance portability scores of 68% versus OpenMP's 28% across diverse architectures [7]. This proven track record makes it an ideal choice for CFD applications requiring both numerical precision and hardware portability.

Beyond performance metrics, this work emphasizes numerical portability as essential infrastructure for high-order CFD. Compiler optimizations targeting machine learning workloads increasingly threaten IEEE-compliant behavior required by physics applications. We advocate for open-source, continuously tested reference implementations with automated regression detection across evolving compiler and hardware landscapes. As GPU computing matures with diverging precision requirements between AI and CFD domains, the computational physics community requires sustained verification infrastructure [8]. This work developed performance portable building blocks and a sustainable verification framework based on Google Test and a curated suite of PDE benchmarks with manufactured and analytical solutions for maintaining numerical integrity across heterogeneous architectures.

Performance portability is measured within a single unified codebase and does not claim to measure against hand optimized implementations across programming models. All timing comparisons use identical Kokkos enhanced source code compiled against different Kokkos execution backends (Serial: single-threaded CPU execution, OpenMP: multi-threaded CPU execution, CUDA: NVIDIA GPU execution, HIP: AMD execution).

III. Mathematical Formulation

Compact finite difference schemes achieve high-order accuracy through implicit relationships between derivative values at adjacent grid points. The general form is:

$$\alpha f'_{i-1} + f'_i + \alpha f'_{i+1} = a\frac{f_{i+1} - f_{i-1}}{2\Delta x} + b\frac{f_{i+2} - f_{i-2}}{4\Delta x} \tag{1}$$

where f'_i represents the derivative at point i, and coefficients α, a, b determine the scheme's order and accuracy. This implicit coupling creates tridiagonal systems requiring specialized parallel solvers for efficient CPU and GPU implementation.

Our implementation includes explicit schemes (E1-E6), compact schemes (C2-C6), and specialized boundary treatments (DE1-DE6 for explicit boundaries, AC4-AC6 for compact boundaries).

The code tests multiple combinations of schemes, denoted using the format "Interior-SecondPoint-FirstPoint-LastSecondPoint-LastPoint". For example, "C4-AC4-C4-AC4-C4" indicates a fourth-order compact scheme in the interior with alternative compact boundary schemes, while "E4-DE4-E4-DE4-E4" uses fourth-order explicit schemes throughout.

For curvilinear grids, coordinate transformations map physical space (x, y, z) to computational space (ξ, η, ζ). Following Thomas and Lombard [9], the chain rule relates physical derivatives to computational derivatives:

$$\frac{\partial \phi}{\partial x} = \frac{\partial \phi}{\partial \xi}\frac{\partial \xi}{\partial x} + \frac{\partial \phi}{\partial \eta}\frac{\partial \eta}{\partial x} + \frac{\partial \phi}{\partial \zeta}\frac{\partial \zeta}{\partial x} \tag{2}$$

with analogous expressions for y and z derivatives.

The geometric conservation law (GCL) governs the time variation of the coordinate transformation Jacobian J:

$$\frac{\partial J}{\partial t} + \frac{\partial}{\partial \xi}(J\xi_t) + \frac{\partial}{\partial \eta}(J\eta_t) + \frac{\partial}{\partial \zeta}(J\zeta_t) = 0 \tag{3}$$

where $J = \left|\frac{\partial(x,y,z)}{\partial(\xi,\eta,\zeta)}\right|$ is the transformation Jacobian. Satisfying the GCL numerically ensures that grid motion does not introduce spurious numerical errors, particularly critical for high-order compact schemes.

Our implementation provides two metric calculation approaches:

1) **Standard approach:** Direct computation using coordinate derivatives
2) **Conservative approach:** GCL-based formulation ensuring geometric conservation

The conservative approach, based on Thomas and Lombard's differential GCL formulation [9], maintains consistency between the finite-difference discretization and the effective volume elements, preventing grid motion-induced errors that can corrupt high-order accuracy [3].

Further details on scheme constants and stability preserving filtering methods can be found in [1, 3, 4].

IV. Numerical Implementation

A. Fortran Code Base

The initial implementation of compact finite difference schemes used for this work was developed in Fortran and has been extensively validated in previous work [4, 10]. The reference Fortran codebase supports a variety of stencil orders, with modular components for setting scheme coefficients, applying compact differencing operators, and computing grid metrics in curvilinear coordinates.

At its core, the Fortran solver applies high order compact schemes to structured three dimensional grids by computing spatial derivatives along each computational coordinate direction. Derivative routines are configured using coefficient generation modules tailored to the chosen order of accuracy. Derivatives in each direction are computed via dimensionally split routines that operate on 2D data slices extracted from the full 3D field.

The solver supports both standard and conservative forms of metric term evaluation to accommodate curvilinear geometries. These metric terms are derived from coordinate transformation functions and are used to convert between computational and physical space gradients.

Tridiagonal systems arising from the compact discretization are solved using the classical Thomas algorithm in a sequential, in-place fashion. While efficient for CPU execution on a single thread, this serial algorithm forms the primary bottleneck when targeting parallel architectures such as GPUs or multi-core systems. This limitation motivated the current effort to develop a portable, high-performance C++ implementation using Kokkos.

In the reference Fortran implementation, the innermost loops of the derivative and tridiagonal solve routines are explicitly structured to promote SIMD vectorization on CPUs[11]. This is achieved through contiguous memory access patterns along the fastest varying index and compiler level vectorization directives, enabling efficient utilization of wide vector registers. While this approach is excellent on cache based CPU architectures, it is fundamentally incompatible with GPU execution. On GPUs, performance is governed by massive thread level parallelism and memory coalescing across thousands of concurrent threads, rather than long vector operations within a single thread. The sequential data dependencies inherent to the Thomas algorithm and its vectorized inner loops prevent effective mapping onto SIMT execution models, where warp level synchronization [12] and occupancy dominate performance. Thus, to target GPUs and achieve performance, a complete algorithmic redesign is required.

B. Tridiagonal Solvers

The tridiagonal system arising from compact schemes can be represented as:

$$Ax = d \tag{4}$$

where A is a tridiagonal matrix:

$$A = \begin{pmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & c_3 & \\ & & \ddots & \ddots & \ddots \\ & & & a_n & b_n \end{pmatrix} \tag{5}$$

Each equation has the form:

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i \tag{6}$$

where a_i, b_i, and c_i are the coefficients of the tridiagonal matrix, and d_i is the right-hand side.

For our implementation, we employ three different solvers: the Thomas algorithm for sequential execution on CPU, and Cyclic Reduction (CR) [13] and Parallel Cyclic Reduction (PCR) for parallel execution on GPU. CR and PCR have been shown to be effective for GPU implementation of tridiagonal solvers [14]. In this work the behavior of the PCR algorithm is investigated in depth, the CR algorithm analysis and performance is deferred to future work.

The Thomas algorithm is the standard sequential method for solving tridiagonal systems, representing a specialized form of Gaussian elimination optimized for tridiagonal matrices. It consists of two phases: forward elimination and backward substitution.

Forward elimination:

$$c'_1 = \frac{c_1}{b_1}, \quad c'_i = \frac{c_i}{b_i - c'_{i-1} a_i}, \quad i = 2, 3, \ldots, n-1 \tag{7}$$

$$d'_1 = \frac{d_1}{b_1}, \quad d'_i = \frac{d_i - d'_{i-1} a_i}{b_i - c'_{i-1} a_i}, \quad i = 2, 3, \ldots, n \tag{8}$$

Backward substitution:

$$x_n = d'_n, \tag{9}$$

$$x_i = d'_i - c'_i x_{i+1} \quad i = n-1, n-2, \ldots 1 \tag{10}$$

The Thomas algorithm is inherently sequential because each step depends on the results of the previous calculation. While computationally efficient with $O(n)$ operations, it requires $2n$ strictly sequential steps, making it unsuitable for parallel architectures.

The CR and PCR algorithms address this limitation by trading additional computational work for reduced algorithmic depth and increased parallelism [13, 14].

The CR algorithm operates by successively reducing the system size through elimination of odd-indexed unknowns. The algorithm consists of two phases: forward reduction and backward substitution.

In each reduction step, even-indexed equations are updated as linear combinations of three consecutive equations. For equation i, the update formulas are:

$$k_1 = \frac{a_i}{b_{i-1}}, \quad k_2 = \frac{c_i}{b_{i+1}} \tag{11}$$

$$a'_i = -a_{i-1} k_1, \quad b'_i = b_i - c_{i-1} k_1 - a_{i+1} k_2 \tag{12}$$

$$c'_i = -c_{i+1} k_2, \quad d'_i = d_i - d_{i-1} k_1 - d_{i+1} k_2 \tag{13}$$

This process continues until a 2×2 system remains, which is solved directly. The algorithm then works backward, solving for the previously eliminated odd-indexed unknowns:

$$x_i = \frac{d'_i - a'_i x_{i-1} - c'_i x_{i+1}}{b'_i} \tag{14}$$

Although the CR algorithm provides greater parallelism than the Thomas algorithm, its parallel efficiency decreases as the reduction progresses. In particular, the final reduction stages and the initial back-substitution steps expose insufficient parallelism to fully utilize GPU cores [14].

The PCR algorithm further improves GPU utilization by eliminating the backward substitution phase entirely. Instead of reducing to a single smaller system, PCR simultaneously reduces the original system to multiple independent subsystems. In each reduction step, all equations are processed in parallel. With stride $s = 2^{\text{step}}$, each equation i is updated using the linear combination of equations $i - s$, i, and $i + s$. The reduction formulas are identical to those of the CR algorithm, but the application pattern differs significantly.

The key advantage of the PCR algorithm is that it maintains full parallelism throughout the entire solution process.After $\log_2 n$ reduction steps, the algorithm produces multiple independent 1×1 or 2×2 systems that can be solved simultaneously. For 1×1 systems the solution is directly available as

$$x_i = \frac{d_i}{b_i}. \tag{15}$$

This approach provides consistent GPU utilization throughout the algorithm execution.

Algorithm	Operations	Steps	Parallelism	Memory Pattern
Thomas	$O(n)$	$2n$	None	Sequential
CR	$O(n)$	$2\log_2 n - 1$	Variable	Bank conflicts
PCR	$O(n \log_2 n)$	$\log_2 n$	Full	Conflict-free

Table 2 Comparison of tridiagonal solvers. More details on these statistics are available in [14]

Transition from CPU to GPU implementation necessitates abandoning the Thomas algorithm due to its inherently sequential nature. While Thomas provides optimal computational complexity, its lack of parallelism renders it inefficient on massively parallel architectures. At the same time PCR incurs added cost while running on single threads. Thus, it is imperative to have both implementations available in any truly hardware agnostic code base.

The CR algorithm offers a compromise between computational efficiency and parallelism. It maintains linear computational complexity but provides limited parallelism, particularly problematic in later algorithm stages. Additionally, its memory access pattern can lead to bank conflicts (when multiple GPU threads access different memory addresses that map to the same shared-memory bank, forcing serialized access) in GPU shared memory. The PCR algorithm's regular memory access pattern avoids the bank conflicts[14]. PCR truly shines when problem sizes get very large, this is demonstrated by our operator performance results. A truly scalable system should be able to adapt and switch between solvers as needed. The choice between CR and PCR should depend on the specific problem size, available GPU memory bandwidth, and the trade-off between computational work and parallel efficiency for the target hardware architecture.

V. Parallelization Strategy and Implementation Takeaways

Kokkos abstracts parallel execution and memory management through six core concepts: execution spaces define where computation occurs, execution patterns express algorithmic parallelism, and execution policies control scheduling. memory spaces, memory layouts, and memory traits manage data placement and access patterns. This separation allows numerical schemes to maintain their mathematical properties while adapting to different hardware architectures automatically [5] [7].

Compact schemes benefit significantly from proper data layout. In this work, we explicitly use `Kokkos::LayoutRight` for 2D and 3D Views to guarantee stride-1 access in the innermost index, which is optimal for CPU cache utilization and vectorization. On GPU backends, this study prioritizes numerical correctness and cross-platform consistency over detailed optimization of memory coalescing. Although Kokkos Views do enable automatic layout selection based on the target architecture, this feature has not yet been investigated for performance or correctness in the present work.

The hardware-agnostic approach was validated across diverse computing platforms, including NVIDIA GPUs (A30, L40S, H100) and MI50 AMD architectures. The same source code achieved reasonable performance and near identical accuracy on each platform after recompiling the code and Kokkos for the appropriate backend selection, demonstrating excellent portability.

It was observed that code accessing class members inside device lambdas compiled and executed under older CUDA toolchains, but failed under HIP. This indicates that earlier CUDA compilers silently tolerated a pattern that is not robust in a heterogeneous setting, namely device code implicitly de-referencing host-resident objects. The stricter behavior in HIP exposed this portability bug, which was resolved by explicitly creating local aliases of all required class members inside each kernel-launching function (e.g., `auto var = _var`) prior to invoking device lambdas, this avoids any implicit `this` capture. This approach was preferred over explicit function parameter forwarding to minimize call overhead and enable aggressive compiler inlining and register allocation inside device kernels. Related undefined host–device behavior has been previously reported by Mejstrik [15].

