Crafting software in HPC: a quantitative approach

By D. Rossinelli, T. Trabacchin†, A. Voci, L. Brown, H. Collis, M. Khanwale, T. Zahtila, D. Brouzet AND G. Iaccarino

Resolution is a determining factor of simulation quality and is ultimately bounded by the fraction of nominal peak performance we are able to extract. This study reports on the rapid development of Lean Vertical Rewrite (LVR), a compressible flow solver engineered to reproduce a subset of flow physics modeled by HTR, a multi-physics Legion-based CFD solver. Benchmarks demonstrate improvements approaching one order of magnitude in time to solution and fractional peak performance on representative workloads. We also report on the rapid deployment and efficient utilization of LVR on Argonne National Laboratory (ANL)'s Aurora exascale system, which is notable for its peculiar GPU architecture. Recognizing the substantial impact of acquiring and effectively leveraging underutilized computing resources, we delineate the guidelines employed to develop LVR. These guidelines, which have proven robust over a decade of disruptive advancements and diversification in GPU and CPU architectures, represent a significant departure from conventional methodologies.

1. Introduction

High-order discretization of the governing equations is a cornerstone of high-fidelity CFD. Such methods typically use wider stencils and richer operators than second-order schemes, which raises computational cost and in turn requires more effective use of the underlying computing system. As such, large-scale scientific software demands pursuit of the following goals: high fractions of peak performance, high performance portability, and high software productivity. Performance portability means that high performance is maintained across diverse computing systems, with essentially the same code base, and software productivity means that new modeling approaches, discretization schemes, and workflows can be rapidly included.

High performance here is the ability to turn nominal peak performance into delivered throughput; accordingly, a measure of how effectively a computing system is exploited is given by the achieved fraction of peak, which is defined as:

$$\%\text{ of Peak} = \frac{\text{FLOP count}}{\text{TTS}} \cdot \frac{100}{\text{Peak}}, \tag{1.1}$$

where Peak is the nominal FP64 peak performance of the underlying computing system, TTS is the time to solution, and FLOP count is the overall FP64 arithmetic operations exhibited by the algorithm (although the implementation FLOP count could be substantially higher). Unless otherwise specified, the connection between % of peak and TTS throughout this report, is therefore direct. Instead, metrics such as weak and strong scaling, often reported in compressible CFD literature, may be misleading, as they are relative performance metrics. On their own, they provide little information regarding attainable TTS or hardware utilization.

† Politecnico di Milano.

The evidence on achieved fractions of peak performance for compressible CFD is scarce, but published work suggests that existing state-of-the-art solvers generally reach up to 2-5% (Decyk *et al.* 1996; Gropp 1997; Kaushik *et al.* 1998; Cameron *et al.* 2005; Di Renzo *et al.* 2020), with outliers around 13% regarded as exceptional successes (Fu *et al.* 2023). Fragility and heterogeneity of the exascale landscape complicate the issue even further: slight changes in the instruction stream can incur severe performance penalties, and microarchitectural diversity often forces source-level specialization.

Over the last decade, the rapid evolution and diversification of CPU and GPU architectures have made the simultaneous pursuit of the aforementioned goals more challenging for the programmer. Complexity rises further when orchestrating computation and data transfers at the system level, but this burden promises large payoffs due to game-changing performance improvements introduced in the past decade. Chiplets, multi-die GPUs, DMA engines, GPU-to-GPU links, and process placement require careful coordination and awareness of the system architecture. Despite hardware diversity, instruction streams that deliver high performance share similar traits across execution units of different systems: they must be rich in datapath and sporadic in control flow, feature dense tensor operations, and issue few, regular, and contiguous memory accesses.

The unifying trait of supercomputing software is the need to accommodate an abstraction layer to distribute work across HPC nodes. Decisions made at this layer have large repercussions, because the chosen workload decomposition and memory layout determine the attainable performance downstream in the abstraction.

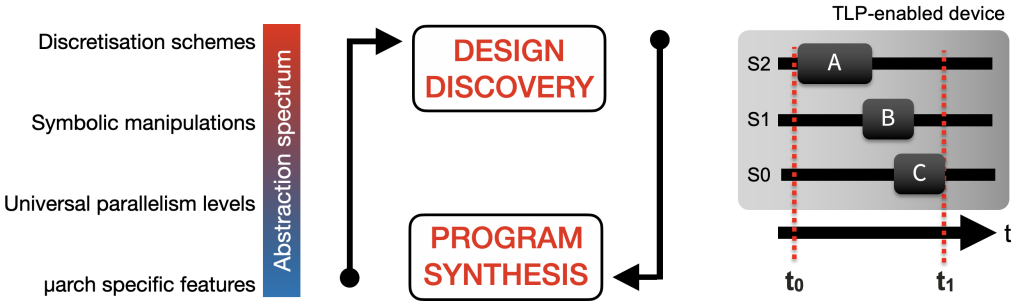
The overarching challenge lies not within devising any single abstraction layer, but in building a system that allows layers to function as intended. Our core premise is that such synergies are discovered through quantitative baselines and cannot be easily designed in advance. Despite a substantial intersection of features across systems, we recognize that offering guidelines across architectures is difficult and met with skepticism. The endeavor remains crucial, nonetheless, because turning performance into solution quality has broad impact and relevance. Here, we report what has worked for us in CFD and beyond. We also discuss practices that have repeatedly failed to serve our cases over more than a decade, yet persist as common wisdom. These guidelines do not claim to be exhaustive; they offer points of consideration for building systems with performance robustness in mind.

1.1. *Contributions of the present work*

The primary contribution is a set of guidelines for crafting software that leverages the throughput of the underlying system. We also present their net effect in compressible flow simulations on HPC infrastructure. Specifically, our software

- achieves up to 30% of the nominal peak performance of the underlying platforms, overall.
- achieves bursts up to 50% of peak across more than half of the duty cycle,
- achieves these results on diverse CPU and GPU architectures,
- has been built within 2–3 months with a group of 2–3 researchers.

showing that the challenges discussed above are addressable in practice and doing so has produced results that radically outperform prior works. We compare performance against HTR (Di Renzo *et al.* 2020), a compressible flow solver targeting the same physics and employing the same numerical schemes, and we assess performance on Aurora at Argonne National Laboratory, a notoriously challenging exascale system owing to its unconventional hardware and software ecosystem.



(a) (b)

FIGURE 1. Bottom-up quantitative approach to discover successful HPC software designs, and subsequent top-down program synthesis to expand the software scope (left). Timeline view of an oversubscribed, threaded device (right). Kernels A, B, and C occupy logical streams S2, S1, and S0 and run between t_0 and t_1. Consequently, performance measurement for an isolated kernel does not directly relate to the overall performance.

