Scalable Hydrodynamics on multiple Field-Programmable Gate Arrays (FPGAs)

François-Xavier Mordant, Charles Prouveur, Pascal Tremblin, Nicolas Gac

RESEARCH-ARTICLE

Scalable Hydrodynamics on multiple Field-Programmable Gate Arrays (FPGAs)

FRANÇOIS XAVIER MORDANT, University of Versailles Saint-Quentin-en-Yvelines, Versailles, Ile-de-France, France

CHARLES PROUVEUR, University of Versailles Saint-Quentin-en-Yvelines, Versailles, Ile-de-France, France

PASCAL TREMBLIN, University of Versailles Saint-Quentin-en-Yvelines, Versailles, Ile-de-France, France

NICOLAS GAC, Paris-Saclay Normal School, Gif-sur-Yvette, Ile-de-France, France

Open Access Support provided by:

University of Versailles Saint-Quentin-en-Yvelines

Paris-Saclay Normal School

Published: 16 November 2025

Citation in BibTeX format

SC Workshops '25: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis
November 16 - 21, 2025
MO, St Louis, USA

Conference Sponsors:
SIGHPC

Scalable Hydrodynamics on multiple Field-Programmable Gate Arrays (FPGAs)

François-Xavier Mordant
Maison de la Simulation
Université Paris-Saclay, UVSQ, CNRS, CEA
Gif-sur-Yvette, Île-de-France, France
SATIE
Université Paris-Saclay, ENS Paris-Saclay, CNRS
Gif-sur-Yvette, Île-de-France, France
francois-xavier.mordant@cea.fr

Charles Prouveur
Maison de la Simulation
Université Paris-Saclay, UVSQ, CNRS, CEA
Gif-sur-Yvette, Île-de-France, France
charles.prouveur@cea.fr

Pascal Tremblin
Maison de la Simulation
Université Paris-Saclay, UVSQ, CNRS, CEA
Gif-sur-Yvette, Île-de-France, France
pascal.tremblin@cea.fr

Nicolas Gac
SATIE
Université Paris-Saclay, ENS Paris-Saclay, CNRS
Gif-sur-Yvette, Île-de-France, France
nicolas.gac@universite-paris-saclay.fr

Abstract

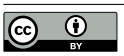
This paper presents scalable 2D and 3D Hydrodynamics solver implementations on FPGAs using Intel's oneAPI framework, addressing challenges in porting stencil-based computations from CPU/GPU architectures on FPGAs. FPGAs offer customizable hardware with on-chip memory that can be custom-tailored for High Performance Computing (HPC) applications. Despite a much lower throughput, FPGAs achieve comparable energy efficiency with domain decomposition compared to NVIDIA A100 GPU. Optimizations leverage FPGA's large cache to minimize data streaming and ensure data reuse, reducing memory bandwidth. Domain decomposition enabled pipeline duplication for scalable single FPGA workload, but also the distribution of the work on two Agilex 7 cards. This work demonstrates FPGA viability for computationally intensive simulations, contributing to scalable algorithms for heterogeneous HPC systems and offering insights for exascale computing.

CCS Concepts

• **Hardware** → **Reconfigurable logic and FPGAs**; *Emerging tools and methodologies*; *Power estimation and optimization*; *High-level and register-transfer level synthesis*; • **Computing methodologies** → **Parallel computing methodologies**; • **Mathematics of computing** → *Partial differential equations*; • **Applied computing** → Physics.

Keywords

FPGA, Hydrodynamics, HPC, Dataflow programming, Parallel Computing, Intel oneAPI, Green Computing



SC Workshops '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1871-7/25/11
https://doi.org/10.1145/3731599.3767545

ACM Reference Format:
François-Xavier Mordant, Charles Prouveur, Pascal Tremblin, and Nicolas Gac. 2025. Scalable Hydrodynamics on multiple Field-Programmable Gate Arrays (FPGAs). In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3731599.3767545

1 Introduction

HydroDynamics (HD) and MagnetoHydroDynamics (MHD) simulations play a central role in modeling physical processes in fields as diverse as astrophysics, nuclear fusion, and plasma physics. These simulations often involve the resolution of hyperbolic systems of partial differential equations using finite-volume methods, yielding stencil-like computation patterns over structured grids. While CPUs and GPUs have long dominated HPC workloads, their efficiency is increasingly challenged by the need for better control over memory access patterns and energy consumption. In contrast, Field-Programmable Gate Arrays (FPGAs) offer reconfigurable hardware with customizable memory hierarchies and dataflow pipelines, making them especially appealing for streaming and memory-bound applications. Their potential to reduce the energy footprint of scientific simulations has attracted growing attention, especially in the context of exascale computing, where power constraints are becoming a limiting factor.