Host–device lambdas may not be defined inside private or protected class member functions. This restriction breaks the use of Kokkos kernels inside GoogleTest `TEST_F` bodies, since GoogleTest generates a private `TestBody()` method in which the device lambda is implicitly defined. All affected tests were updated to comply with CUDA 12.6.0 rules. Interestingly, no corresponding fixes were required on the HIP backend. This behavior contrasts with an earlier issue in which HIP flagged a memory-access pattern that CUDA permitted. The combined observations highlight subtle but important differences in backend compiler enforcement of C++ device rules [16].

Recent large-scale portability studies show that GPU programming models can exhibit differing backend and compiler dependent behavior for identical code [16], thus expecting exact enforcement of device code rules can backfire when writing portable code and remains the responsibility of the programmer.

The largest performance gain was obtained not from additional solver-level micro-optimizations, but from a restructuring of the overall execution model to reduce kernel launch overhead. The original load–solve–store implementation processed each tri-diagonal system with an independent kernel launch, yielding millions of very small kernels even for moderately sized three-dimensional grids. In this regime, kernel launch latency dominated the runtime, and the underlying cost of the tri-diagonal algorithms was effectively masked by orchestration overhead. To address this, the implementation was reformulated as a fully batched, plane-based strategy in which all tri-diagonal systems associated with a given two-dimensional slice are solved within a single kernel launch. This approach amortizes launch overhead across thousands of independent systems, increases arithmetic intensity, and exposes sufficient concurrency to better utilize the GPU. Once batching was introduced, both the Thomas and PCR solvers exhibited substantial speedups, and their relative performance could be assessed in a regime where kernel launch costs are no longer the dominant factor.

Another optimization introduced was reuse of cached temporary workspaces across calls to reduce repeated allocation overhead.

Persistent workspace caching and batch launching vectorized grids, significantly impacts the peak resident memory footprint of the application. As a consequence of this intentional design shift, the memory limits were hit on available hardware for the largest problem sizes which were previously computable. These results highlight a fundamental memory–performance trade-off in large-scale high-order compact GPU implementations. An important direction for future work is a more nuanced memory aware and flexible caching strategy.

An additional practical advantage of the Kokkos-based implementation is the dramatic reduction in required large-scale GPU resources during development. Because the same code path executes on CPUs, local GPUs, and leadership-class GPU systems, nearly all debugging, verification, and unit testing were performed locally. Only the final large-scale performance benchmarks required access to H100 GPUs. As a result, the entire project consumed under 500 H100 GPU-hours in total, which is exceptionally low for a three-dimensional, high-order compact finite-difference solver with implicit tridiagonal line solves. This demonstrates that performance portability not only enables cross-architecture execution, but also substantially reduces the real computational cost of GPU-based research.

VI. Base Operator Verification

To validate the correctness of the compact and explicit derivative operators, we performed a manufactured–solution accuracy study across all supported stencil families, including explicit (E1–E6), diagonal–enhanced (DE1–DE6), approximate compact (AC4–AC6), and fully compact (C4, C6) schemes. For each stencil, we computed the derivative of the smooth analytic field:

$$f(x, y, z) = \sin(\pi x) \sin(\pi y) \sin(\pi z) \tag{16}$$

and recorded the global error as well as directional errors e_x, e_y, e_z. As the grid is refined, all schemes exhibit the expected convergence behavior: low–order explicit stencils reduce the error at first or second order rates, while higher order compact and AC/DE hybrid stencils achieve errors in the 10^{-8}–10^{-14} range at $N = 256$. Directional errors remain nearly identical, demonstrating that the assembled three–dimensional operator is isotropic and that the one dimensional d–plane solves are consistent across all spatial directions. These tests confirm that the compact coefficients are generated

correctly for all stencil families, the solver backend does not affect accuracy, and the base derivative operator attains its formal order on uniform Cartesian grids.

Cross-platform numerical consistency was verified by comparing results from the reference Fortran implementation (Thomas algorithm) against Kokkos/C++ implementations using OpenMP (CPU), CUDA (NVIDIA H100), and HIP (AMD MI50) backends with the PCR algorithm and the Thomas algorithm. Table 3 shows maximum infinity norm errors at $N = 256$ for representative schemes.

Platform to platform differences remain at machine precision ($< 10^{-14}$), with most schemes achieving bit-level consistency (deviations of 10^{-16} to 10^{-17}) at the finest grids. These results confirm that both the PCR and Thomas algorithm implementations preserve numerical accuracy to machine precision across heterogeneous architectures, validating the algorithmic portability approach for production CFD applications.

Table 3 Cross-platform numerical consistency for selected compact finite difference schemes at $N = 256$.

Scheme	Fortran	OpenMP	CUDA	HIP	Max. Dev.
E4-AC5-C6-AC5-E4	4.064E-10	4.064E-10	4.064E-10	4.064E-10	3.5E-17
C4-AC4-C4-AC4-C4	2.991E-10	2.991E-10	2.991E-10	2.991E-10	3.8E-17
E5-DE5-C6-DE5-E5	6.183E-12	6.183E-12	6.182E-12	6.182E-12	3.2E-16
C6-AC6-C6-AC6-C6	7.541E-14	7.540E-14	6.720E-14	6.720E-14	8.2E-15
Platform differences $\leq 10^{-14}$					

VII. PDE Verification Tests

A. Laplacian Stiffness Matrix Test

We used a smooth separable trigonometric function:

$$u(x, y, z) = \sin(a\pi x)\sin(b\pi y)\sin(c\pi z) \tag{17}$$

where a, b and c are non-integer frequencies $(0.21, 0.23, 0.27)$ to avoid symmetry artifacts. This function has an analytical solution,

$$\nabla^2 u = -\pi^2(a^2 + b^2 + c^2)\, u \tag{18}$$

which provides a clean comparison benchmark. Max error (Maximum absolute pointwise error across the domain),

$$L_\infty = \max_{i,j,k} \left|\nabla^2 u_{\text{num}}(i, j, k) - \nabla^2 u_{\text{exact}}(i, j, k)\right| \tag{19}$$

L_2error (root mean square error over all grid points),

$$L_2 = \sqrt{\frac{1}{N_x N_y N_z} \sum_{i,j,k} \left(\nabla^2 u_{\text{num}}(i, j, k) - \nabla^2 u_{\text{exact}}(i, j, k)\right)^2} \tag{20}$$

and Relative L_2 error (L_2 error normalized by the L_2 norm of the exact solution),

$$L_{2,\text{rel}} = \frac{\left\|\nabla^2 u_{\text{num}} - \nabla^2 u_{\text{exact}}\right\|_2}{\left\|\nabla^2 u_{\text{exact}}\right\|_2} = \sqrt{\frac{\sum_{i,j,k} \left(\nabla^2 u_{\text{num}} - \nabla^2 u_{\text{exact}}\right)^2}{\sum_{i,j,k} \left(\nabla^2 u_{\text{exact}}\right)^2}} \tag{21}$$

were computed.

This test isolates discretization error from time integration and provides a direct assessment of high-order accuracy, solver consistency (Thomas, PCR), and platform portability (CUDA, HIP, OpenMP). The resulting stiffness matrix is inverted along each grid line using the selected solver, and the discrete Laplacian is compared to the analytical reference.

For low-order or auxiliary stencils where the tridiagonal system is small enough, PCR recursion is unnecessary. In these cases, the implementation automatically falls back to serial Thomas, which is more stable and efficient for such regimes.

A progressive order study demonstrates systematic error reduction with increasing order, measured slopes closely match theoretical predictions.

Solver correctness was verified by comparing the Parallel Cyclic Reduction (PCR) solver against the reference serial Thomas algorithm. Both methods were seen to exhibit identical error profiles across all grid resolutions, demonstrating that PCR recursion introduces no additional numerical diffusion or instability. The observed slope is consistent with the theoretical accuracy of each scheme order.

All architectures reproduce the expected convergence behavior for 4th, 5th, and 6th-order compact Laplacians. The E4 and E6 families achieve the expected $O(h^4)$ and $O(h^6)$ scaling.

The pure 6th-order C6 interior suffers from an error floor at high resolution ($N \geq 128$), this is expected in high-order compact formulations. This accuracy drop is consistent across all backends, confirming the degradation is inherent to the stencil. It was observed that a hybrid scheme using a C6 interior with E5 boundaries restores full $O(h^6)$ convergence. See Fig. 1 and Fig. 2.

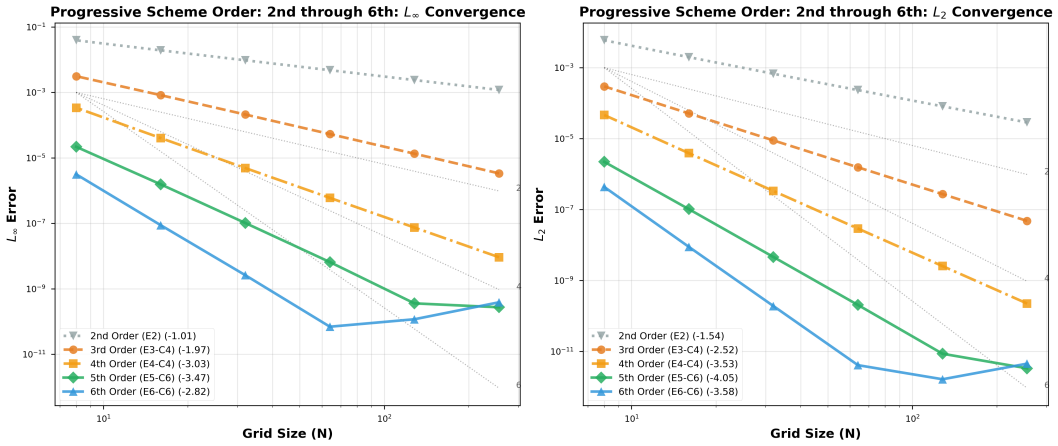


Fig. 1 Convergence behavior of progressively higher-order schemes from 2nd through 6th order.

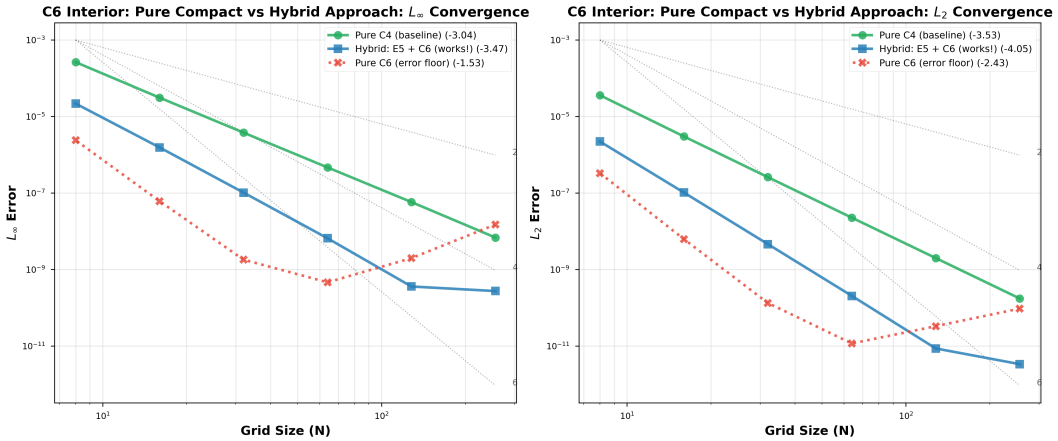


Fig. 2 Comparison of pure C6 interior versus hybrid E5+C6 compact scheme for the 3D Laplacian.

B. Heat Equation 3D

The heat equation is defined as:

$$u_t = \alpha \nabla^2 u \tag{22}$$

1. Manufactured Solution - Convergence

The smooth manufactured solution

$$u_{\text{exact}}(x, y, z, t) = \sin(a\pi x)\sin(b\pi y)\sin(c\pi z)\exp(-\lambda t)$$

where $\lambda = \alpha\pi^2(a^2 + b^2 + c^2)$ and $a = b = c = 1$, was used. The problem is solved using the C6–AC6–C6–AC6–C6 sixth-order compact scheme in space and RK4 for time integration. Gridsize N is varied as $8, 16, 32, 64$. Error is measured along the 3D L_2 norm Eq.(20).

The solver achieves sixth-order spatial accuracy, as expected from the compact stencil and fourth order temporal as expected from RK4 scheme. CUDA, HIP, and OpenMP backends produce numerically indistinguishable results with differences within machine precision. As seen in the strictly overlapping curves shown in Fig. 3.