This report is structured as follows. In Section 2, we report prior work relevant to this work. In Section 3, we propose the guidelines and describe LVR, the software embodying them. In Section 4, we describe the mathematical model and the numerical scheme used for the present use case, and report the performance results and compare with competing software. In Section 5, we discuss the results and outline future work.

2. Prior work

We report prior work at the intersection between HPC and CFD for compressible flows. In particular, we focus on works implementing numerical schemes similar to ours, on structured grids. To the best of our knowledge, the performance of individual kernels is sometimes reported; however, overall performance is not. Even when kernel performance is indicated, little information is provided about the measurement methodology. As kernels are often executed in an oversubscribed environment, ambiguities may arise regarding the effective elapsed time (see Figure 1(b)), making reported achieved performance not completely reliable.

Radhakrishnan *et al.* (2024), through an OpenACC implementation, port the multiphase compressible flow solver MFC (Bryngelson *et al.* 2021) to NVIDIA GPUs and report 84% strong-scaling efficiency and 97% weak-scaling efficiency. However, in that work, the WENO and Riemann solver kernels sustain about 40% and less than 20% of peak FP64 throughput, respectively. Bernardini and colleagues present STREAmS, a DNS solver for compressible flow that targets heterogeneous architectures, including CPUs, and both NVIDIA and AMD GPUs (Bernardini *et al.* 2021; Sathyanarayana *et al.* 2025). For NVIDIA GPUs, kernels are either written in CUDA Fortran or generated with the cuf directive tool. Porting to AMD GPUs required translation to HIP, achieved with the sutils tool. Performance results include an achieved fraction of peak memory bandwidth between 75% and 79%, an achieved fraction of peak compute for the WENO kernel between 10% and 30%, and weak-scaling efficiency up to 98% on hundreds of nodes. De Vanna and colleagues describe a compressible flow solver that employs OpenACC, enabling extensive testing on both NVIDIA and AMD GPUs, but even with additional optimization strategies, the reported fractions of peak compute throughput for different kernels and GPUs do not exceed 20% (De Vanna *et al.* 2023; De Vanna & Baldan 2024).

Beyond mainstream strategies, Di Renzo *et al.* (2020) introduce the HTR solver for compressible multicomponent flows. It is written in the Regent domain-specific language (Slaughter *et al.* 2015) to target the Legion runtime system (Bauer *et al.* 2012), a task-based alternative to OpenACC for heterogeneous CPU and GPU systems, with GPU support restricted to NVIDIA. Only weak- and strong-scaling efficiencies are reported. Jacobs *et al.* (2017) present OpenSBLI, a framework for code generation in compressible fluid dynamics on heterogeneous architectures. From governing equations, discretization strategies, and boundary and initial conditions, the framework generates a complete solver, leveraging SymPy for symbolic manipulation. Per-kernel performance data are reported in a subsequent study (Mudalige *et al.* 2019), and it can be deduced that the overall achieved performance is about 10% of peak on an NVIDIA V100 and 22% on an Intel Xeon Silver.

3. Guidelines

The premise behind our guidelines is that successful HPC software is discovered through measurements, rather than through design. This requires exploring the abstraction spectrum bottom-up (Figure 1(a)), quantitatively, starting from microbenchmarks that run near hardware limits, and adding application features until performance collapses. We bisect the performance space, understand the cause, address it, and refactor the code to clean up. This stands in contrast to top-down designs, which can proceed without ever seeing throughput close to nominal levels.

While bottom-up design is our approach of choice, it is human engineered and manual. Each layer can, in principle, be improved and tailored to a specific target platform with autotuners. Moreover, one may wish to solve slightly different equations that benefit from the schemes discovered in the bottom-up phase. Given the intensive effort required for the bottom-up pass, we wish to avoid repeating the process. After the bottom-up traversal, we instead go back down again (see again Figure 1(a)) with program synthesis. If we can capture the execution patterns that we have shown to run exceptionally well, we can reuse the resulting framework to target different governing equations where applicable, and deploy rapidly for cases we have not anticipated to cover. Program synthesis framework therefore is our way to expand the application scope of what we have learned vertically, bottom-up.

3.1. *High performance*

Our primary goal is to enable software execution to reach high performance, i.e., find ways to minimize TTS, approximated as

$$\mathrm{TTS} = \frac{\mathrm{IC}}{\mathrm{IPC} \times f}, \tag{3.1}$$

where f is the effective clock frequency, IC is the effective instruction count of the instruction stream implementing the selected numerical schemes , and IPC is the average number of instructions executed per clock cycle. This minimization problem is hard to solve, as it involves conflicting minimization subgoals. The diversified presence of computing units across a system results in multiple frequencies, instruction counts, and IPCs, make the minimization problem ambiguous and even more challenging.

3.2. *Introspection.*

3.2.1. *Inspect assembly*

Within a programming language, source code can be considered a message to the compiler. To verify receipt from the compiler as intended, a disassembler (e.g., objdump -S or cuobjdump -sass) inspects the actual produced machine code of the object file. Thereafter, we are able to inspect assembly listings and search for patterns (e.g., heavy-duty loops, presence of SIMD instructions, absence of control flow, and unit-stride memory access). We accelerate our skills in effectively communicating to the compilers with online learning tools like Godbolt. When switching off some components in a debugging session in release mode, the compiler sometimes removes instruction streams that depend on the disabled code, defying the debugging purpose. Without checking the assembly listing, this issue would remain undetected.

3.2.2. *Synthetic microbenchmarks*

The building blocks of HPC software are synthetic microbenchmarks. These minimalistic kernels target specific hardware features and measure how close we can get to nominal rates. Microbenchmarking is also used to detect skill issues: if we cannot extract the target throughput synthetically, can we then expect to leverage it on the field?

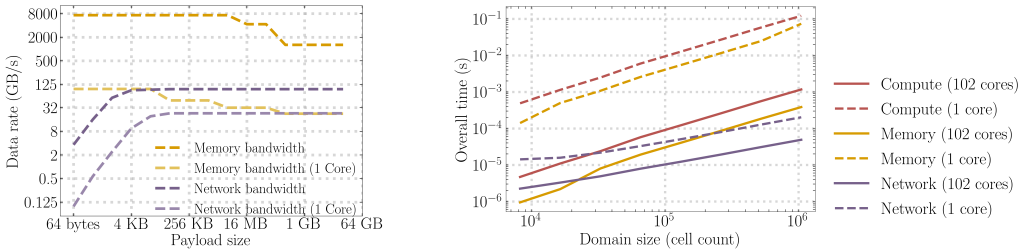
Since they are at a low abstraction layer, microbenchmarks are straightforward to maintain, and are reusable between target projects. Once validated, microbenchmarks may be deployed to characterize new hardware, with STREAM (McCalpin 1995, 2025) and mpiBench (Grove & Coddington 2004)being the most frequently used, in addition to internal ones.