Originally used in embedded systems, FPGAs are now explored for HPC because of their energy efficiency. Previous work, such as [13], has shown that FPGAs can achieve up to twice the energy efficiency of GPUs in Molecular Dynamics (MD) simulations using hardware with comparable transistor size. Their configurability is also an attractive quality as it allows for custom hardware pipelines optimized for specific workloads as studied for 3D tomography in [3]. This flexibility is particularly valuable in stencil-based physics simulations [14], such as computational fluid dynamics [15] [5] and Hydrodynamics [4], where memory access patterns and data locality are critical. Furthermore, in recent years, there has been

a growing interest in utilizing FPGAs for HPC driven by advancements in hardware such as the integration of High Bandwidth Memory (HBM). Early explorations leveraging frameworks such as OpenCL have demonstrated promising results in this context [10]. These results highlight the potential of FPGAs to reduce the energy footprint of HPC applications, making them an increasingly relevant option for optimizing resource-intensive workloads.In parallel with this growing interest, the FPGA programming ecosystem has evolved to include high-level frameworks such as Intel's oneAPI, significantly lowering the barrier for HPC engineers to port existing codes. This enables more accessible exploration of performance gains on modern FPGA architectures, including Intel's Agilex family.

This study investigates the feasibility of porting a C++ HPC code for compressible hydrodynamics to modern FPGA architectures using Intel's oneAPI framework, with the goal of maintaining performance through architectural optimizations. The benchmark solves the Euler equations using a first-order finite-volume scheme on a structured Cartesian grid, representative of stencil-based applications common in physics-based simulations. These computations are inherently memory-bound due to the significant data movement and the computational cost of non-linear flux evaluations at each grid cell. The implementation includes memory streaming for conservative variables, conservative updates, and domain-wide reductions for time step control via CFL conditions-algorithmic features that stress both compute and memory subsystems and are well-suited for evaluating architectural trade-offs. Unlike compute-bound workloads where GPUs excel, hydrodynamics simulations often face memory bandwidth limitations, exacerbated on GPUs by limited cache sizes and register availability. FPGAs, by contrast, enable custom memory hierarchies, pipelined dataflows, and fine-grained control over memory access, making them particularly effective for such memory-bound scenarios. Our implementation targets Agilex 7 FPGAs and proceeds incrementally from 2D to 3D hydrodynamics, paving the way for future extensions to MHD. The use of oneAPI/SYCL facilitates portability while preserving access to FPGA-specific optimizations such as pipeline replication and custom caching strategies, confirming the potential of FPGAs for reducing the energy consumption of large-scale hydrodynamics simulations.

A final practical remark on using FPGAs: the cooling solution cost of an FPGA server would most likely be significantly lower than current GPU-based supercomputers solutions (in our case, our two-FPGAs blade is air-cooled) both in term of energy and water, indeed supercomputers cooling costs are much higher than one would initially think [17] .

2 Methodology

2.1 Targeted Application

To assess the feasibility and performance of FPGAs-based architectures for scientific computing, we employ a benchmark application derived from a hydrodynamics problem. This application solves the Euler equations of compressible fluid dynamics using a first-order finite-volume method on a structured Cartesian grid [2]. We consider a spatial discretization in 2D or 3D with uniform grid spacing Δx, and corresponding indices i, j, and k, along with a time discretization Δt indexed by n. The conservative variables (mass density ρ, momentum $\rho\vec{u}$, and energy ρE in the case of the Euler equations, four fields in 2D, five in 3D) are denoted by $\mathbf{U}^n_{i,j,k}$. The associated numerical fluxes at interfaces in the x-, y-, and z-directions are represented by $\mathbf{F}^n_{i-1/2,j,k}$, $\mathbf{G}^n_{i,j-1/2,k}$, and $\mathbf{H}^n_{i,j,k-1/2}$, respectively. Full details on the computation of these fluxes are provided in [2]. Expressed as a generic finite-volume method in three dimensions,