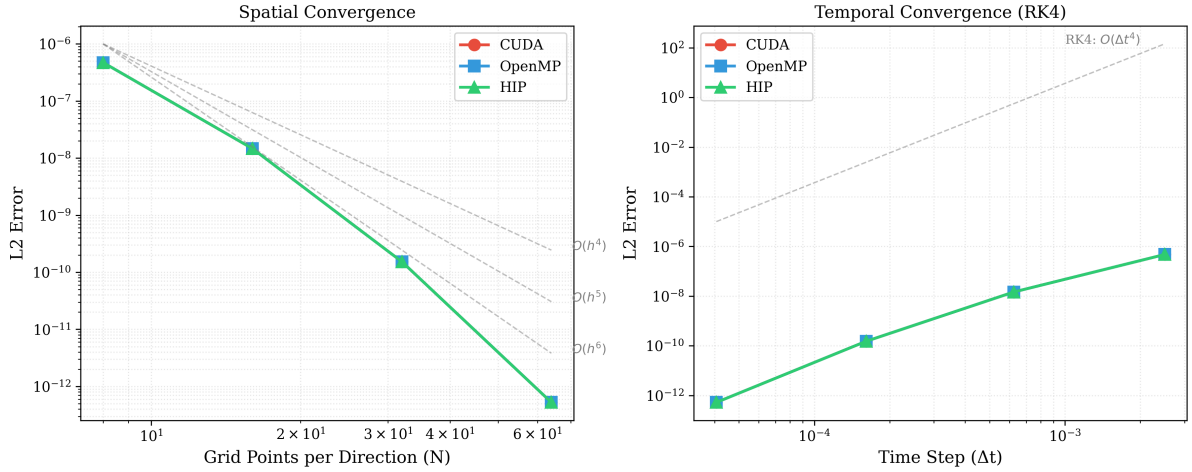


Fig. 3 Convergence behavior of the sixth-order compact diffusion operator with RK4 time integration. (a) Spatial convergence of the discrete Laplacian for the analytical heat-equation mode $u(x, y, z) = \sin(\pi x)\sin(\pi y)\sin(\pi z)$ showing sixth-order accuracy. (b) Temporal convergence of the RK4 integrator demonstrating the expected $O(\Delta t^4)$ behavior.

2. 3D Heat Diffusion from a Gaussian Pulse

We next simulate 3D heat diffusion using:

$$\frac{\partial u}{\partial t} = \alpha\left(\frac{\partial u}{\partial x^2} + \frac{\partial u}{\partial y^2} + \frac{\partial u}{\partial z^2}\right) \tag{23}$$

with Dirichlet boundary conditions $u = 0$ on all faces. and a radially symmetric Gaussian initial condition centered at $(0.5, 0.5, 0.5)$

$$u(x, y, z, 0) = \exp\left(-\frac{(x-0.5)^2 + (y-0.5)^2 + (z-0.5)^2}{2\sigma^2}\right) \tag{24}$$

Spatial discretization: C6–AC6–C6–AC6–C6, time integration RK4 and timestep set as $\Delta t = 0.1 \times \frac{1}{2d}\frac{\Delta x^2}{\alpha}$, where $d = 3$, in accordance with the standard stability limit for multidimensional diffusion. The Gaussian pulse spreads smoothly and symmetrically, and both energy and temperature decay monotonically, consistent with the expected diffusive scaling. CUDA, HIP, and OpenMP versions produce overlapping energy decay curves, identical centerline profiles, and matching 2D/3D temperature fields. See Fig. 4, Fig. 5 and Fig. 6.

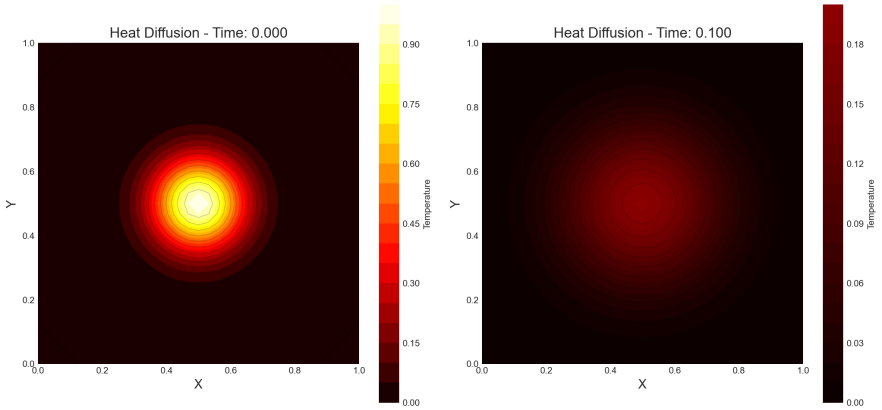


Fig. 4 Evolution of a 3D Gaussian temperature pulse under the heat equation using the sixth-order compact Laplacian and RK4 time integration. The solution exhibits smooth radial diffusion and monotonic decay, consistent with analytical expectation.

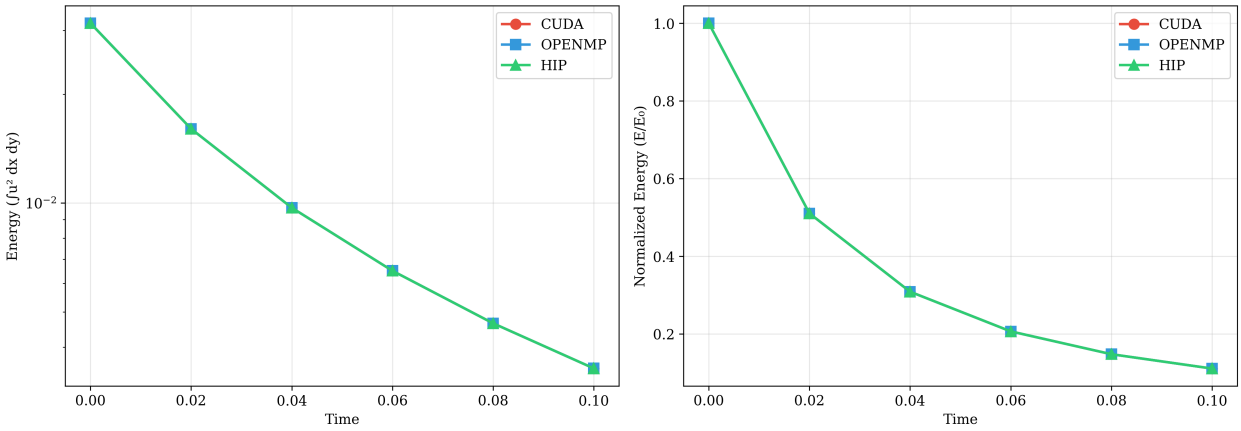


Fig. 5 Energy conservation analysis for the Gaussian heat-pulse diffusion test. (Left) Total energy $E(t) = \frac{1}{2}\int u^2 \, \mathrm{d}V$ decreases monotonically as expected for the heat equation. (Right) Normalized energy $E(t)/E(0)$ showing identical decay curves for CUDA, OpenMP, and HIP backends. The overlap of all curves to machine precision demonstrates backend-invariant behavior of the compact Laplacian and RK4 integrator.

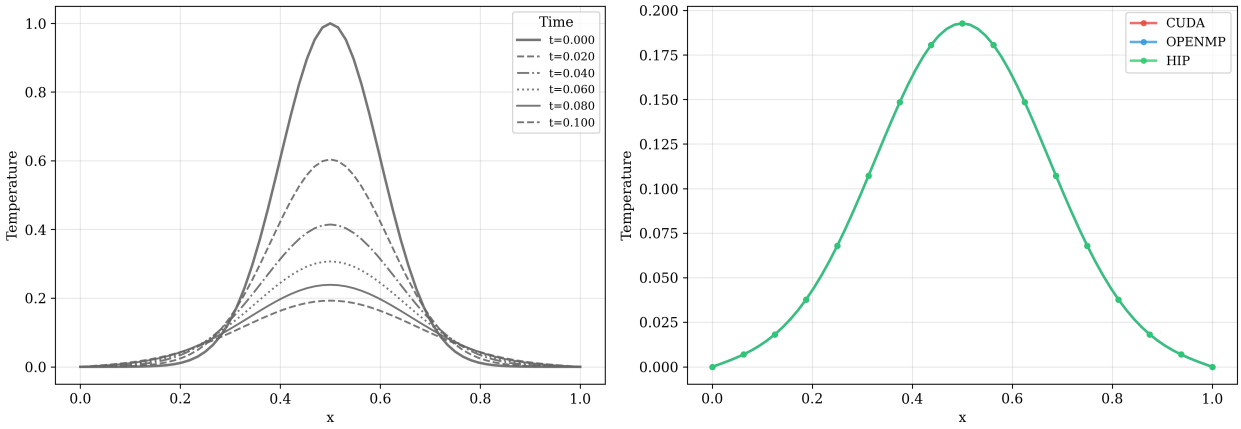


Fig. 6 Temperature profiles through the centerline ($y = 0.5$) of the domain. (Left) Temporal evolution of the Gaussian pulse at times $t = 0.000, 0.020, 0.040, 0.060, 0.080,$ and 0.100, showing smooth radial diffusion and consistent decay. (Right) Cross-platform comparison at $t = 0.100$ demonstrating that CUDA, OpenMP, and HIP produce indistinguishable temperature distributions. This verifies that the discrete diffusion operator is stable, symmetric, and backend-agnostic.

C. Burgers Equation (Nonlinear Advection Test)

1. Viscous 1D Burgers: N wave Shock Formation

We considered the nonlinear, viscous one dimensional Burgers equation

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}, \tag{25}$$

on $x \in [0, 1]$ with homogeneous Dirichlet boundary conditions

$$u(0, t) = u(1, t) = 0, \tag{26}$$

and a smooth initial condition

$$u(x, 0) = A\sin(2\pi x), \qquad A = 0.2. \tag{27}$$

For the numerical test the 2D field was initialized as

$$u(x_i, y_j, 0) = A\sin(2\pi x_i), \qquad v(x_i, y_j, 0) \equiv 0, \tag{28}$$

with viscosity set to $\nu = 0.001$. The j direction was treated as a dummy index dummy index so that the solution remains one-dimensional in x. The streamwise grid used $N_x = 512$ points (and $N_y = 512$ dummy points in the transverse direction). The domain was discretized with a 6th order compact interior stencil and 6th order boundary closures (C6–AC6–C6–AC6–C6). The soslution was advanced to $t_{final} = 1.0$, using a CFL $= \frac{\Delta t}{\Delta x} = 0.25$. At $A = 0.2$ and $\nu = 10^{-3}$ the effective Reynolds number is Re $\approx UL/\nu \approx 200$, and the resulting viscous shock has thickness $\delta = O(\nu/U) \approx 5 \times 10^{-3}$, corresponding to approximately 2.6 grid spacings for $N_x = 513$. The small viscosity regularizes the shock formation into smooth viscous shock layer with thickness $\delta = O(\nu)$, which prevents Gibbs oscillations and captures shock steepening.

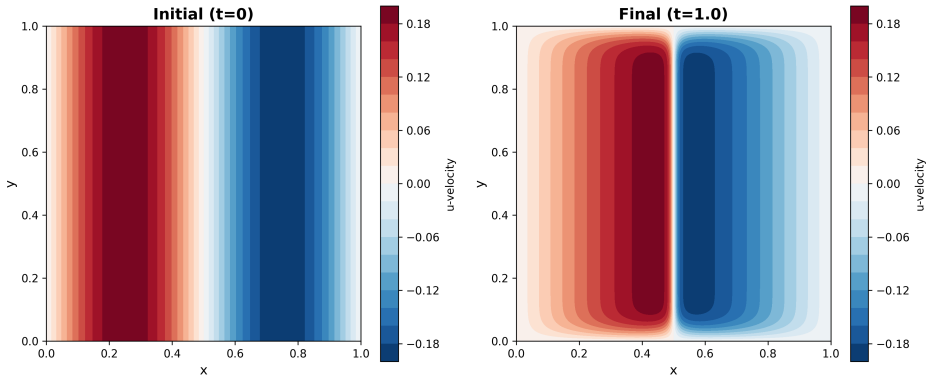


Fig. 7 Fig. 7. Viscous Burgers evolution on the 2-D solver grid. The left panel shows the initial condition $u(x, 0) = A \sin(2\pi x)$ with $A = 0.2$, extended uniformly in y. With viscosity $\nu = 0.001$, nonlinear steepening generates a smooth viscous shock centered at $x = 0.5$ by $t = 1.0$. The field remains constant in the transverse direction, confirming correct derivative assembly and boundary closures in the 2-D code path.

The numerical solution shows the expected N-wave steepening of the initial sine profile. The finite viscosity regularizes the shock into a smooth layer of thickness $\delta = O(\nu/U)$, preventing Gibbs oscillations while still capturing the nonlinear compression. The shock forms at the correct location and remains symmetric about the domain center. The energy decays monotonically in time.

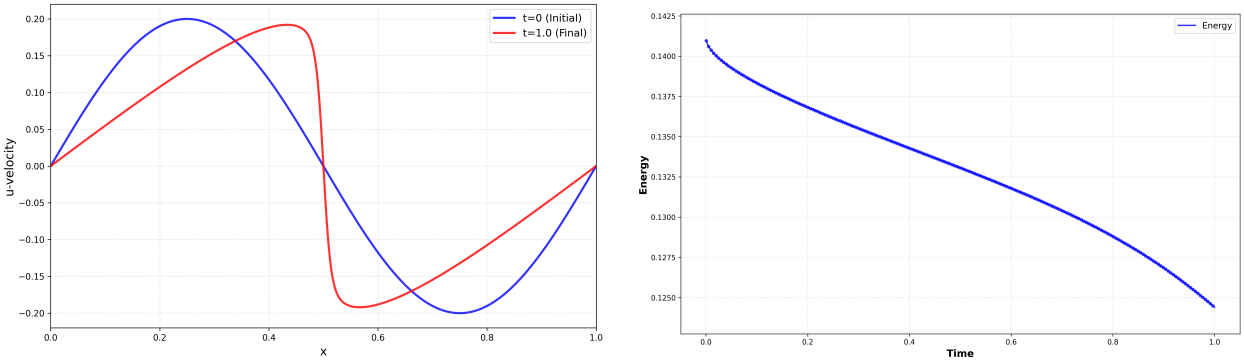


Fig. 8 Left: 1D centerline solution $u(x)$ showing formation of the viscous shock. Right: evolution of the total energy, which decreases monotonically due to viscous dissipation. No spurious oscillations are observed, indicating stable high-order reconstruction and time integration.