Microbenchmarking can lose predictive accuracy in a proper application, as different hardware features may contend the same resources. We can make an analogy to a Taylor expansion of performance as a 2D function. Linear terms in the two directions represent microbenchmarking of two separate features. The non-linear terms then represent the interaction between the features, such as resource contention, uncaptured by individual microbenchmarks. However, they still remain crucial for providing upper bounds that contextualize observed results.

3.2.3. *Symbolic layer*

Ideally, software design choices would be postponed until microbenchmarking is conclusive enough. But sometimes we want to extend a previously developed software to support new physics, or we have to make decisions on the primary memory layout. The strategy we choose is consequential, as it constrains the freedom of the lower layers of abstraction.

A symbolic framework (such as SymPy) provides an overview across the abstraction spectrum. The lowest layers are represented by data rate curves; the middle layers, by symbolic expressions of what kernels compute; and the top layers, by the composition of such expressions. With this representation, payload sizes and operation counts become trivial queries, in turn enabling us to model the attainable performance of different scenarios. For given problem and simulation configurations, we can predict performance by comparing competing timescales: computation, data transfer, and network communication, etc., and we can configure our problem such that no nominal timescale drastically dominates the others. Our approach differs from the roofline model and its advice (Williams *et al.* 2009). In addition to timescales of system memory bandwidth and



(a) (b)

FIGURE 2. (a) Data rates profiles extracted from synthetic microbenchmarking on the ANL Aurora supercomputer, showing curves absorbed in the symbolic framework; (b) coarse-grained view of the abstraction spectrum, which allows for accurate performance modeling in terms of the timescales involved in the simulation.

computing throughput, we consider network and multiple I/O timescales representing contemporary compute nodes at higher fidelity. Also, roofline-based advice encourages us to have higher operational intensities, and to stop after reaching the ridge point. If operational intensity is above the ridge, we actively decrease it, by either decreasing instruction count or trading computation for more data access, in order to reach overall higher utilization of the system components, pruning TTS further. Besides providing a comprehensive vertical performance model, a symbolic framework is the enabler of code synthesis.

3.3. *Throughput.*

We distinguish among three main forms of parallelism: instruction-level (ILP), data-level (DLP), and thread-level parallelism (TLP) (Patterson & Hennessy 2020). Explicit forms of parallelism, including DLP and TLP, have grown considerably in the past two decades.

3.3.1. *Implicit ILP*

CPU cores discover ILP on the fly; they reorder and dynamically schedule instructions to hide hazard latencies. From the 1980s to the 2000s, architectural advances in ILP roughly doubled the performance rate of RISC microprocessors, yielding about 52% annual improvement versus about 25% from transistor scaling alone Hennessy & Patterson (2017). This enhanced extraction of ILP has led a major convenience to programmers, with the peak point arguably reached by out-of-order execution models Tomasulo (1967), used today across most CPUs from Intel, AMD, as well as some ARM and RISC-V architectures. Combined with architecture-aware compilers, ILP is no longer a programmer's problem: we can simply write code as a linear and logical sequence of expressions.

GPUs, by contrast, execute in order. Although SIMT, the execution model used on GPUs, is the primary latency-hiding mechanism for hazards, ILP can still be a limiting factor. This forces the programmer to rely on explicit source-level techniques to expose ILP (e.g., Volkov (2010, 2016) or software pipelining), which remains tedious even with contemporary macro preprocessors.

3.3.2. *Deceiving DLP*

On CPUs, DLP is implemented and exposed through SIMD instruction sets. Despite recent progress in auto-vectorizers, DLP is where communication with the compiler still

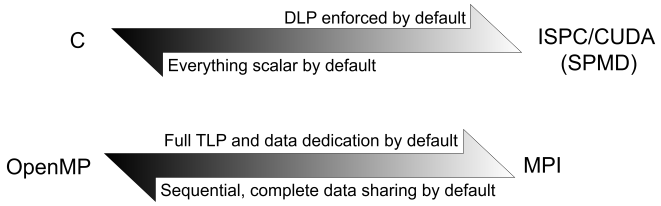


FIGURE 3. Assumption spectrum of programming models for DLP (top) and TLP (bottom).

tends to break down. Aside from choosing a correct memory layout, discussed later, persuading the compiler to emit DLP is a laborious exercise.

C compilers are extremely capable at mapping processing schemes on multicomponent one-dimensional arrays to SIMD instructions, but they struggle to capture multidimensional access patterns. We extend the DLP to multidimensional arrays via a simple compiler deception: we present the data as one-dimensional representations of multichannel arrays, and the same multidimensional algorithm starts leveraging DLP.

Deception techniques are workarounds targeting scalar programming models. Other languages do not require such workarounds because they operate on opposite assumptions, as depicted in Figure 3. Inherited from shading programming/SPMD practice, these include CUDA and HIP on GPU, and ISPC on CPUs/GPUs. DLP-friendly programming models can target both CPUs and GPUs. However, SIMT, is considerably more flexible, as it seamlessly fuses DLP and TLP.

3.3.3. *Troubleshooting TLP*

TLP is a critical performance lever in hardware designs, which often feature tens to thousands of processing elements. It not only scales computational throughput, but also significantly impacts data rates by driving and saturating DRAM and network bandwidth, as illustrated in Figure 2(a). Effective placement, considering hierarchies like sockets, NUMA nodes, and chiplets, is crucial; ignoring this can result in a 20% or more reduction in aggregate data rate. Individual cores are limited by line fill buffers, extracting only a small fraction of DRAM bandwidth. Therefore, multiple cores are employed to sustain memory requests in flight, with specialized cores dedicated to data movement to ensure communication keeps pace with computation. Interprocess communication via shared memory facilitates coordination among these roles. On GPU nodes, core specialization and TLP are particularly vital, as primary processes manage work dispatch to devices (Figure 4).