$$\begin{aligned}\mathbf{U}^{n+1}_{i,j,k} = \mathbf{U}^n_{i,j,k} &- \frac{\Delta t}{\Delta x}\left(\mathbf{F}^n_{i+1/2,j,k} - \mathbf{F}^n_{i-1/2,j,k}\right)\\ &- \frac{\Delta t}{\Delta y}\left(\mathbf{G}^n_{i,j+1/2,k} - \mathbf{G}^n_{i,j-1/2,k}\right)\\ &- \frac{\Delta t}{\Delta z}\left(\mathbf{H}^n_{i,j,k+1/2} - \mathbf{H}^n_{i,j,k-1/2}\right),\end{aligned} \tag{1}$$

the complexity and non-linearity in the numerical scheme, combined with the explicit time integration, makes it a representative workload for many stencil-based HPC applications. The benchmark encapsulates core computational patterns, including memory streaming patterns for $\mathbf{U}^n_{i,j,k}$, non-linear flux evaluations to compute $\mathbf{F}^n_{i-1/2,j,k}$, $\mathbf{G}^n_{i,j-1/2,k}$, $\mathbf{H}^n_{i,j,k-1/2}$, a conservative update for $\mathbf{U}^{n+1}_{i,j,k}$, and a timestep evaluation for Δt with a reduction on the whole domain. The chosen configuration enables us to isolate key bottlenecks in memory throughput, arithmetic intensity, and data reuse, which are critical factors when porting to FPGAs platforms. Our FPGAs implementation targets a spatially two- and three-dimensional version of the solver, allowing an initial exploration of the design-space under constrained resource budgets. All simulations presented in this paper were performed using the standard 2D and 3D blast-wave setup [16]: uniform density ($\rho = 1.0$), ambient pressure ($p = 0.1$) with an over-pressurized central region ($p = 10$ for $r < 0.1$), zero initial velocities, and a perfect-gas equation of state with $\gamma = 5/3$. Periodic boundary conditions were applied, computations were performed in double precision, and the timestep was set according to a Courant-Friedrichs-Lewy (CFL) number of 0.5.

2.2 FPGA Programming with oneAPI

Intel's oneAPI framework provides a unified programming model for heterogeneous computing, enabling development across CPUs, GPUs, and FPGAs using SYCL, a C++-based open standard [9]. Unlike Nvidia's proprietary CUDA, oneAPI's open-standard approach ensures code portability, reducing development overhead and enhancing flexibility [9]. For FPGAs, oneAPI employs High-Level Synthesis (HLS) to convert SYCL code into hardware descriptions, synthesized into bitstreams, abstracting low-level hardware details and lowering the barrier for HPC researchers [9].

Compared to GPU programming, which relies on massive thread parallelism, FPGA development emphasizes deep pipelining and explicit memory management. SYCL's queue, buffer and kernel abstractions provide a portable analogue to CUDA streams, memory copies and kernels-which, when compiled for an FPGA backend, become continuous, deeply pipelined dataflows rather than sets of parallel GPU threads [8]. For example, FPGA kernels process stencil computations in MHD by pipelining grid updates, whereas GPUs distribute threads across grid points. Challenges include longer compilation times due to place-and-route and the need for

hardware-aware optimizations, such as very-precise cache control and managing on-chip memory to avoid DRAM bottlenecks and ensuring full pipelining, as noted in [6]. These issues make FPGA programming more complex but offer greater flexibility and tailored hardware designs.

2.3 Key Performance Indicators

2.3.1 Initiation Interval (II) and Throughput. The **Initiation Interval (II)** measures the number of clock cycles between launching consecutive computations in a pipeline, or equivalently, the number of cycles required to compute two successive elements. An ideal II of 1 implies that a new computation can be initiated every clock cycle, allowing the accelerator to output one updated stencil cell per cycle. In practice, however, factors such as pipeline depth, resource constraints, memory bandwidth limitations, instruction latencies, and irregular memory access patterns can lead to a higher effective II compared to the theoretical minimum.