2. Viscous 2D Burgers: Diffusion of a Gaussian Pulse

Coupled 2D viscous Burgers system was tested.

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = \nu\nabla^2 u \tag{29}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = \nu\nabla^2 v \tag{30}$$

on $[0, 1] \times [0, 1]$ with homogeneous Dirichelet boundaries, $u = 0, v = 0$ on $\partial\Omega$. The initial condition is a Gaussian pulse, that enforces zero values at the boundaries:

$$\begin{aligned} u(x, y, 0) &= A \exp\left(-\frac{(x - \frac{1}{2})^2 + (y - \frac{1}{2})^2}{2\sigma^2}\right), \\ v(x, y, 0) &= u(x, y, 0). \end{aligned} \tag{31}$$

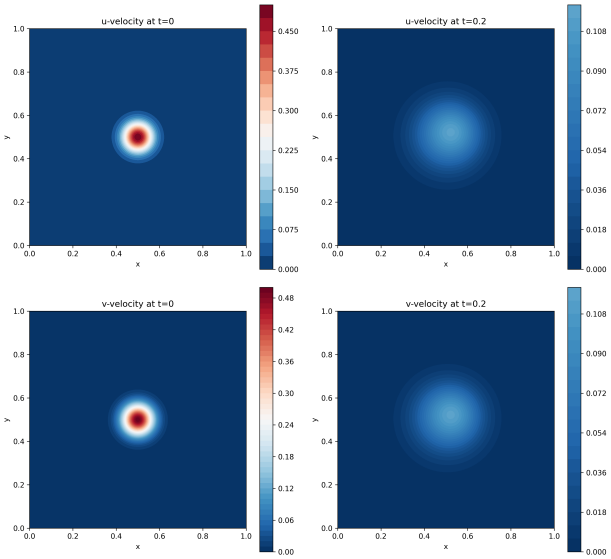


Fig. 10 Contour plots of the u and v velocity fields for the 2D viscous Burgers problem. The initially compact Gaussian pulse (left column) diffuses outward and decreases in amplitude by $t = 0.2$ (right column). Both components remain smooth and symmetric, with no spurious oscillations.

with amplitude $A = 0.5$ and width $\sigma = 0.01$. The viscosity was set to $\nu = 0.02$. A uniform 129×129 grid was used. 6th order compact stencils were employed in both directions with 6th order boundary closures. Timestep was chosen according to diffusive CFL $\nu\Delta t/\Delta x^2 = 0.25$, and convective CFL $\Delta t/\Delta x = 0.25$ as $\Delta t = \min(0.25\frac{\Delta x^2}{\nu}, 0.25\Delta x)$ the solution was advanced to $t = 0.2$. Total kinetic energy:

$$E_t = \frac{1}{N_x N_y} \sum_{i,j} u_{i,j}^2 + v_{i,j}^2 \tag{32}$$

was monitored, and seen to decay in time, as expected.

Snapshots of $u(x, y, t)$ show the initially compact pulse diffusing outwards and remaining centered in the domain and also decaying in amplitude. No spurious oscillations are observed, indicating that the high–order compact derivatives and RK4 time integration remain stable and non-oscillatory for this nonlinear convective–diffusive flow.

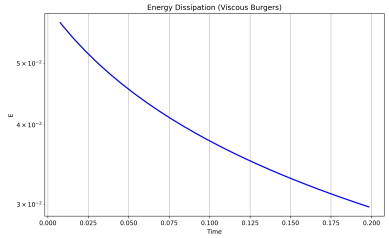


Fig. 9 Monotonic decay of the kinetic energy, shows viscous dissipation and the stability of the 6th-order compact scheme with RK4 time integration.

3. Manufactured Solution for 2D Burgers with Dirichlet BCs

Lastly, method of manufactured solutions was employed to verify the spatial accuracy of the solver for a fully coupled, source-driven problem with Dirichlet boundaries.

The 2D Burgers system is defines as:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = \nu\nabla^2 u + S_u(x, y, y) \tag{33}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = \nu\nabla^2 v + S_v(x, y, t) \tag{34}$$

on $[0, 1] \times [0, 1]$ with

$$u(x, y, t) = v(x, y, t) = \sin(\pi x) sin(\pi y) \exp(-2\pi^2 \nu t) \tag{35}$$

homogeneous Dirchlet boundary conditions are satisfied naturally at the boundaries for this setup. The source terms S_u and S_v are obtained by substituting the exact solution into the PDEs and are evaluated analytically in the code. To isolate spatial discretization error a very small timestep of $\Delta t = 10^{-5}$ was used. The simulations were run from $t = 0$ to $t_{final} = 10^{-3}$ on grid size $[32 \times 32]$,$[64 \times 64]$, $[128 \times 128]$, $[512 \times 512]$. Fourth-order schemes (E3-DE3-C4-DE3-E3 through C4-AC4-C4-AC4-C4) achieve convergence rates of 3.2-4.7, while sixth-order schemes (E5-DE5-C6-DE5-E5 through C6-AC6-C6-AC6-C6) achieve rates of 5.2-6.9 before saturating at machine precision (10^{-16}) on $N \geq 256$ grids. Reference dashed lines indicate theoretical slopes of 4th, 5th, and 6th order. All schemes demonstrate design order accuracy. See Fig. 11.

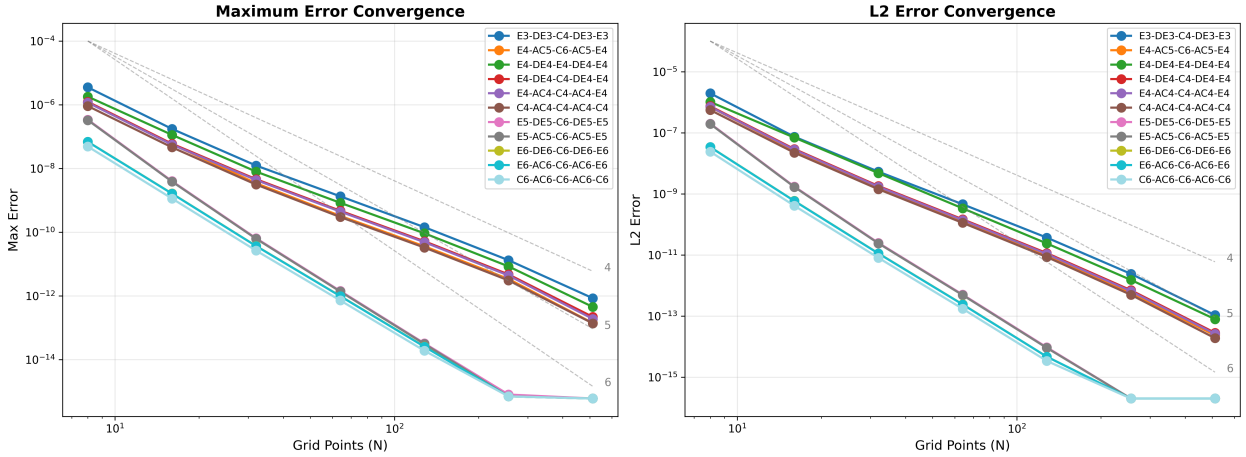


Fig. 11 Spatial accuracy verification for viscous Burgers equation using Method of Manufactured Solutions. All compact finite difference schemes achieve their theoretical convergence rates, with 6th-order schemes reaching machine precision on fine grids.

D. 2D Euler-Shu /Isentropic Vortex Propagation

The isentropic Euler vortex problem is a standard benchmark for assessing the accuracy, dispersion/dissipation characteristics, and long-time stability of high-order schemes. The test consists of superimposing an analytical vortical purturbation on to a uniform mean flow. The initial conditions are constructed by superimposing a vortical purturbation onto a unifrom mean flow. The exact solution art time t is simply the initial conditon translated by the uniform convection velocity. The numerical error is measured by comparing the solution after one or more vortex periods to the initial conditions. Following Speigel et al. [17], who surveyed the vortex problem extensively for discontinuous Galerkin and flux reconstruction methods, vortex parameters, presented in Table 4), were chosen using the standard Shu formulation [18]. The test was implemented within the framework using C4-AC4-C4-AC4-C4 and C6-AC6-C6-AC6-C6 stencils. Periodicity was enforced through the Sherman-Morrison correction, enabling a truly periodic grid with no ghost cells. Sherman Morrison requires applying tri-diagonal solver twice. PCR was applied twice. This is an important detail to consider while establishing performance estimates. The Euler solver uses fourth order RK4 time integration and positivity preserving floors on density and pressure.

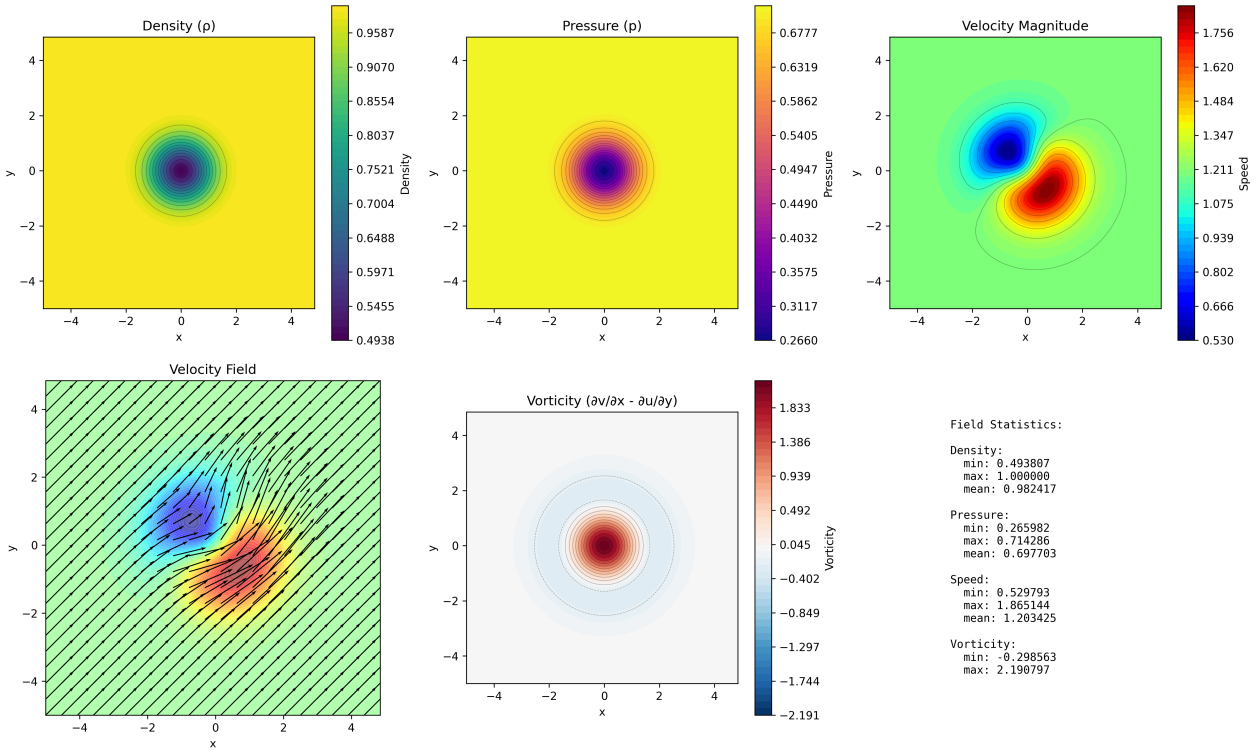


Fig. 12 Initial conditions for the isentropic Shu vortex at $t = 0$. Density, pressure, velocity magnitude, velocity vector field, and vorticity distribution over the domain $[-5, 5] \times [-5, 5]$.

Table 4 Parameters for the isentropic Shu vortex benchmark.

Quantity	Value
Ratio of specific heats, γ	1.4
Free-stream Mach number, M_∞	$\sqrt{2/\gamma}$
Free-stream velocity direction	$\alpha = 45°$
Free-stream pressure	$p_\infty = 1/\gamma$
Free-stream density	$\rho_\infty = 1$
Free-stream temperature	$T_\infty = 1$
Vortex strength parameter	$\beta = \dfrac{5M_\infty\sqrt{2}}{4\pi}\, e^{1/2}$
Core radius / length scale	$R = 1$
Gaussian width parameter	$\sigma = 1$
Domain size	$[-5, 5] \times [-5, 5]$
Period of one vortex revolution	$T = \dfrac{10\sqrt{2}}{M_\infty}$

Governing Equations

The Shu vortex evolves under the two–dimensional compressible Euler equations:

$$\frac{\partial}{\partial t}\begin{bmatrix}\rho\\ \rho u\\ \rho v\\ \rho E\end{bmatrix}+\frac{\partial}{\partial x}\begin{bmatrix}\rho u\\ \rho u^2+p\\ \rho uv\\ u(\rho E+p)\end{bmatrix}+\frac{\partial}{\partial y}\begin{bmatrix}\rho v\\ \rho uv\\ \rho v^2+p\\ v(\rho E+p)\end{bmatrix}=0. \tag{36}$$

The total specific energy is defined as

$$E = e + \frac{1}{2}(u^2 + v^2), \tag{37}$$

where e is the internal energy per unit mass. The pressure is given by the ideal-gas equation of state,

$$p = (\gamma - 1)\,\rho e,$$

or equivalently,

$$p = (\gamma - 1)\left(\rho E - \frac{1}{2}\rho(u^2 + v^2)\right). \tag{38}$$

Detailed notes on the analytical solution can be found in Spiegel et.al [17].