TLP manifests as threads and processes (both scheduled equivalently on Linux via the clone family; see Figure 3). While OpenMP threads initially appear attractive due to shared address space and simple pragmas, complex data flows introduce significant debugging challenges (e.g., deadlocks, races, and memory inconsistency). Optimal patterns often involve non-interacting workers operating on dedicated data. Furthermore, thread creation/destruction, though rapid, lacks direct hardware counterparts. Job schedulers manage processes with clear policies for placement and oversubscription, whereas threads necessitate their own binding semantics and are prone to oversubscription.

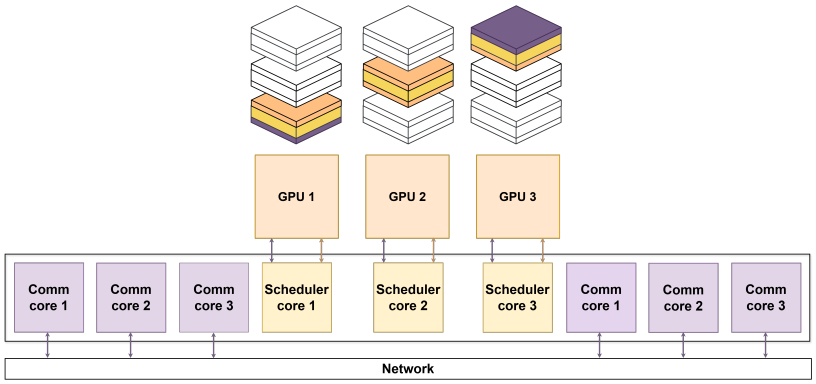


FIGURE 4. Core specialization is essential for maximizing DRAM and network bandwidth at the node level.

3.3.4. *Synchronous programming*

All major components—cores, memory interfaces, buses, and NICs—are clocked. At the timescales that matter on HPC nodes, we can treat the system as synchronous, with deterministic delays. Asynchronous software for HPC has always been a subject of academic interest and research. Unless one uses coroutines or nonstandard programming languages, source code of asynchronous software implies the separation of program execution from state transitions. The instruction stream no longer dictates when and in what order execution state changes, which is error prone. The resulting blooming of dependencies and event handling makes reasoning and debugging much harder. Our practice is to rely on blocking calls and explicit local barriers because they are easy to reason about and debug. We then avoid penalties by building explicit latency-hiding mechanisms with double buffering within well-marked phases, as elaborated later in this brief.

3.3.5. *System-level awareness*

Computing performance is unlocked when software stack and runtime are aligned with the system architecture. Effective subscription levels and accurate TLP placement, which reflect underlying hardware structures (e.g., sockets, NUMA nodes, and chiplets), can significantly amplify the benefits of combined TLP and DLP. We verify runtime binding using lightweight tools such as `xthi` and inspect PCIe fabric and GPU links to ensure correct process-to-GPU affinity. System-level awareness is an interplay among the scheduler, software stack, and problem configuration, rather than solely an internal software property. Source code must be cognizant of external directives and avoid conflicts.

3.3.6. *Data layout and object-disoriented programming*

The memory subsystem is central to HPC due to its deep parallelism and the cross-abstraction impact of data layout decisions. Skepticism regarding object-oriented programming (OOP) in HPC stems from several compounding factors. Fine-grained encapsulation often impedes DLP. While array of structures (AoS) hinders SIMD instruction utilization, structure of arrays (SoA) facilitates it by exposing homogeneous data streams. We employ AoS judiciously, only when it offers clear locality benefits and the computation disallows DLP, frequently preferring composite layouts like AoSoA.

OOP's separation of state management from execution tends to break the sequential nature of how computational flows are expressed with temporal coupling exemplifying

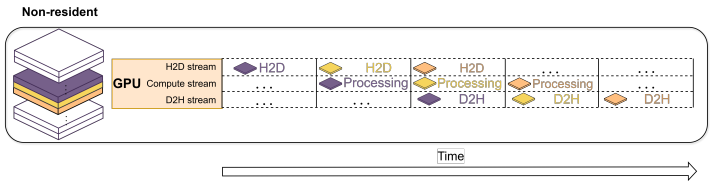


FIGURE 5. Hiding I/O latencies when data do not reside on GPU. Each domain slab undergoes three pipeline stages (upload, compute, and download), each carried out by a task-specific CUDA stream.

this issue, and in turn may disorient compilers. This often leads to instruction streams with excessive data-dependent control flow, increasing instruction count and diminishing instructions per cycle.

Metaprogramming, particularly template recursion, substitutes control flow for code replication, a practice that, among other issues, negates the benefits of loop stream detectors (LSD). In conjunction with header-only libraries, this typically results in a few large translation units and protracted rebuild times. Template diagnostics remain notoriously opaque. Instead of accepting the C++ standard's ok preprocessor, we advocate for true code synthesis that generates clear and concise kernel sources, thereby simplifying execution paths for both human comprehension and compilers.

3.3.7. *Sharing data*

MPI-only software is often memory-consuming. For intranode communication, we favor interprocess communication via shared memory over MPI extensions, given their historical brittleness and slow performance across platforms and versions.

Two primary approaches are considered: POSIX and System V shared memory. On Cray/HPE systems, `xpmem` is preferred, though not universally available. POSIX shared memory, typically backed by `tmpfs` at `/dev/shm`, is usually limited to 50% of RAM. System V shared memory circumvents this cap but imposes per-segment limits, necessitating chunking for multi-GB regions. Both methods involve reserving address space and allocating once at startup, publishing names or keys, and allowing processes to attach. Data sharing among MPI processes not only enhances memory efficiency but also eliminates redundant `MPI_Send` and `MPI_Recv` calls.

The argument that MPI processes with shared memory should be threads overlooks the selective sharing of simulation degrees of freedom, while maintaining private memory allocations that better reflect hardware architecture. This approach largely mitigates false sharing, cache pollution, and memory inconsistencies. Furthermore, processes, unlike threads, are first-class citizens in job schedulers.