We quantify the effective II as follows:

$$II_{\text{effective}} = \frac{T_{\text{execution}} \times f_{\text{clock}}}{N_{\text{cells}}} \tag{2}$$

where $T_{\text{execution}}$ is the execution time, f_{clock} is the operating frequency of the FPGA design, and N_{cells} is the number of stencil cells updated during execution. Although this measurement is influenced by both the number of elements processed and the pipeline depth, it serves as an effective indicator of performance. Achieving an II of 1 is ideal, as it indicates that the design is effectively outputting one element per clock cycle, and often serves as a checkpoint in the optimization process. However, achieving this ideal II can require significant development effort, and limitations such as memory bandwidth may prevent it. For example, in MHD simulations where there are 8 variables per cell, current bandwidth constraints (42.7 GB/s) might limit the effective II to a minimum of 1.2, even with optimization, if the frequency is of 400 MHz (8 inputs and 8 outputs in double precision at this frequency require a total bandwidth of 51.2 GB/s). Other factors, such as kernel overhead and pipeline latency (the amount of clock cycles required for one computation to be finished) can reduce that effective II, but often is negligible in HPC context where the amount of computations performed completely outweighs the pipeline latency.

Beyond II = 1, further improvements in throughput can be achieved by deploying multiple pipelines or employing loop unrolling, with the final performance being limited by the available FPGA resources.

Closely related to the II is the **computational throughput**, often expressed in MegaCells updates per second (MC/s) for stencil-based application such as hydrodynamics. Since throughput is directly affected by the II, approaching the ideal value is critical for maximizing the computational rate imposed by the FPGA's clock frequency and the inherent parallelism of the design.

2.3.2 Memory Bandwidth. In the 2D hydrodynamics case, a single pipeline operating at 400 MHz requires a memory bandwidth of 25.6 GB/s, which remains well within the available limits - leaving room for a second pipeline. While duplicating the pipeline would shift the bottleneck to memory bandwidth, the potential performance gain (over 60% efficiency) justifies the trade-off. For 3D hydrodynamics, the required bandwidth increases by approximately 20%, but still remains within the available capacity. However, the significantly higher resource usage in 3D makes pipeline duplication impractical, as it exceeds what the compiler can successfully implement (see details below).

2.3.3 Resource Utilization: Area vs. Precision Trade-offs. FPGA resource utilization is heavily influenced by numerical precision choices. Efficient resource allocation is an aspect for maximizing FPGA performance. Disproportionate reliance on a single resource type - such as extensive logic utilization while leaving DSPs underutilized - can hinder pipeline replication and reduce overall efficiency. A more balanced distribution of resources facilitates better utilization of available hardware and simplifies pipeline duplication.

Moreover, achieving 100% resource utilization is neither practical nor desirable. Near-total saturation of FPGA resources severely complicates placement and routing, often making it impractical to meet performance constraints such as target frequency or II. Additionally, a portion of FPGA resources - up to 14% in our case - is statically reserved for the Intel oneAPI board interface, which manages host-FPGA communication. While this overhead is inherent to the oneAPI ecosystem, similar interface requirements exist in other FPGA software stacks to ensure host-device integration.

Efficient resource utilization and high computational throughput are essential in FPGA-based HPC, particularly for demanding physics simulations such as 3D hydrodynamics. By reducing resource usage, more FPGA area can be allocated either to simulating complex physics or to replicating pipelines, thereby increasing overall performance. One effective strategy is to adjust numerical precision to meet the specific requirements of the application. Lowering precision reduces hardware overhead, improves pipelining (which can decrease latency and increase design frequency), and can potentially boost throughput under favorable conditions. This fine-grained control over precision is a distinct advantage of FPGA architectures compared to traditional HPC platforms.

To achieve this, Intel oneAPI provides the `ac_int` and `ap_float` data types. The `ac_int` type offers arbitrary-precision fixed-point arithmetic, enabling designers to specify integer bit-widths with exact control. In contrast, `ap_float` provides configurable floating-point precision. Although it is limited to a set of predefined exponent-mantissa configurations, these options are carefully selected to ensure compatibility and optimized performance for common floating-point operations (e.g., square root, exponential functions) while offering greater flexibility than standard FP16/FP32/FP64/FP80 formats. This enhanced flexibility allows for better adaptation to the specific needs of an application. In theory, reducing floating-point precision should decrease resource consumption and allow for higher pipeline replication, potentially increasing throughput. While we observed non-negligible resource savings using `ac_int` for index computations, our experiments have revealed challenges in fully exploiting the `ap_float` feature for achieving correct simulations, suggesting that further investigation is needed or that the maturity of this capability on our FPGA is still evolving.

2.3.4 Power Measurement Methodology. Our Intel FPGA card, manufactured by BittWare, provides precise power measurement capabilities with high-frequency updates. We consider two key power

metrics: **Total Board Power** and **Total FPGA Fabric Power**, both measured in watts.