1. Accuracy and Convergence

The L_2 error reported represents total discretization error, including both spatial and temporal contributions. since the temporal integration is sufficiently resolved, the observed convergence sloped reflect the design spatial order of the compact schemes. For short-term simulations (up to 5 periods), both schemes achieve design order accuracy; for long-term unfiltered runs, the C4 scheme offers better robustness. Long term stability was assessed by extending simulations to 20 periods without filtering. All grids eventually experienced failure due to aliasing induced oscillations driving with failure period increasing with grid size.

The C6 scheme initially results in lower errors but experiences earlier failure caused by accumulating aliasing errors. The C4 scheme demonstrated superior stability surviving 2 to 5 additional periods on average before instability onset. This confirms that higher order schemes require more aggressive filtering for long term integration, as noted by Spiegel et al. [17] and by Visbal and Gaitonde [19].

We adopt the original Shu formulation on the domain $[-5, 5] \times [-5, 5]$ with fully periodic boundaries. As noted by Spiegel et al. [17], this domain size is not large enough for the velocity perturbations to decay to zero at the boundaries, which introduces weak artificial shear layers. For the range of resolutions considered here, these shear layers do not contaminate the observed 4th- and 6th-order convergence rates over 1–5 periods, but they do contribute to the eventual loss of stability at long times. A more stringent 'clean' verification study could employ the enlarged domain $[-10, 10] \times [-10, 10]$ recommended by Spiegel et al., however this is left for future work. Compared to the one period tests commonly reported in the literature, the present 5 to 20 period runs on a relatively small $[-5, 5]$ domain constitute a significantly harsher test of phase accuracy and nonlinear stability.

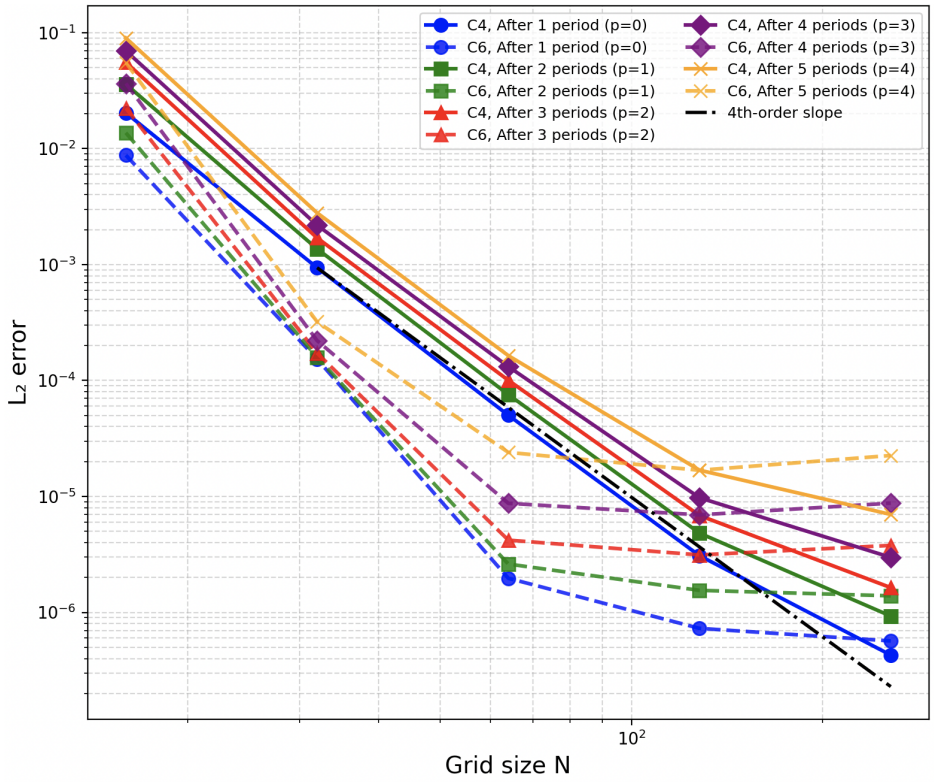


Fig. 13 Grid-refinement study for the Shu isentropic vortex showing the density error after 1 to 5 vortex periods for fourth-order (C4) and sixth-order (C6) compact schemes.

VIII. Performance Analysis

Table. 5 lists the details of the hardware platforms utilized for the performance and accuracy analysis presented in this work.

Table 5 Hardware platforms cross-architecture performance benchmarking on Purdue RCAC clusters.

Platform	Bell (CPU)	Bell (HIP)	Gautschi (CUDA)
Processor / GPU	Dual-socket AMD EPYC 7662 (2×64 cores, Zen 2)	AMD Radeon Instinct MI50	NVIDIA H100
Host memory per node	256 GB	256 GB	512GB to 1TB (node dependent)
GPU memory (per GPU)	–	32 GB HBM2	80 GB HBM3
Programming backend	OpenMP	HIP	CUDA
Tri-diagonal Solver	Thomas / PCR	Thomas / PCR	Thomas / PCR
Cluster / institution	Bell / Purdue RCAC	Bell / Purdue RCAC	Gautschi / Purdue RCAC

As explained in section V, gains were achieved in two phases. In phase one, the PCR algorithm was used in place of Thomas to solve tri-diagonal algorithms to exploit GPU parallelism. In phase two, the PCR algorithm was equipped with a variable-sized cache to reduce repeated memory initialization (memset) overhead. Next, the 3D derivative driver was restructured to replace a plane-by-plane launch strategy with a fully batched formulation.

In the original implementation, the compact derivative was applied independently to each k-plane, resulting in a separate tridiagonal solve for every slice. See Algorithm 1. This approach restricts available GPU parallelism to the j-direction and incurs repeated kernel launch and memory initialization overhead. The optimized formulation instead

flattens the (j, k) directions into a single batched dimension, enabling all tridiagonal systems associated with the $\partial/\partial\xi$ operator to be solved simultaneously in a single PCR invocation. See Algorithm 2.

This restructuring exposes $O(j_{\max} k_{\max})$ independent systems to the GPU, substantially increases effective occupancy, and amortizes solver and memory setup overhead. The combined effect of batched execution and variable-sized cache reuse yields an order-of-magnitude performance improvement relative to the original plane-by-plane formulation on both CUDA and HIP backends.

The pre-batched formulation also exhibited considerable speedups over CPU baselines (Serial and OpenMP) for single-derivative, see Appendix section X.B and Laplacian operator see Appendix section X.D. However, once time stepping is introduced, the number of kernel launches increases rapidly, leading to significant performance degradation.

Detailed speed-up tables are presented in the Appendix, section X for both implementations. Figures 14 and 15 summarize the performance improvements obtained through batched kernel launching across all backends. Both PCR and Thomas solvers benefit from batching within compact difference formulations, with consistent gains observed on CPU and GPU architectures.

It is imprtant to acknowledge that batched kernel launching introduces additional temporary storage requirements associated with solver workspaces, increasing peak memory usage relative to plane-by-plane execution.

Algorithm 1 Pre-batched 3D $\partial/\partial\xi$ driver (plane-by-plane)

for $k = 0$ to $k_{\max} - 1$ **do**
    Pack 3D slice: `phi_plane`$(j, i) \leftarrow \phi(i, j, k)$
    Solve tridiagonal system:
        `dphi_plane` $\leftarrow$ `PCR(phi_plane)`
    Unpack result:
        $\phi_\xi(i, j, k) \leftarrow$ `dphi_plane`(j, i)
end for

Algorithm 2 Batched 3D $\partial/\partial\xi$ driver (flattened j–k)

$m_{\max} \leftarrow j_{\max} \cdot k_{\max}$
$n_{\max} \leftarrow i_{\max}$
Pack 3D field:
for all (k, j, i) in parallel **do**
    $m \leftarrow j + k \cdot j_{\max}$
    `phi_lines`$(m, i) \leftarrow \phi(i, j, k)$
end for
Single batched tridiagonal solve:
    `dphi_lines` $\leftarrow$ `PCR(phi_lines)`
Unpack result:
for all (k, j, i) in parallel **do**
    $m \leftarrow j + k \cdot j_{\max}$
    $\phi_\xi(i, j, k) \leftarrow$ `dphi_lines`(m, i)
end for

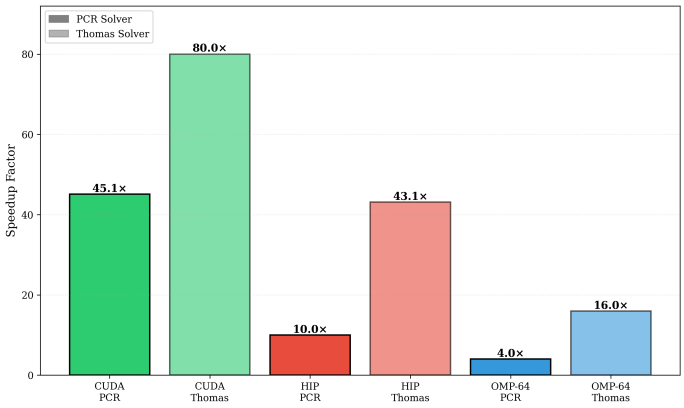


Fig. 14 Performance improvement of the PCR algorithm from batched kernel launching and variable caching at $N = 128^3$ for the 3D heat equation. Bars report the speedup factor defined as Speedup = $Time_{\text{PCR/Thomas}}/T_{\text{PCR/Thomas batched}}$, where $T_{\text{PCR/Thomas}}$ corresponds to the original plane-by-plane execution and $T_{\text{PCR/Thomas enhanced}}$ to the batched kernel launch with variable caching introduced for PCR. Results are shown for PCR and Thomas solvers across CUDA (H100), HIP (MI50), and OpenMP (64 threads) backends.

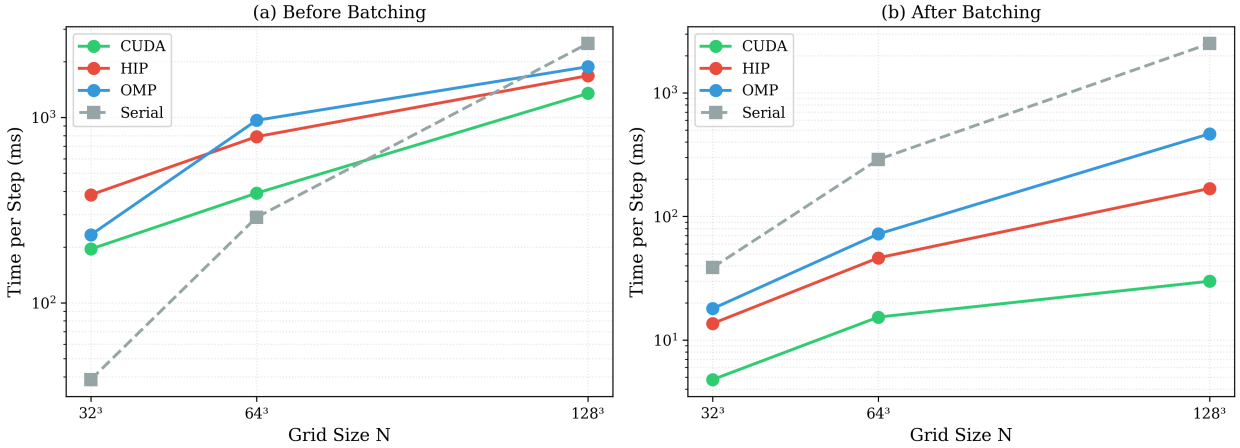


Fig. 15 Effect of batched kernel launching on PCR performance for the 3D heat equation. (a) Baseline plane-by-plane execution exhibits limited scaling due to repeated kernel launch and memory initialization overhead. (b) After batching the (j, k) dimensions into a single invocation, PCR achieves near-linear scaling with substantial reductions in time per step across CUDA, HIP, and OpenMP backends.

A. Single Derivative – Performance

The performance characteristics of the Thomas and PCR algorithm are well established. The Thomas algorithm minimizes arithmetic cost and is typically optimal on scalar or lightly threaded CPU execution, whereas PCR exposes substantial parallelism and maps efficiently to many-core and GPU architectures [14].

The results in Tables 13 to 17 are consistent with the expected behavior of tridiagonal solvers reported by Zhang et al. [14]. On serial CPUs, Thomas remains the fastest solver for both C4 and C6 schemes and PCR shows slowdown. Under OpenMP, PCR becomes competitive and achieves modest speedups at larger grid sizes as thread-level parallelism increases. On GPUs, PCR provides the highest throughput, outperforming Thomas by several factors on both CUDA and HIP backends, particularly for moderate and large three-dimensional grids where the parallel reduction steps can fully utilize device concurrency. See Appendix sections X.A and X.B.