3.4. *Latency*

Higher throughput mandates larger workloads, in turn extending latencies. However, a few long latencies are preferred over numerous shorter ones, as the former are more amenable to explicit latency-hiding schemes via double buffering. Despite repeated proposals, such techniques lack widespread adoption. Here, we rediscover two GPU latency-hiding techniques. The first addresses non-resident degrees of freedom, concealing off-chip I/O with three streams per device: one for host-to-device uploads, one for computation,

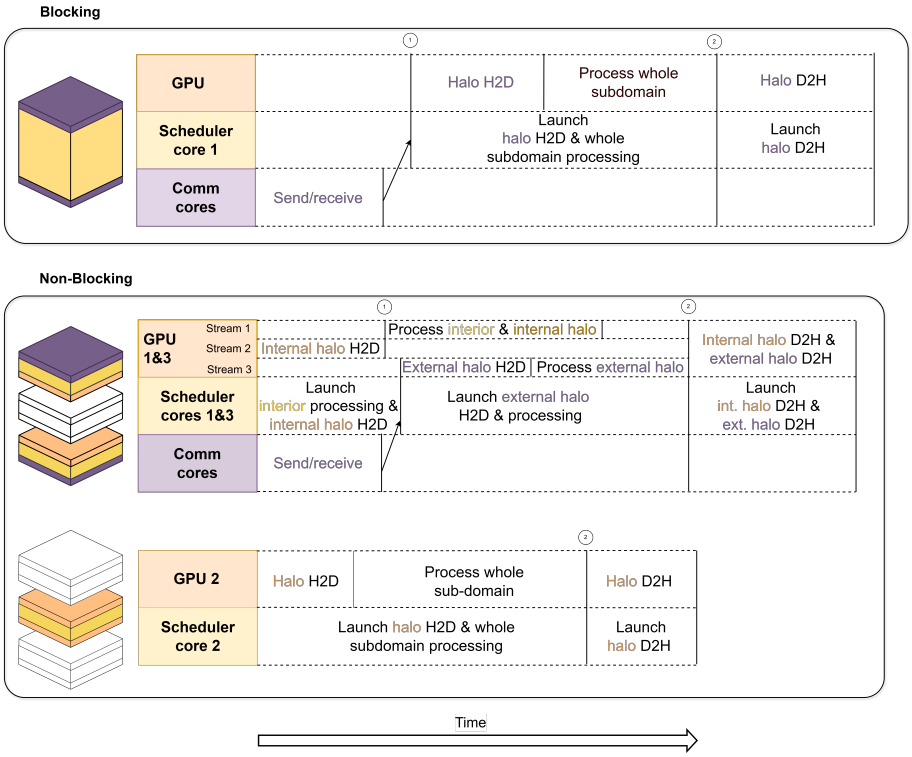


FIGURE 6. Sequence diagram of the computation–communication overlap compared to a non-blocking approach.

and one for device-to-host downloads. Workloads are finely sliced (see Figure 5). The runtime maps transfer streams to DMA engines, with overlap verified on profiler timelines. This method requires page-locked host buffers, is inspired by dated software pipelining techniques, and pairs effectively with MPI plus interprocess communication via shared memory.

The second technique assumes that degrees of freedom are GPU-resident. GPU-aware MPI looks attractive but has been brittle across drivers and MPI stacks and tends to hide complexity. We prefer a TLP-assisted scheme. We split the grid into three launches: two halo slabs and an inner region that touches only on device data. The inner kernel runs while the upload stream moves the next halos and the download stream drains results from the previous step (Figure 6). This approach has outperformed a blocking GPU-aware MPI path in our tests. The achievable fraction of peak network bandwidth by GPU-aware MPI remains unclear. As shown in Figure 2(a), throughput depends on TLP-assisted communication, and TLP subscription level control appears implementation-specific. While unified virtual memory (UVM) initially seems attractive for CPU-to-GPU transfers, its page-fault-driven on-demand migration introduces fine-grained, difficult-to-hide latencies.

Alternatively, for governing equations exhibiting locality, we employ overlapping subdomains, an aggressive technique suppressing communication latency. This involves domain decomposition with overlapping adjacent subdomains, creating thicker halos. Exchanges are deferred during substeps, gradually corrupting boundary data. Before cor-

ruption reaches the interior, halos are exchanged to restore consistency. Thus, latency is traded for redundant computation and enhanced effective bandwidth through larger payloads.

Certain irregular computational schemes encounter the worst-case scenario of numerous short latencies, preventing close-to-peak data rates. The strategy there focuses on improving fine-grained spatiotemporal locality, as it directly impacts the memory subsystem, particularly first-level data caches. This strategy, implemented by data- and computation-reordering techniques including kernel fusion, is beyond the scope of this discussion.

4. Results

4.1. *Compressible flow simulations*

This work focuses on Euler equations and their convective fluxes

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}) = 0 \tag{4.1}$$

$$\frac{\partial (\rho \boldsymbol{u})}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}\boldsymbol{u}^T + p\mathbb{I}) = \boldsymbol{0} \tag{4.2}$$

$$\frac{\partial (\rho E)}{\partial t} + \nabla \cdot ((\rho E + p)\boldsymbol{u}) = 0, \tag{4.3}$$

where $\rho, \boldsymbol{u}, p, E,$ and T are the density, velocity vector, pressure, energy, and absolute temperature of the fluid. The above system of equations is closed by the equation of state $p = \rho R T$, where R is the gas constant. This system can be expressed in flux-divergence form as

$$\frac{\partial \boldsymbol{U}}{\partial t} = \frac{\partial \boldsymbol{F}}{\partial x} + \frac{\partial \boldsymbol{G}}{\partial y} + \frac{\partial \boldsymbol{H}}{\partial z}, \tag{4.4}$$

where $\boldsymbol{U}(\boldsymbol{x}, t) = (\rho, \rho \boldsymbol{u}, \rho E)^T$ is the vector of conservative variables with initial condition $\boldsymbol{q}(\boldsymbol{x}, t) = \boldsymbol{q}_0(\boldsymbol{x}, 0)$, and $\boldsymbol{F}, \boldsymbol{G},$ and $\boldsymbol{H}$ are the convective flux functions. The right-hand side of Eq. (4.4) is discretized employing a structured Cartesian 3D grid,

$$\frac{\partial \boldsymbol{U}}{\partial t} \approx \frac{F^n_{i+1/2,j,k} - F^n_{i-1/2,j,k}}{h_x} + \frac{G^n_{i,j+1/2,k} - G^n_{i,j-1/2,k}}{h_y} + \frac{H^n_{i,j,k+1/2} - H^n_{i,j,k-1/2}}{h_z}, \tag{4.5}$$

where F^n, G^n, and H^n are the numerical approximations of intercell fluxes $\boldsymbol{F}$, $\boldsymbol{G}$, and $\boldsymbol{H}$ at the subscribed locations, and h_x, h_y, and h_z is the uniform grid spacing. Similarly, time is discretized with an (adaptive) time step Δt_n, so that $t = \sum_n \Delta t_n$. Solution of Eq. (4.5) is advanced in time according to a low-storage third-order accurate total variation diminishing Runge–Kutta method. For each time-step, it is necessary to evaluate the right-hand side three times, which is the most computationally expensive aspect of the integration process. Its computation can be split along the three spatial dimensions. Computing numerical fluxes requires solving a Riemann problem at each cell interface, for which we employ an HLLC approximated solver (Toro *et al.* 1994). The HLLC's left and right initial states are obtained from a fifth-order WENO reconstruction. The full algorithm followed is detailed in Collis *et al.* (2025).