Total Board Power represents the overall power consumption of the FPGA card, encompassing the FPGA fabric, voltage regulators, cooling systems, and auxiliary components. **Total FPGA Fabric Power**, in contrast, accounts only for the power consumed by the FPGA fabric itself, excluding any external components. For consistency and accurate cross-platform analysis, we primarily report **Total Board Power**, as it best reflects the real-world power cost of using an FPGA in an HPC environment.

On the GPU side, the power measurement is based on `nvidia-smi` output which is run every 10 ms after booking an entire 8-A100 node on the Jean Zay supercomputer *gpu_p5* partition. Three runs are done for each simulation to give the average performance and the average power draw.

2.4 Verification and Validation

To ensure correctness, we leveraged oneAPI's ability to target both FPGA and CPU with the same SYCL code, comparing outputs for consistency. We also compute the total mass of the domain (density ρ) at initialization, and compare it at intermediate steps and completion, to verify conservation laws. This check was integrated into the workflow and executed during FPGA-to-host data transfers, ensuring physical accuracy and numerical stability. Additionally, data dumps were triggered at fixed iteration intervals or upon reaching a specified timestep, facilitating seamless import into visualization tools such as ParaView, confirming that results remain identical across platforms and verifying the physical plausibility of the simulated behavior.

2.5 Cache Strategy and Optimization Techniques

Efficient memory access is critical for high-performance FPGA applications. The two primary challenges include:

- **High Latency of External DRAM Access:** Retrieving data from off-chip memory (DDR or HBM) introduces delays, particularly problematic in stencil computations where multiple neighboring elements must be accessed.
- **Limited On-Chip Memory Resources:** While BRAM and MLABs offer high-speed memory, they have limited capacity. Efficient cache management is essential to reduce reliance on external memory.

A key metric affected by memory access is the *Initiation Interval* (II), which denotes the clock cycles required to initiate a new pipeline iteration. Poor memory access patterns increase II, lowering overall throughput. Optimization is therefore necessary to maximize FPGA efficiency.

Stencil computations require accessing multiple neighboring data points per computation step. Fetching each value separately from external memory introduces unnecessary latency. To optimize performance, caching techniques are employed to reduce redundant memory accesses.

In our approach, data are received in column-major order from the FPGA RAM. By the time the FPGA processes a given (i, j) stencil cell (in 2D), the data for the left and top neighbors have already been streamed and stored in a local cache, while the right and bottom neighbors remain in transit (with the bottom value arriving in the next stream and the right value arriving after n streams, where n is the column size, including ghost cells). To avoid the delays associated with high-latency RAM fetches, the stencil computation is delayed until all five required values are locally available thanks to caching effectively 2 columns, we call it the 2n caching strategy.

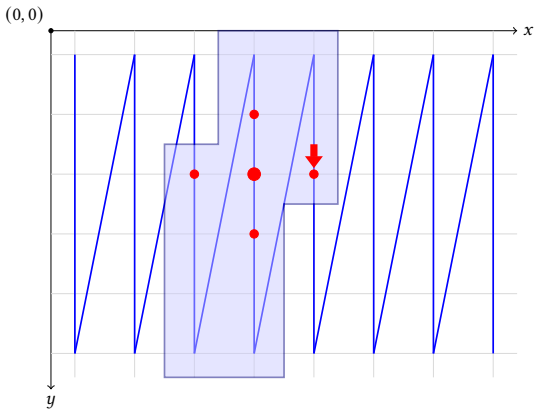


Figure 1: Graph illustrating the 2n-cache strategy in 2D. Blue lines show the data streaming path from FPGA RAM, in a column-major way, red dots indicate the stencil points (with the central, largest dot being the updated cell), the red arrow marks the current cell being streamed, and the light blue box denotes the on-chip cache contents.

Cache Sizing. To support the 2n-cache strategy, it is essential to maintain an on-chip cache that holds the necessary data for two columns plus the required ghost cells. Formally, the minimal cache size required is $2n + 1$ where n corresponds to the grid dimension in the y direction + the ghost cell on each side.

For instance, with a maximum of 32 GB FPGA RAM, streaming the entire domain (with multiple values per cell) limits the domain size to approximately 23000×23000 cells. The minimal cache requirement for such a domain becomes 46005 elements. Due to FPGA BRAM allocation being performed in power-of-two sizes, and because 46005 lies between $2^{15} = 32768$ and $2^{16} = 65536$, each cache is actually allocated $2^{16} = 65536$ elements per variable.