Fig. 16 illustrates the massive gains achieved through batch kernel launching and creating variable cache storage available for reuse in the PCR algorithm implemtation.

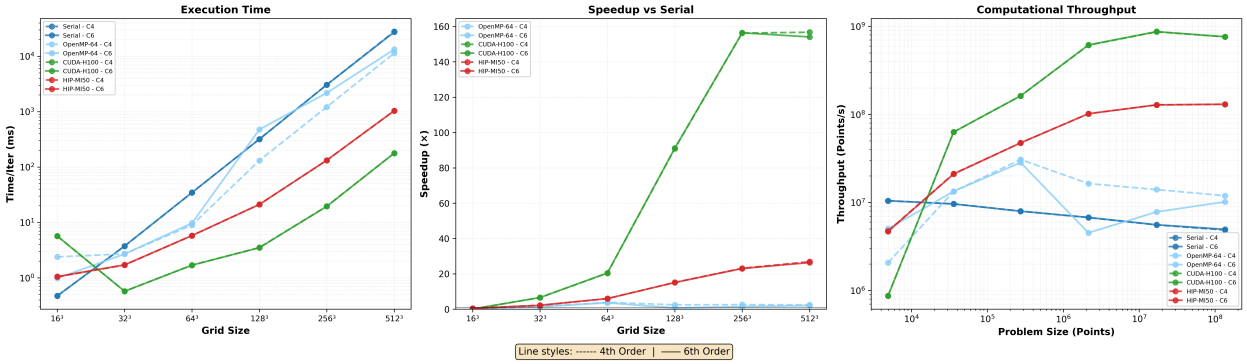


Fig. 16 Performance of the 3D compact finite-difference *single-derivative* operator (all three directions evaluated sequentially) using the PCR algorithm. Absolute execution time per iteration (left), speedup relative to the serial baseline (center), and sustained computational throughput (right) are shown for C4 and C6 schemes on Serial, OpenMP, CUDA (H100), and HIP (MI50) backends. Throughput is defined as Throughput $= \frac{N^3}{T_{\text{iter}}}$, where N^3 is the total number of grid points and T_{iter} is the average wall-clock time per full derivative sweep over all three spatial directions. The trends closely mirror the Laplacian operator results, confirming that tridiagonal solver performance dominates the overall cost.

Prior to batch launching the parallel PCR solver only delivers its full performance benefits on sufficiently large grids. As shown in Table 17, speedups of over an order of magnitude relative to the serial Thomas baseline are observed only once problem sizes are sufficiently large to amortize parallel overheads and memory traffic. On smaller problems, these overheads limit achievable gains. In practice, realizing the asymptotic performance characteristics of PCR on GPUs requires strict, problem-aware memory management strategies (batched solves, domain decomposition, and careful data layout).

In this initial implementation we have relied on CUDA unified memory and HIP managed memory for ease of portability and faster backend verification. This choice improves portability and development productivity, but restricts the deeper performance gains expected from device-resident memory management.

Achieving strong parallel performance for tri-diagonal systems required algorithmic changes: on serial architectures, the classical Thomas algorithm consistently outperforms PCR, whereas on GPU platforms the parallel PCR formulation decisively outperforms Thomas by factors of 10–20 at large grid sizes. Hardware-agnostic frameworks such as Kokkos are therefore powerful enablers of portability and code reuse, but they require careful attention to algorithm design and an awareness of underlying architectural differences to achieve true performance. Parallelizing inherently serial algorithms through such frameworks—and lowering onto CUDA, HIP, or pthreads can result in performance penalties rather than gains.

B. Laplacian Operator - Performance

The Laplacian benchmark results reinforce the trends observed in the single-derivative operator, PCR initially under performs compared to Thomas across all architectures for small grids ($N \leq 64$), which is expected given PCR's higher arithmetic intensity and larger temporary workspace requirements. As N increases, PCR begins to outperform serial Thomas around $N = 256$ on both CUDA and HIP, and the gains grow rapidly: by $N = 512$, CUDA achieves $s \approx 10.5$ relative to serial PCR and $s \approx 4.7$ relative to serial Thomas. At $N = 1024$, CUDA reaches $s \approx 42$ relative to serial PCR and $s \approx 28$ relative to serial Thomas, representing the true asymptotic benefit of PCR's parallelism in three dimensions. See Appendix section X.D.

OpenMP-64 shows a contrasting scaling behavior: while it achieves modest speedups ($s \approx 3.4$ at $N = 1024$ for C4), the gains are limited by both thread contention and NUMA effects on the underlying 64 core $\times$ 2 dual-socket test system. The relatively poor OpenMP scaling ($s = 3.4$ with 64 threads from a serial baseline) confirms that CPU-side parallelism for tridiagonal solvers faces fundamental memory bandwidth constraints even on modern architectures.

In summary, the Laplacian results demonstrate that PCR delivers substantial speedups, achieving up to 42 times over serial PCR and 28 times over serial Thomas on CUDA. The next layer of performance gains is obtained through batch launching the PCR solve kernels. This is illustrated in Fig. 17 and Appendix section X.C. For the C4 operator,

speedups of up to 295 times relative to serial Thomas on the CPU are observed on CUDA, while the C6 operator achieves speedups of up to 291 times.

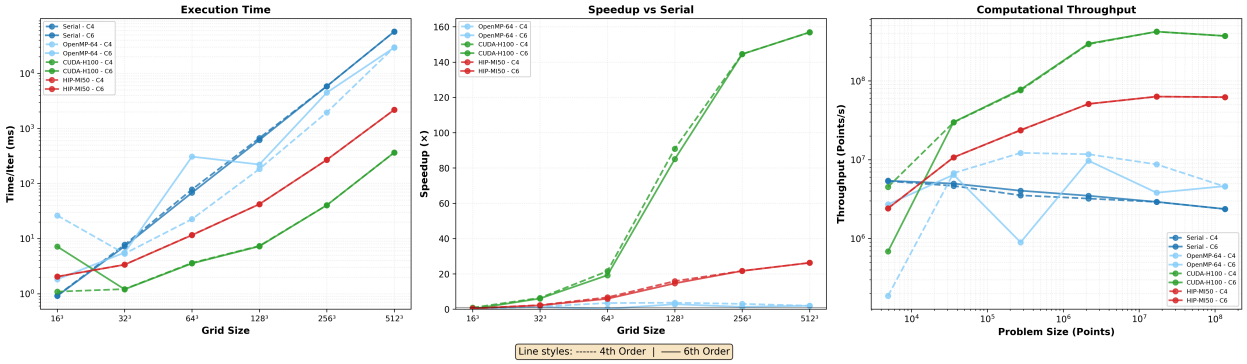


Fig. 17 Performance of the 3D compact finite-difference Laplacian (all directions) using the PCR algorithm. Absolute execution time (left), speedup over serial (center), and sustained computational throughput (right) are shown for C4 and C6 schemes on Serial, OpenMP, CUDA (H100), and HIP (MI50) backends. Throughput is defined as the number of grid points processed per second and is computed as Throughput = N^3/T_{iter}, where N^3 is the total number of grid points and T_{iter} is the average wall-clock time per Laplacian application. GPU backends achieve over two orders of magnitude speedup at large problem sizes, with CUDA sustaining the highest throughput.

C. Heat Equation Performance

Table 6 Problem size definition for the 3D Gaussian heat-diffusion test.

Grid Size (Interior Cells)	N^3 Points	Time Steps
32^3	35,937	62
64^3	274,625	246
128^3	2,146,689	984
256^3	16,974,593	3933
512^3	135,005,697	15729

The 3D heat equation benchmark exercises the complete solution pipeline: spatial derivatives via compact Laplacian operators combined with time integration. This test isolates the cost of repeated tridiagonal solves across multiple time steps, providing a realistic performance assessment for time-dependent PDE applications. The setup details are provided in section VII.B.1. The problem size is defined in Table 6.

The performance characteristics reinforce the resolution dependent trends observed in the Laplacian benchmarks, substantial GPU acceleration truly materializes at sufficiently large problem sizes where parallelism can saturate the device. For the 128^3 grid, CUDA achieves 4.6× speedup of PCR over Thomas and HIP delivers 6.4×, while smaller grids show minimal or negative speedups, parallel launch overhead dominates. So much so that there are seriously dimishing returns with increasing timestep. The tables in Appendix section X.E and section X.F provide detailed speedup comparisons.

Cross platform numerical consistency remains robust, PCR and Thomas implementations with RK4 produce identical L_2 error profiles, decreasing from 10^{-10} at 32^3 to 10^{-15} at 128^3; illustrated in Fig 18 in the Appendix X. For production time-dependent simulations, extracting GPU performance requires both large problem sizes and careful management of memory hierarchy to minimize host-device memory traffic per time step. After kernel launch overheads are amortized, the achieved performance is substantial. Total wall time for the largest grid sizes computed on each backend is reported in Table 7; corresponding accuracy and throughput summaries are shown in Fig. 18. For the 256^3

grid, the CUDA backend achieves a 30 times speedup over 64-thread OpenMP using identical C++ Kokkos code, and a 6.5 times speedup over HIP.

Future work will include weak-scaling studies to quantify performance with increasing thread and device concurrency, enabling direct comparison of scalability trends between CPU and GPU backends.

Table 7 Runtime comparison across backends for the 3D heat-diffusion test. N denotes the nominal grid size in each spatial direction (interior cells).

Backend	Grid Size N^3	Time Steps	Wall Time
CUDA	512^3 (1.35×10^8)	15729	24,593 s (6 h 50 min)
CUDA	256^3 (1.69×10^7)	3833	660 s (11 min)
HIP	256^3 (1.69×10^7)	3933	4,350 s (1 h 12 min)
OpenMP	256^3 (1.69×10^7)	3933	20,220 s (5 h 37 min)

IX. Conclusion

This work demonstrates that achieving performance portability for high-order compact finite difference schemes requires fundamental algorithmic redesign, not mere syntactic translation of sequential code. The transition from CPU to GPU architectures necessitates replacing the inherently sequential Thomas algorithm with Parallel algorithms like PCR, trading increased arithmetic cost (O(n log n) versus O(n)) for the massive parallelism required to saturate modern GPUS. Our Kokkos-based PCR implementation achieves 189 times speedup over CPU serial Thomas baselines on NVIDIA H100 GPUs and up to 32 times on AMD RadeonInstinct MI50 for 3D single derivatives, for 4th and 6th order compact schemes. Laplacian operator achieves upto 295 times speedup on NVIDIA and 49 times on AMD. The 3D heat equation achieves upto 61 times speedup on NVIDIA and 11 times on AMD per timestep.

The total runtime for the 3D heat equation with the PCR tri-diagonal algorithm for a 256^3 grid with 3933 timesteps is 5 hrs and 37 minutes on a single CPU with OpenMP backend running 64 threads and this is reduced to 11 minutes on NVIDIA CUDA backend and 1 hr and 37 minutes on the AMD HIP backend, with identical C++ Kokkos code.

Comprehensive validation through Method of Manufactured Solutions confirms that the framework maintains design-order accuracy across all test cases. Sixth-order compact schemes achieve spatial convergence rates approaching 6.0, while RK4 time integration delivers fourth-order temporal accuracy. Testing across multiple governing equations—heat diffusion, inviscid and viscous Burgers equations, and isentropic Euler vortex propagation—verifies robust performance in both linear and nonlinear regimes. Cross-platform numerical consistency is preserved to sub-percent relative agreement, with absolute L_2 errors approaching machine precision. This reproducibility demonstrates that performance portability and numerical portability are distinct challenges, both requiring explicit verification.

The current implementation's reliance on managed memory (CUDA unified memory and HIP equivalents) provides portability and development productivity but limits access to the full performance potential available through explicit device memory management. Future work will leverage architecture-specific memory optimizations: device-resident allocations, elimination of host-device migration overhead, and aggressive reduction of temporary storage.

Long-time stability analysis exposes a critical trade-off between accuracy and robustness: while sixth-order schemes achieve lower initial errors, fourth-order schemes survive 2-5 additional convection periods before aliasing-induced instability in unfiltered simulations. This confirms that higher-order methods require more aggressive filtering for extended time integration. This is consistent with established literature, it is quantified here across multiple spatial orders and test problems. For production CFD applications requiring long-time integration, scheme selection must balance instantaneous accuracy against accumulated aliasing effects.

As GPU computing matures with diverging precision requirements between AI and computational physics domains [20], the scientific computing community requires verification frameworks that detect numerical regressions before they contaminate production simulations[21].

Compiler optimizations increasingly target machine learning workloads, where approximate arithmetic and relaxed IEEE compliance accelerate training. [8] These same optimizations can degrade accuracy in high-order CFD schemes, corrupting convergence rates that directly determine solution fidelity [22]. Maintaining bit-for-bit reproducibility across

evolving compiler tool-chains and diverging hardware architectures requires verification testing not as a one time validation exercise, but as sustained infrastructure analogous to established numerical libraries.

The presented framework provides verified building blocks for performance-portable CFD applications requiring high-order accuracy on heterogeneous architectures.

References

[1] Lele, S. K., "Compact finite difference schemes with spectral-like resolution," *Journal of computational physics*, Vol. 103, No. 1, 1992, pp. 16–42.