The experimental setup consists of three platforms, summarized in Table 1. Using Lawrence Livermore National Laboratory (LLNL) computing platforms, we compare LVR against HTR. The latter has been well validated on both platforms. We focused on overall simulation throughput because it is a reliable metric, and TTS is straightforward

		Dane	Lassen	Aurora
	Facility	LLNL LC	LLNL LC	ALCF
	Sockets	2	2	2
CPU	Vendor	Intel	IBM	Intel
	Model	Xeon 8480+	Power9	CPU Max
	Core count	112	44	104
	Threading	HT	SMT	HT
	FP64 Peak	7.2 TF/s	1.1 TF/s	6.7 TF/s
	DRAM Peak	0.41 TB/s	0.11 TB/s DDR5	0.44 TB/s 1.28 TB/s HBM
GPU	Vendor		NVIDIA	Intel
	Model		V100	GPU Max
	Count		4	6
	FP64 Peak		28 TF/s	178.2 TF/s
	BW Peak		2.9 TB/s	10.8 TB/s
	Notes		1,2,4-way SMT	Exascale class

TABLE 1. Per-node performance characteristics of the computing platforms used in this study.

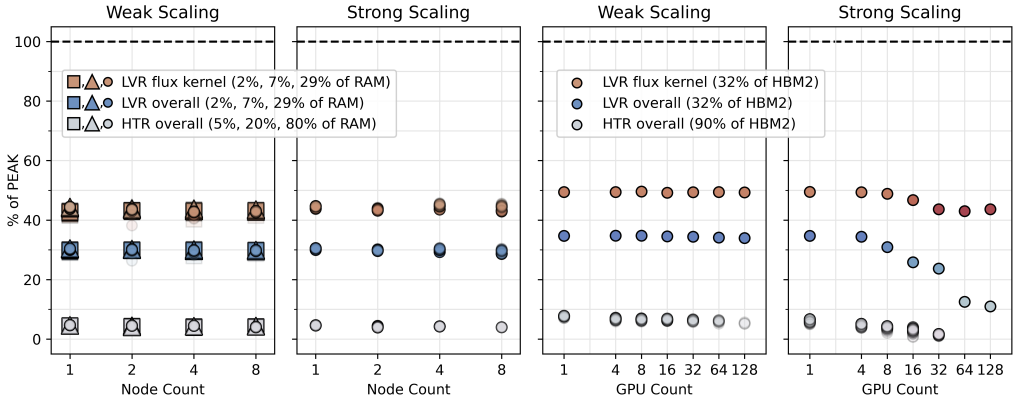
to measure (see Figure 1(b)) when it is not too small. For strong- and weak- scaling studies, we considered a slab domain decomposition on the slowest changing index. Unlike prior work that reports only relative numbers, our symbolic framework is able to precisely count floating point operations, and we rely on Eq. (1.1) to compute the fraction of peak performance.

The presented results were obtained from production runs and hence include real workload issues like resource contention. We also tracked iteration-to-iteration variability: each chart presented in Figure 7 is a scatter plot, with one semitransparent point representing a single iteration. The total number of iterations for all cases is 500, for a TTS spanning multiple minutes, in order to average out transient effects such as frequency scaling. For the problem size used on LLNL Dane (square blue marker, Figure 7), each simulation step involves about 0.5 TFLOP. The FLOP workload can be linearly scaled with the problem size. Performance fluctuations were consistent across runs and platforms, with LVR featuring a per-time step standard deviation of 1 μs and HTR featuring one order of magnitude higher.

4.1.1. *LLNL Dane*

On Dane we tested 1, 2, 4, and 8 nodes (896 cores). Three problem sizes were run for weak-scaling assessment to probe cache effects and communication cost. With HTR, this corresponded to about 5%, 20%, and 80% of the 256 GB DDR5 per node, corresponding to 308^3, 476^3, and 728^3 grid points. With LVR, the footprint for the same grid is about one third of that with HTR. The largest case exceeds three billion grid points.

Both codes show acceptable weak scaling. At 8 nodes, HTR sustains above 90% and LVR above 98%. LVR delivers about 30% of nominal peak performance overall, and the major kernel (flux evaluation) reaches about 43% of peak performance. HTR sustains between 4% and 5% of peak performance. Within this range, the problem size has little impact on throughput, which is consistent with the generous memory per node. Strong scaling follows the same pattern, with slightly lower performance. At 8 nodes, HTR



(a) 2× Intel Xeon Platinum 8480+ **(b) NVIDIA V100 SXM2**

FIGURE 7. HTR and LVR performance on (a) LLNL Dane and (b) LLNL Lassen, with each panel illustrating weak-scaling (left) and strong-scaling (right) efficiencies, reported as fractions of peak performance (PEAK).

retains above 86% efficiency and LVR above 95%. The test relies on the 80% memory consumption case of HTR as the starting point. At 8 nodes, the effective footprint per node is about 10%, which is still higher than the 5% case that already scaled well.

HTR and LVR use different forms of parallelism, with LVR being laid out to match the chiplet-based Xeon Platinum 8480+ design. We place one process per core, grouping 14 processes inside each chiplet, eight chiplets per node, with each group kept within a NUMA domain. All communications use blocking calls. Within a node, LVR shares data through shared memory with explicit process placement, and across nodes it uses MPI. HTR exposes thread-level parallelism with OpenMP inside the node, while the Legion runtime orchestrates and moves data.

4.1.2. *LLNL Lassen*

Scaling benchmarks consider up to 32 nodes (128 GPUs, 320 cores). HTR relies on GPU-resident degrees of freedom. On V100, this considerably limits problem size, as there are just 16 GB of GDDR. We consider a system size of 300^3 cells per GPU, amounting to about 90% of GDDR capacity for HTR and 32% for LVR, as the latter is significantly more memory efficient. CPU cores are busy dispatching work to GPUs and exercise pressure on DRAM/network. Weak-scaling efficiency for LVR is more than 98% on 128 GPUs, due to the latency-hiding scheme proposed herein (Figure 7). HTR drops to about 88% when considering all 4 GPUs within a node because of the blocking transfers involved. Beyond two nodes, efficiency stabilizes, consistent with prior reports. On strong scaling, LVR efficiency remains qualitatively constant from 1 to 4 GPUs, then declines as the case becomes small or internode traffic grows. A marked drop appears from 32 to 64 GPUs in overall throughput, while the flux kernel stays flat, which indicates other kernels become the limiter. HTR degrades quickly under strong scaling.