Resource Trade-offs and Pipeline Efficiency. To maintain an initiation interval of 1 - i.e., one computation per clock cycle - the compiler replicates the cache (in our implementation, four times). This replication effectively quadruples the BRAM usage for the cache, consuming roughly 40% of the available BRAM resources. When combined with an additional 11% of BRAM allocated for static partitioning (to manage FPGA communication), the design supports instantiating two pipelines, which pushes overall BRAM usage to about 91%. This replication not only enables concurrent access to the four required stencil neighbors (allowing a new value to be streamed in from the FPGA RAM concurrently) but also ensures that RAM latency is hidden, thereby sustaining a

continuous throughput of one computed stencil value per cycle. Notably, if the number of cache replicates can be reduced while still preserving the II of 1, the freed BRAM resources could be reallocated to support additional pipelines, thereby potentially enhancing overall performance.

3D cache. Extending to 3D added a z-velocity variable, increasing the four variables to five. Caching shifted to 2D plane buffers, requiring $N_x \times N_y \times 5 \times 8$ bytes per plane. With a 2-2D-planes cache implementation, this results in ≈ 5 MB for $N_x = N_y = 256$, within the 35.9 MB SRAM. For larger grids, plane sizes (e.g., $1024 \times 1024 \approx 40$ MB) exceed the on-chip memory capacity available, furthermore hardware limits cache arrays to 65,536 elements per variable.

In summary, the 2n-cache strategy for stencil computations on FPGAs involves streaming domain data while caching only the immediately needed columns/planes. This ensures that all neighbor values for a given stencil are available in fast on-chip memory when required, thereby eliminating the latency penalties associated with FPGA RAM accesses. Furthermore, by keeping all necessary data on-chip at the time of computation, the approach avoids FPGA-to-RAM latency while sustaining high throughput (i.e., maintaining an II of 1). With the cache storage requirement reduced from $O(n^2)$ to $O(2n)$ in 2D, this method not only eases the pressure on BRAM resources for larger problem sizes (RAM limitation ignored) but also offers improved scalability compared to full prefetching.

2.6 Scalability with Domain Decomposition

Using uniform Cartesian grids in both 2D and 3D, our computational domain can be readily partitioned to distribute the workload across multiple computing devices. In our current setup, consisting of a single host node with two FPGAs, we choose to decompose the domain along the X-axis-which is contiguous in memory-to align with efficient access patterns and minimize inter-device communication. We replaced the non-x axis periodic boundary conditions to reflective boundary conditions, which simplifies device-side kernel design and reduces memory access overhead, yielding improved kernel performance with negligible impact on physical fidelity.

We employ MPI to handle inter-domain exchanges via ghost zones, maintaining solver correctness across FPGAs. Decomposing the domain in this manner enables us to test and validate multi-FPGA operation within our hardware constraints. This approach closely mirrors the methodology implemented in the ARK code developed using MPI+Kokkos programming model for compressible hydrodynamics on Cartesian grids with performance portability across CPU/GPU architectures [12].

Although our current experiments are limited to two FPGAs, this domain decomposition strategy is inherently scalable and modular. In principle, there is no architectural barrier to extending the implementation to larger FPGA clusters. Future work will target testing on multi-node platforms based on Intel's Agilex FPGAs - though such large clusters are not yet accessible for experimentation.

2.7 Hardware Platform

The most important component of any FPGA design is of course the chip itself. Our system is based on the `BittWare IA-840f` accelerator card, featuring an `Intel Agilex 7 AGF027` FPGA. Released in Q2 2019 and fabricated on Intel's 10 nm SuperFin process, this FPGA offers `912,800 adaptive logic modules (ALMs)` and `8,528 DSP blocks`, delivering up to `12.8 TFLOPS` in single-precision floating-point performance.

The FPGA integrates `35.9 MB of on-chip SRAM`, split into `32.4 MB (M20K)` block RAM (BRAM) and `3.5 MB (MLAB)`, an ultra-low-latency distributed RAM. It also contains `3,651,200 registers` (about `456KB`), for temporary storage for pipelined computations. However, not all of the FPGA's resources can be utilized. Approximately 10% of the FPGA resources are allocated for the static partition, which includes the board and platform interface logic. This ensures efficient communication with external devices. Additionally, some resources are left unused to facilitate easier routing and fitting of the FPGA design.