[2] Kravchenko, A., and Moin, P., "On the effect of numerical errors in large eddy simulations of turbulent flows," *Journal of computational physics*, Vol. 131, No. 2, 1997, pp. 310–322.

[3] Gaitonde, D. V., and Visbal, M. R., "High-Order Schemes for Navier–Stokes Equations: Algorithm and Implementation into FDL3DI," Tech. Rep. AFRL-VA-WP-TR-1998-3060, Air Force Research Laboratory, Air Vehicles Directorate, Wright-Patterson Air Force Base, OH, 1998.

[4] Poggie, J., "Compact Difference Methods for Discharge Modeling in Aerodynamics," *40th AIAA Plasmadynamics and Lasers Conference*, 2009. AIAA Paper 2009-3908.

[5] Trott, C. R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., Gayatri, R., Harvey, E., Hollman, D. S., Ibanez, D., et al., "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 33, No. 4, 2021, pp. 805–817.

[6] Hammond, S. D., Trott, C. R., Ibanez, D., and Sunderland, D., "Profiling and debugging support for the kokkos programming model," *International Conference on High Performance Computing*, Springer, 2018, pp. 743–754.

[7] Edwards, H. C., Trott, C. R., and Sunderland, D., "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of parallel and distributed computing*, Vol. 74, No. 12, 2014, pp. 3202–3216.

[8] Dongarra, J., Gunnels, J., Bayraktar, H., Haidar, A., and Ernst, D., "Hardware trends impacting floating-point computations in scientific applications," *arXiv preprint arXiv:2411.12090*, 2024.

[9] Thomas, P. D., and Lombard, C. K., "Geometric conservation law and its application to flow computations on moving grids," *AIAA journal*, Vol. 17, No. 10, 1979, pp. 1030–1037.

[10] Poggie, J., "High-Order Compact Difference Methods for Glow Discharge Modeling," , 2009.

[11] Wolfe, M. J., *High performance compilers for parallel computing*, Addison-Wesley Longman Publishing Co., Inc., 1995.

[12] Guide, D., "Cuda c++ programming guide," *NVIDIA, July*, 2020.

[13] Hockney, R. W., "A fast direct solution of Poisson's equation using Fourier analysis," *Journal of the ACM (JACM)*, Vol. 12, No. 1, 1965, pp. 95–113.

[14] Zhang, Y., Cohen, J., and Owens, J. D., "Fast tridiagonal solvers on the GPU," *ACM Sigplan Notices*, Vol. 45, No. 5, 2010, pp. 127–136.

[15] Mejstrik, T., "_host_ _device_–Generic programming in Cuda," *arXiv preprint arXiv:2309.03912*, 2023.

[16] Davis, J. H., Sivaraman, P., Kitson, J., Parasyris, K., Menon, H., Minn, I., Georgakoudis, G., and Bhatele, A., "Taking GPU programming models to task for performance portability," *Proceedings of the 39th ACM International Conference on Supercomputing*, 2025, pp. 776–791.

[17] Spiegel, S. C., Huynh, H. T., and DeBonis, J. R., "A Survey of the Isentropic Euler Vortex Problem Using High-Order Methods," *22nd AIAA Computational Fluid Dynamics Conference*, 2015. AIAA Paper 2015-2444.

[18] Shu, C.-W., "Essentially non-oscillatory and weighted essentially non-oscillatory schemes for hyperbolic conservation laws," *Advanced Numerical Approximation of Nonlinear Hyperbolic Equations: Lectures given at the 2nd Session of the Centro Internazionale Matematico Estivo (CIME) held in Cetraro, Italy, June 23–28, 1997*, Springer, 2006, pp. 325–432.

[19] Visbal, M. R., and Gaitonde, D. V., "On the use of higher-order finite-difference schemes on curvilinear and deforming meshes," *Journal of Computational Physics*, Vol. 181, No. 1, 2002, pp. 155–185.

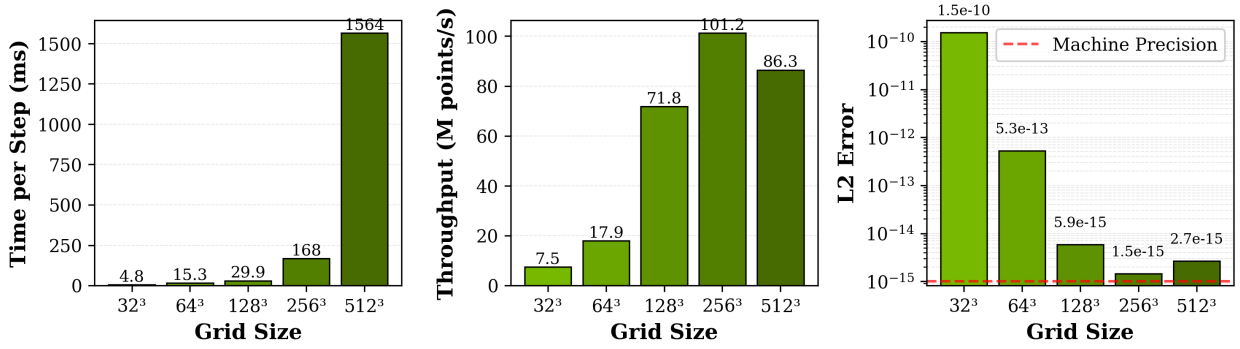
[20] NVIDIA, "A New Precision Format for Deep Learning: TensorFloat-32 (TF32)," , 2020. NVIDIA Technical Blog.

[21] Zou, D., Zeng, M., Xiong, Y., Fu, Z., Zhang, L., and Su, Z., "Detecting floating-point errors via atomic conditions," *Proceedings of the ACM on Programming Languages*, Vol. 4, No. POPL, 2019, pp. 1–27.

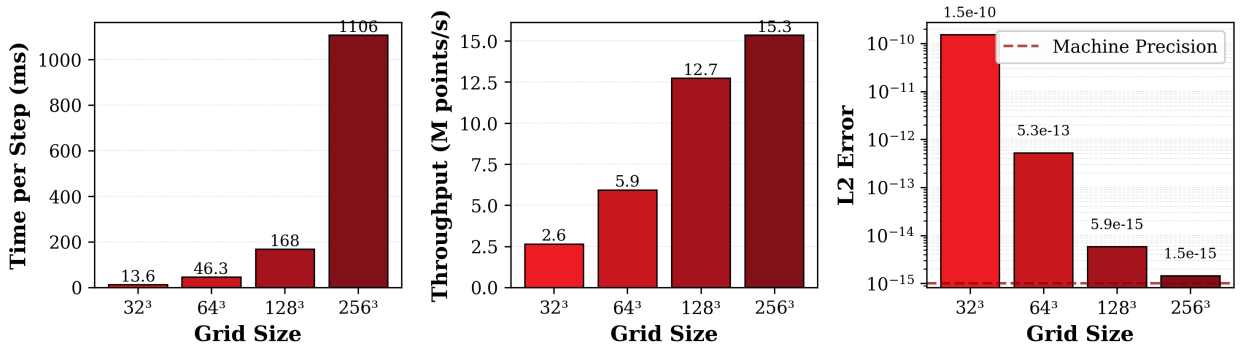
[22] Kashi, A., Lu, H., Brewer, W., Rogers, D., Matheson, M., Shankar, M., and Wang, F., "Mixed-precision numerics in scientific applications: survey and perspectives," *arXiv preprint arXiv:2412.19322*, 2024.

X. Appendix

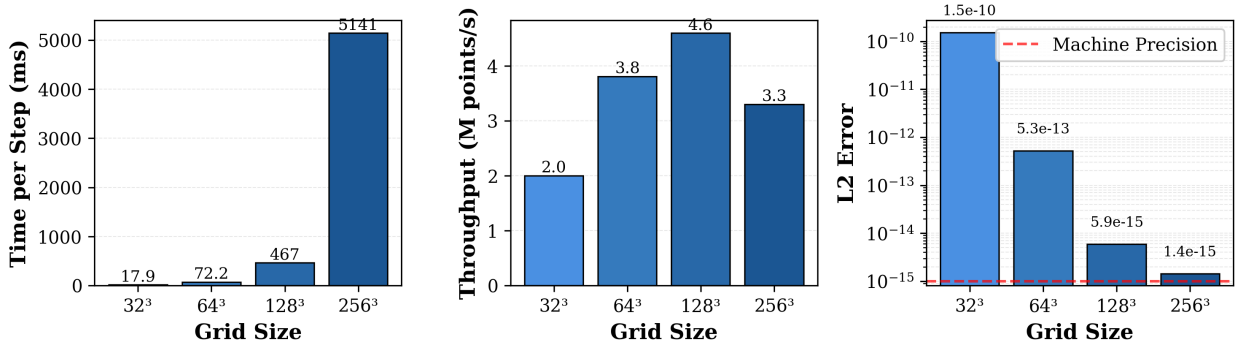
This appendix provides detailed performance details for the compact derivative implementation, written in C++ Kokkos. Sections X.A to X.F provide speedup comparisons for single derivatives computed in all three directions, Laplacian operator and the heat equations benchmark, which solves the three-dimensional heat equation using RK4 time integration and a sixth-order compact finite-difference scheme (C6-AC6-C6-AC6-C6). Performance of the PCR and Thomas tridiagonal solvers is compared across OpenMP (64 threads), CUDA, and HIP backends. The computing hardware used for these experiments is summarized in Table 5. Speedup is defined as $s = \frac{t_{\text{baseline algorithm/backend}}}{t_{\text{algorithm/backend}}}$. Values in green (greater than 1) indicate a speedup, while values in red (less than 1) indicate a slowdown.



(a) NVIDIA H100 (CUDA)



(b) AMD MI50 (HIP)



(c) OpenMP (64 threads)

Fig. 18 Performance and accuracy of the sixth-order compact finite-difference scheme with a parallel cyclic reduction (PCR) solver across different execution backends. Shown are the time per timestep, computational throughput (in millions of grid points per second), and L2 error for increasing three-dimensional grid sizes.

A. Single Derivative Results : With Batch Kernel Launch

Table 8 PCR vs Thomas speedup on GPUs. $s = t_{\text{GPU Thomas}}/t_{\text{GPU PCR}}$

	CUDA		HIP	
N	C4	C6	C4	C6
16	0.94	0.95	5.05	5.01
32	5.18	5.39	3.86	3.84
64	4.43	4.43	2.30	2.30
128	3.78	3.81	1.82	1.82
256	2.20	2.22	1.64	1.66
512	2.09	2.09	1.60	1.59
1024	—	—	—	—

Table 9 PCR vs Thomas speedup on CPUs. $s = t_{\text{CPU Thomas}}/t_{\text{CPU PCR}}$

	OpenMP-64		Serial	
N	C4	C6	C4	C6
16	1.05	2.90	0.62	0.57
32	2.31	2.46	0.48	0.46
64	2.40	2.79	0.49	0.46
128	1.36	0.50	0.72	0.73
256	1.49	2.00	0.75	0.75
512	2.24	2.42	1.21	1.21
1024	—	—	—	—

Table 10 Speedup of the parallel PCR single-derivative operator (Direction = ALL, C4) on CUDA, HIP, and OpenMP-64 backends relative to the serial Thomas baseline. Here, speedup = $t_{\text{serial, Thomas}}/t_{\text{parallel, PCR}}$. Values > 1 (green) indicate that parallel PCR is faster than serial Thomas.

	CUDA		HIP		OpenMP-64	
N	C4	C6	C4	C6	C4	C6
16	0.05	0.05	0.28	0.26	0.12	0.28
32	3.12	3.00	1.05	1.00	0.66	0.64
64	9.91	9.50	2.90	2.78	1.87	1.66
128	65.40	66.55	10.88	11.02	1.75	0.49
256	116.92	116.71	17.30	17.17	1.89	1.05
512	189.19	187.06	32.39	32.01	2.96	2.49
1024	—	—	—	—	—	—

Table 11 Speedup of PCR single-derivative operator relative to Serial PCR $s = t_{\text{serial PCR}}/t_{\text{parallel PCR}}$.

	CUDA		HIP		OMP-64	
N	C4	C6	C4	C6	C4	C6
16	0.08	0.08	0.45	0.45	0.20	0.48
32	6.54	6.56	2.19	2.19	1.38	1.39
64	20.38	20.46	5.96	5.98	3.85	3.57
128	90.86	91.06	15.12	15.08	2.43	0.67
256	156.27	156.42	23.12	23.01	2.52	1.40
512	156.73	154.07	26.83	26.37	2.45	2.05
1024	—	—	—	—	—	—

Table 12 Speedup of PCR single-derivative operator relative to OMP-64. $s = t_{\text{CPU PCR}}/t_{\text{GPU PCR}}$

	CUDA		HIP	
N	C4	C6	C4	C6
16	0.42	0.17	2.29	0.93
32	4.74	4.72	1.59	1.57
64	5.29	5.73	1.55	1.67
128	37.36	136.22	6.22	22.55
256	61.97	111.57	9.17	16.41
512	63.93	75.01	10.94	12.84
1024	—	—	—	—

B. Single Derivative Results : Without Batch Kernel Launch

Table 13 Speedup of PCR single-derivative operator relative to Serial PCR $s = t_{\text{serial PCR}}/t_{\text{parallel PCR}}$.