4.1.3. *ANL Aurora*

We assessed LVR on weak scaling up to hundreds of Aurora nodes, using the same aforementioned metrics, except iteration-to-iteration variability. Per node, grid sizes for CPUs and GPUs were more than 2 billion points and 1.5 billion points, respectively.

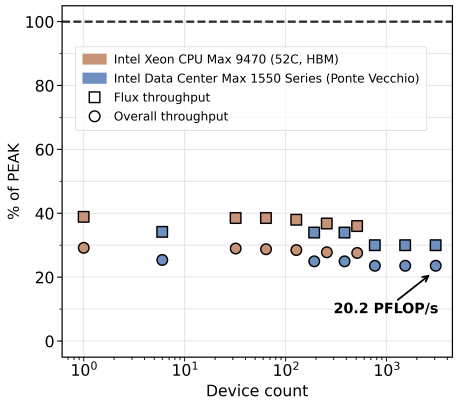


FIGURE 8. LVR weak scaling on Aurora, reported as fraction of peak performance (PEAK).

On 512 nodes, a 20.2 PFLOP/s compute performance was achieved. Results in terms of overall fraction of peak are reported in Figure 8. Weak-scaling efficiency remains above 92% and 87% for CPUs and GPUs, respectively, even when several hundreds of devices are employed. A performance penalty emerges when the node count further increases, with a sharp drop for the GPU version when going from 64 to 128 nodes. In the dragonfly network topology of Aurora, nodes are divided in fully connected groups of 64 nodes, and any two groups are connected by two links. As the node count increases beyond 64, a performance deterioration is therefore expected.

4.2. *Incompressible multiphase flows*

We assess the effect of a small subset of the present guidelines to an incompressible multiphase solver. The solver features a phase field to implicitly represent the interface. The solver relies on staggered grids and a fourth-order accurate time-stepping scheme to discretize and solve for the evolution of the interface (Jain 2022), which is expressed as

$$\frac{\partial \phi}{\partial t} + \nabla \cdot (\mathbf{u}\phi) = \nabla \cdot \left[\Gamma \left[\epsilon \nabla \phi - \frac{1}{4} \left(1 - \tanh^2 \left(\frac{\psi}{2\epsilon} \right) \right) \hat{\mathbf{n}} \right] \right] + \phi (\nabla \cdot \mathbf{u}), \tag{4.6}$$

where ϕ is the volume fraction, $\mathbf{u}$ is the velocity vector, Γ is a stiffness coefficient, ψ is the signed distance function, ϵ is the interfacial thickness, and $\hat{\mathbf{n}}$ is the interface normal. Eq. (4.6) is solved in addition to a Poisson equation for pressure and with the momentum equations

$$\frac{\partial \rho \mathbf{u}}{\partial t} + \nabla \cdot (\rho \mathbf{u} \otimes \mathbf{u}) = -\nabla p + \nabla \cdot (\mathbf{f} \otimes \mathbf{u}) + \nabla \cdot \tau + \sigma \kappa \nabla \phi + \rho \mathbf{g}, \tag{4.7}$$

where ρ is density, P is pressure, $\mathbf{f}$ is the phase-field mass correction term (Jain 2022), $\boldsymbol{\tau}$ is the viscous stress tensor, σ is the surface tension coefficient, and $\mathbf{g}$ is the acceleration due to gravity. More information regarding this solver may be found in Brown & Moin (2024), in which the code is used to perform simulations of droplet sliding.

Within the previously existing software, the memory layout of the flow fields is SoA. This is expected to already help leveraging DLP. The solver algorithm initially was implemented in C++ with minimal OOP limited to field accessors, and expressed multidimen-

		XM kernel		**PF** kernel	
	Per-point arithmetic workload	181 F		72 F	
	Operational intensity	1.74 F/B		1 F/B	
	Problem size (% of DRAM capacity)	3%	42%	3%	42%
BASE	Effective BW	0.97 GB/s	0.95 GB/s	0.52 GB/s	0.53 GB/s
	Fraction of DRAM peak speed	8%	8%	10%	11%
OURS	Effective BW	3.5 GB/s	3.6 GB/s	8.5 GB/s	8.8 GB/s
	Fraction of DRAM peak speed	30%	31%	72%	74%
	Outperforming factor	3.6X	**3.8X**	16.4X	**16.5X**

TABLE 2. Performance metrics for the x-momentum (XM) and phase-field (PF) compute kernels, before (top) and after (bottom) applying a subset of the guidelines. Performance was measured during simulation, with two different system sizes and RAM footprint. Simulations were run on a single core of a two-socket Intel Xeon E5-2670 CPU system. Peak DRAM speed 11.8 GB/s has been measured with single-thread STREAM pinned on the secondary socket.

sional indexing access patterns. We followed a few of the present guidelines: microbenchmarking, using compiler deception techniques for DLP, avoiding OOP, and capturing the computation with a symbolic framework. On identical grids and compiler flags, kernel speedups are 3X to 16X (Table 2), on clearly memory-bounded kernels. As the nominal DLP gain is 4:1, the rewrite successfully leveraged SIMD instructions, but it also largely eliminated unnecessary control flow instructions, as discussed earlier. Use of SIMD instruction is important, even for improving performance of memory-bound kernels: among other benefits it maximizes data traffic per instruction (frontend) as well as traffic per transaction request (Load/Store units, backend).

5. Conclusion and outlook

We proposed and discussed a set of guidelines to ease the development of HPC simulation software. Our solver LVR, built rapidly, sustains high fractions of peak compute performance across multiple architectures, including an exascale system. To our knowledge, no prior work has reported comparable results. Moreover, we exceed HTR, an established compressible flow solver, by up to one order of magnitude.

However, these guidelines are not meant to represent a single unifying road to HPC development. Instead, their purpose is to serve as useful guidance, based on our experience, and without any claim of generality. They have been herein applied in a rather unconstrained use case, characterized by a regular grid, restricted physics, and no complex couplings. Extending beyond this scenario may prove challenging and require further work.

Given the demonstrated performance portability across architectures, we expect these strategies to remain effective on other systems with only minimal modifications. Furthermore, because our methods do not rely on assumptions specific to CFD, they may be

applied to other domains, including Python-driven use cases, like heavy HPC workloads of machine learning, especially training.

For brevity, we covered only the most important guidelines and did not cover performance portability, nor productivity.

This brief is intended to lay the groundwork for follow-up papers that will quantify, via ablation studies, how much each technique contributes to TTS, present the symbolic synthesis layer in detail, and outline how to enhance productivity and portability via plugin mechanisms. Moreover, another future venue of exploration is how the use of agentic AI could assist in the development of complex microbenchmarking to assess multiple features at once, accounting for their interaction. AI tools may also help in making performance modeling decisions, by integrating complex platform-specific knowledge, which may be difficult for a human being to access and interpret.

5.1. *Acknowledgements*

This investigation was funded by the U.S. Department of Energy's National Nuclear Security Administration (NNSA) via the Stanford PSAAP-III Center for prediction of laser-induced ignition of rocket engines (Grant Number DE-NA0003968).

REFERENCES

Bauer, M., Treichler, S., Slaughter, E. & Aiken, A. 2012 Legion: expressing locality and independence with logical regions. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)*. IEEE.

Bernardini, M., Modesti, D., Salvadore, F. & Pirozzoli, S. 2021 STREAmS: a high-fidelity accelerated solver for direct numerical simulation of compressible turbulent flows. *Comput. Phys. Commun.* **263**, 107906.

Brown, L. & Moin, P. 2024 Diffuse interface boundary conditions for dynamic contact angles. *Annual Research Briefs,* Center for Turbulence Research, Stanford University, pp. 281-291 .

Bryngelson, S. H., Schmidmayer, K., Coralic, V., Meng, J. C., Maeda, K. & Colonius, T. 2021 MFC: an open-source high-order multi-component, multi-phase, and multi-scale compressible flow solver. *Comput. Phys. Commun.* **266**, 107396.

Cameron, K., Ge, R. & Feng, X. 2005 High-performance, power-aware distributed computing for scientific applications. *Computer* **38**, 40–47.

Collis, H., Bezgin, D. A., Mirjalili, S. & Mani, A. 2025 A thermodynamically consistent and robust four-equation model for multi-phase multi-component compressible flows using ENO-type schemes including interface regularization. *arXiv Preprint 2504.14063* .

De Vanna, F., Avanzi, F., Cogo, M., Sandrin, S., Bettencourt, M., Picano, F. & Benini, E. 2023 URANOS: a GPU accelerated Navier-Stokes solver for compressible wall-bounded flows. *Comput. Phys. Commun.* **287**, 108717.

De Vanna, F. & Baldan, G. 2024 URANOS-2.0: improved performance, enhanced portability, and model extension towards exascale computing of high-speed engineering flows. *Comput. Phys. Commun.* **303**, 109285.

Decyk, V. K., Karmesin, S. R., de Boer, A. & Liewer, P. C. 1996 Optimization of particle-in-cell codes on reduced instruction set computer processors. *Compu. Phys.* **10**, 290–298.

Di Renzo, M., Fu, L. & Urzay, J. 2020 HTR solver: an open-source exascale-oriented

task-based multi-GPU high-order code for hypersonic aerothermodynamics. *Comput. Phys. Commun.* **255**, 107262.

Fu, Y., Shen, W., Cui, J., Zheng, Y., Yang, G., Liu, Z., Zhang, J., Ji, T., Xie, F., Lv, X., Liu, H., Liu, X., Liu, X., Song, X., Tao, G., Yan, Y., Tucker, P., Miller, S., Luo, S., Koric, S. & Zheng, W. 2023 Toward exascale computation for turbomachinery flows. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23)*. Association for Computing Machinery.

Gropp, W. 1997 Performance driven programming models. In *Massively Parallel Programming Models (MPPM)*.

Grove, D. & Coddington, P. 2004 Communication benchmarking and performance modelling of MPI programs on cluster computers. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*, (pp. 201–217).

Hennessy, J. L. & Patterson, D. A. 2017 *Computer architecture: a quantitative approach*, 6th edn. Morgan Kaufmann.

Jacobs, C. T., Jammy, S. P. & Sandham, N. D. 2017 OpenSBLI: a framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures. *Comput. Sci.* **18**, 12–23.

Jain, S. S. 2022 Accurate conservative phase-field method for simulation of two-phase flows. *J. Comput. Phys.* **469**, 111529.

Kaushik, D., Smith, B., Keyes, D., Barry, D. & Smith, F. 1998 Prospects for CFD on petaflops systems. In Bjørstad, P. & Luskin, M. (Eds.), *Parallel Solution of Partial Differential Equations. The IMA Volumes in Mathematics and its Applications* (vol. 120, pp. 247–277). Springer .

McCalpin, J. D. 1995 Memory bandwidth and machine balance in current high performance computers. *IEEE Comput. Archit. Tech. Comm. Newsl* .

McCalpin, J. D. 2025 STREAM: sustainable memory bandwidth in high performance computers.

Mudalige, G., Reguly, I., Jammy, S., Jacobs, C., Giles, M. & Sandham, N. 2019 Large-scale performance of a DSL-based multi-block structured-mesh application for direct numerical simulation. *J. Parallel Distrib. Comput.* **131**, 130–146.

Patterson, D. A. & Hennessy, J. L. 2020 *Computer organization and design: the hardware software interface: RISCV edition*.

Radhakrishnan, A., Le Berre, H., Wilfong, B., Spratt, J.-S., Rodriguez, M., Colonius, T. & Bryngelson, S. H. 2024 Method for scalable and performant GPU-accelerated simulation of multiphase compressible flow. *Comput. Phys. Commun.* **302**, 109238.

Sathyanarayana, S., Bernardini, M., Modesti, D., Pirozzoli, S. & Salvadore, F. 2025 High-speed turbulent flows towards the exascale: STREAmS-2 porting and performance. *J. Parallel Distrib. Comput.* **196**, 104993.

Slaughter, E., Lee, W., Treichler, S., Bauer, M. & Aiken, A. 2015 Regent: a high-productivity programming language for HPC with logical regions. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*.

Tomasulo, R. M. 1967 An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. Dev.* **11**, 25–33.

Toro, E. F., Spruce, M. & Speares, W. 1994 Restoration of the contact surface in the HLL-Riemann solver. *Shock Waves* **4**, 25–34.

Volkov, V. 2010 Better performance at lower occupancy. [conference presentation]. GPU Technology Conference (GTC), San Jose, CA, September 20–23, 2010.

Volkov, V. 2016 Understanding latency hiding on GPUs. [conference presentation]. GPU Technology Conference (GTC), San Jose, CA, April 4–7, 2016.

Williams, S., Waterman, A. & Patterson, D. 2009 Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* **52**, 65–76.