	CUDA		HIP		OMP-64	
N	C4	C6	C4	C6	C4	C6
16	0.07	0.07	0.04	0.04	0.09	0.09
32	0.20	0.20	0.10	0.10	0.18	0.16
64	0.73	0.72	0.36	0.36	0.50	0.47
128	1.85	1.88	1.50	1.50	0.81	0.86
256	4.31	4.35	5.53	5.51	1.13	1.09
512	10.35	10.39	6.93	6.98	2.07	2.13
1024	38.88	38.90	—	—	3.02	1.52

Table 14 Speedup of PCR single-derivative operator relative to OMP-64. $s = t_{\text{CPU PCR}}/t_{\text{GPU PCR}}$

	CUDA		HIP	
N	C4	C6	C4	C6
16	0.73	0.73	0.42	0.42
32	1.13	1.22	0.58	0.62
64	1.46	1.54	0.73	0.77
128	2.29	2.20	1.85	1.75
256	3.83	3.98	4.91	5.04
512	4.99	4.88	3.34	3.28
1024	12.86	25.59	—	—

Table 15 PCR vs Thomas speedup on GPUs. $s = t_{\text{GPU Thomas}}/t_{\text{GPU PCR}}$

	CUDA		HIP	
N	C4	C6	C4	C6
16	1.71	1.69	1.81	1.79
32	3.16	3.15	2.59	2.57
64	5.56	5.51	4.51	4.48
128	6.46	6.49	8.07	7.98
256	6.32	6.39	13.90	13.81
512	6.42	6.41	8.30	8.26
1024	9.83	9.86	—	—

Table 16 PCR vs Thomas speedup on CPUs. $s = t_{\text{CPU Thomas}}/t_{\text{CPU PCR}}$

	OpenMP-64		Serial	
N	C4	C6	C4	C6
16	3.80	4.72	1.20	1.19
32	8.09	7.79	0.94	0.94
64	13.83	12.22	0.64	0.64
128	9.92	10.03	0.50	0.49
256	5.10	5.12	0.40	0.40
512	3.10	3.09	0.53	0.48
1024	3.12	0.97	0.58	0.59

Table 17 Speedup of the parallel PCR single-derivative operator (Direction = ALL, C4) on CUDA, HIP, and OpenMP-64 backends relative to the serial Thomas baseline. Here, speedup $= t_{\text{serial, Thomas}}/t_{\text{parallel, PCR}}$**. Values** > 1 **(green) indicate that parallel PCR is faster than serial Thomas.**

N	CUDA	HIP	OpenMP-64
16	0.08	0.05	0.11
32	0.19	0.10	0.17
64	0.46	0.23	0.32
128	0.92	0.74	0.40
256	1.70	2.19	0.45
512	5.43	3.64	1.09
1024	22.74	—	1.77

C. Laplacian Operator Results: With Batch Kernel Launch

Table 18 Speedup of compact Laplacian PCR relative to Serial PCR.
$s = t_{\text{serial PCR}}/t_{\text{parallel PCR}}$.

	CUDA		HIP		OMP-64	
N	C4	C6	C4	C6	C4	C6
	0.84	0.13	0.45	0.45	0.03	0.50
	6.39	6.03	2.30	2.15	1.45	1.30
	21.52	19.21	6.69	5.88	3.45	0.22
	90.91	84.99	15.87	14.62	3.65	2.79
	144.55	144.47	21.67	21.66	2.99	1.31
	156.82	156.90	26.25	26.26	1.93	1.95
	—	—	—	—	—	—

Table 19 Speedup of compact Laplacian PCR relative to OMP-64 PCR.
$s = t_{\text{OMP-64 PCR}}/t_{\text{GPU PCR}}$.

	CUDA		HIP	
N	C4	C6	C4	C6
	24.15	0.25	12.90	0.89
	4.40	4.65	1.58	1.66
	6.23	86.78	1.94	26.58
	24.90	30.50	4.35	5.25
	48.27	110.56	7.24	16.58
	81.46	80.39	13.64	13.46
	—	—	—	—

Table 20 PCR vs Thomas speedup for Laplacian on GPUs.
$s = t_{\text{GPU Thomas}}/t_{\text{GPU PCR}}$.

	CUDA		HIP	
N	C4	C6	C4	C6
	8.62	1.31	2.66	2.68
	5.36	5.34	3.27	3.26
	4.24	4.33	2.29	2.30
	3.67	3.72	1.82	1.82
	2.16	2.17	1.63	1.63
	2.05	2.05	1.52	1.52
	—	—	—	—

Table 21 PCR vs Thomas speedup for Laplacian on CPUs.
$s = t_{\text{CPU Thomas}}/t_{\text{CPU PCR}}$.

	OMP-64		Serial	
N	C4	C6	C4	C6
	12.02	41.05	0.62	0.62
	2.03	2.20	0.45	0.47
	10.57	0.24	0.43	0.55
	2.37	4.60	0.69	0.74
	3.96	0.84	0.77	0.77
	1.79	2.17	1.88	1.86
	—	—	—	—

Table 22 Speedup of the parallel PCR Laplacian operator (C4 and C6) on CUDA, HIP, and OMP-64 backends relative to the serial Thomas baseline.
$s = t_{\text{serial, Thomas}}/t_{\text{parallel, PCR}}$. **Values > 1 (green) indicate that parallel PCR is faster than serial Thomas.**

	CUDA		HIP		OMP-64	
N	C4	C6	C4	C6	C4	C6
16	0.52	0.08	0.28	0.27	0.02	0.31
32	2.89	2.83	1.04	1.01	0.66	0.61
64	9.18	10.55	2.85	3.23	1.47	0.12
128	62.47	63.00	10.91	10.84	2.51	2.07
256	111.90	110.62	16.78	16.59	2.32	1.00
512	295.44	291.49	49.46	48.79	3.63	3.63
1024	—	—	—	—	—	—

D. Laplacian Operator Results: Without Batch Kernel Launch

Table 23 Speedup of compact Laplacian PCR relative to Serial PCR.
$s = t_{\text{serial PCR}}/t_{\text{parallel PCR}}$.

	CUDA		HIP		OMP-64	
N	C4	C6	C4	C6	C4	C6
16	0.08	0.07	0.04	0.03	0.09	0.09
32	0.20	0.20	0.10	0.10	0.11	0.16
64	0.74	0.73	0.36	0.36	0.31	0.39
128	1.89	1.74	1.46	1.47	0.90	1.08
256	4.41	3.75	5.39	5.39	1.37	1.33
512	10.49	8.28	6.88	6.83	3.08	2.12
1024	42.05	35.85	—	—	3.40	2.23

Table 24 Speedup of compact Laplacian PCR relative to OMP-64 PCR.
$s = t_{\text{OMP-64 PCR}}/t_{\text{GPU PCR}}$.

	CUDA		HIP	
N	C4	C6	C4	C6
16	0.84	0.75	0.41	0.38
32	1.72	1.25	0.86	0.63
64	2.35	1.86	1.14	0.92
128	2.10	1.60	1.62	1.35
256	3.22	2.82	3.93	4.06
512	3.41	3.90	2.24	3.22
1024	12.36	16.10	—	—

Table 25 PCR vs Thomas speedup for Laplacian on GPUs.
$s = t_{\text{GPU Thomas}}/t_{\text{GPU PCR}}$.

	CUDA		HIP	
N	C4	C6	C4	C6
16	2.01	1.71	1.63	1.39
32	3.15	3.18	2.53	2.57
64	5.55	5.54	4.38	4.47
128	6.74	6.15	7.90	7.96
256	6.58	5.63	13.71	13.85
512	6.68	5.31	8.36	8.37
1024	9.89	8.20	—	—

Table 26 PCR vs Thomas speedup for Laplacian on CPUs.
$s = t_{\text{CPU Thomas}}/t_{\text{CPU PCR}}$.

	OMP-64		Serial	
N	C4	C6	C4	C6
16	5.65	5.32	1.16	1.17
32	5.20	8.08	0.92	0.92
64	7.08	10.53	0.63	0.63
128	7.56	12.98	0.48	0.48
256	5.60	5.53	0.39	0.39
512	4.51	3.08	0.45	0.50
1024	2.21	1.67	0.67	0.64

Table 27 Speedup of the parallel PCR Laplacian operator (C4) on CUDA, HIP, and OMP-64 backends relative to the serial Thomas baseline.
$s = t_{\text{serial, Thomas}}/t_{\text{parallel, PCR}}$. **Values** > 1 **(green) indicate that parallel PCR is faster than serial Thomas.**

N	CUDA	HIP	OMP-64
16	0.09	0.04	0.11
32	0.18	0.09	0.11
64	0.46	0.22	0.20
128	0.91	0.70	0.43
256	1.72	2.11	0.54
512	4.67	3.06	1.37
1024	28.03	—	2.27

E. Heat Equation Results: With Batch Kernel Launch

Table 28 Speedup of heat equation PCR solver relative to Serial PCR.
$s = t_{\text{serial PCR}}/t_{\text{parallel PCR}}$.

N	CUDA	HIP	OMP-64
32	6.71	2.36	1.79
64	19.65	6.52	4.18
128	92.50	16.42	5.93
256	—	—	—

Table 29 Speedup of heat equation Thomas solver relative to Serial Thomas.
$s = t_{\text{serial Thomas}}/t_{\text{parallel Thomas}}$.

N	CUDA	HIP	OMP-64
32	0.63	0.36	0.33
64	2.73	1.57	1.01
128	17.03	6.04	1.59
256	—	—	—

Table 30 Speedup of heat equation PCR solver relative to OMP-64 PCR.
$s = t_{\text{OMP-64 PCR}}/t_{\text{GPU PCR}}$.

N	CUDA	HIP
32	3.75	1.32
64	4.70	1.56
128	15.61	2.77
256	30.65	—

Table 31 Speedup of heat equation Thomas solver relative to OMP-64 Thomas.
$s = t_{\text{OMP-64 Thomas}}/t_{\text{GPU Thomas}}$.

N	CUDA	HIP
32	1.92	1.11
64	2.70	1.55
128	10.72	3.81
256	—	—

Table 32 PCR vs Thomas speedup for heat equation on GPUs.
$s = t_{\text{GPU Thomas}}/t_{\text{GPU PCR}}$.

N	CUDA	HIP
32	5.32	3.23
64	3.95	2.28
128	3.62	1.81
256	2.11	—

Table 33 PCR vs Thomas speedup for heat equation on CPUs.
$s = t_{\text{CPU Thomas}}/t_{\text{CPU PCR}}$.

N	OMP-64	Serial
32	2.73	0.50
64	2.27	0.55
128	2.49	0.67
256	—	—

Table 34 Speedup of the parallel PCR heat equation solver on CUDA, HIP, and OMP-64 backends relative to the serial Thomas baseline.
$s = t_{\text{serial, Thomas}}/t_{\text{parallel, PCR}}$. **Values** > 1 **(green) indicate that parallel PCR is faster than serial Thomas.**

N	CUDA	HIP	OMP-64
32	3.35	1.18	0.81
64	10.81	3.59	2.27
128	61.63	10.94	3.35
256	—	—	—

F. Heat Equation Results: Without Batched Kernel Launch

Table 35 Speedup of heat equation PCR solver relative to Serial PCR.
$s = t_{\text{serial PCR}}/t_{\text{parallel PCR}}$.

N	CUDA	HIP	OMP-64
32	0.20	0.10	0.17
64	0.74	0.37	0.30
128	1.86	1.49	1.33
256	—	—	—

Table 36 Speedup of heat equation Thomas solver relative to Serial Thomas.
$s = t_{\text{serial Thomas}}/t_{\text{parallel Thomas}}$.

N	CUDA	HIP	OMP-64
32	0.06	0.04	0.03
64	0.09	0.06	0.03
128	0.14	0.09	0.07
256	—	—	—

Table 37 Speedup of heat equation PCR solver relative to OMP-64 PCR.
$s = t_{\text{OMP-64 PCR}}/t_{\text{GPU PCR}}$.

N	CUDA	HIP
32	1.19	0.61
64	2.47	1.22
128	1.39	1.12
256	—	—

Table 38 Speedup of heat equation Thomas solver relative to OMP-64 Thomas.
$s = t_{\text{OMP-64 Thomas}}/t_{\text{GPU Thomas}}$.

N	CUDA	HIP
32	1.95	1.24
64	2.88	1.83
128	2.14	1.41
256	—	—

Table 39 PCR vs Thomas speedup for heat equation on GPUs.
$s = t_{\text{GPU Thomas}}/t_{\text{GPU PCR}}$.

N	CUDA	HIP
32	3.15	2.51
64	5.54	4.32
128	6.42	7.83
256	—	—

Table 40 PCR vs Thomas speedup for heat equation on CPUs.
$s = t_{\text{CPU Thomas}}/t_{\text{CPU PCR}}$.

N	OMP-64	Serial
32	5.13	0.95
64	6.45	0.66
128	9.88	0.50
256	—	—

Table 41 Parallel PCR vs Serial Thomas.
$s = t_{serialThomas}/t_{parallelPCR}$.

N	CUDA	HIP	OMP-64
32	0.19	0.10	0.16
64	0.49	0.24	0.20
128	0.92	0.74	0.66