The software stack is based on `Intel oneAPI 2024.1`, which fully supports `SYCL 2020` and introduces enhancements for `ap_float` arithmetic. However, our specific combination of `oneAPI 2024.1` and `Accelerator Support Package (ASP) version 2023.1.2` does not yet fully support `ap_float`, limiting its practical use in our setup. In terms of device memory, 2×16 GB DDR4-2666 modules for `32 GB` total are accessible with our current ASP.

Table 1: Key Hardware Specifications

Component	Specification
FPGA Model	Intel Agilex 7 F-Series AGF027
Release Year	Q2 2019
NM technology	Intel's 10 nm SuperFin
Logic Blocks	912,800 ALMs
DSPs / 32b FPUs	8,528 DSPs
Theoretical Peak	12.8 TFLOPS (SP)
On-Chip SRAM	35.9 MB
Registers	up to 3,651,200 (≈ 456 kB)
DDR4 Memory	32 GB (2×16 GB)
Peak Memory Bandwidth	42.7 GB/s
PCIe Bandwidth	32 GB/s (PCIe 4.0 x16)
ASP Version	2023.1.2
Quartus Version	23.1
Software Stack	Intel oneAPI 2024.1

Our hardware platform compares with the state-of-the-art accelerators released in 2019 in several key aspects. Although our FPGA - released in Q2 2019 and fabricated on Intel's 10 nm SuperFin process - delivers impressive compute capability (up to 12.8 TFLOPS in single-precision), its device memory is based on DDR4 technology, limiting the maximum theoretical memory bandwidth to approximately 42.7 GB/s. In contrast, leading accelerators around that time leveraged advanced memory technologies: for instance, Nvidia's A100, built on the 7 nm FinFET process from TSMC in 2020 integrated 40 to 80 GB of HBM2 to achieve bandwidth levels from 1 600 GB/s up to 2 000 GB/s [11]; AMD's Radeon Instinct MI60 provided 32 GB of HBM2 memory with a

bandwidth of up to 1 024 GB/s, delivering competitive double-precision performance at around 300 W [1]. Similarly, Xilinx's Virtex UltraScale+ VU19P, offered configurations with 16 GB of HBM2 (approximately 460 GB/s) or DDR4 with up to 192 GB/s [18].

Additionally, contemporary processors of 2019, such as Intel's Cascade Lake - supporting DDR4-2933 for up 282 GB/s - and AMD's EPYC Rome - operating with DDR4-3200 to reach around 204.8 GB/s - also illustrate how high-end systems were increasingly optimized for high memory bandwidth.

Given the memory-bound nature of many HPC applications, this bandwidth gap is significant. A possible enhancement for our system would be to migrate to an Agilex 7 M-tile variant, which integrates up to 32 GB of HBM2 with about 1 TB/s bandwidth, fixing the bandwidth gap [7].

3 Experimental Results and Analysis

3.1 Throughput and Effective Pipeline Behavior

The sustained throughput of a pipelined kernel is given by

$$T = \frac{f}{\mathrm{II}_{\mathrm{eff}}}, \tag{3}$$

with clock frequency f and effective initiation interval $\mathrm{II}_{\mathrm{eff}}$. An ideal $\mathrm{II}_{\mathrm{eff}} = 1$ means that one cell update is performed per clock cycle, making the throughput directly proportional to f.

In the 2D hydrodynamics solver, Intel oneAPI synthesis reports a nominal II = 1.00. Empirical measurements at the solver level indicate $\mathrm{II}_{\mathrm{eff}} \approx 1.25$, i.e. 432 MC/s sustained versus ∼ 539 MC/s theoretical for the kernel alone. This gap reflects the difficulties in exploiting the memory bandwidth efficiently, rather than structural limitations of the pipeline itself. The problem size used in this experiment is 20000 × 20000 cells, limited by on-chip RAM availability.

In the 3D hydrodynamics solver, the increase in per-cell state variables (from four in 2D to five in 3D) and the larger stencil footprint require more on-chip buffers and interconnect, which in turn increase routing complexity. This leads to a reduced maximum frequency of 425 MHz and an effective $\mathrm{II}_{\mathrm{eff}} \approx 1.16$. As a result, the sustained performance is 365 MC/s for a single pipeline, representing a 15.5% drop in per-cycle throughput compared to the 2D design. That decrease is directly correlated to the bandwidth issue mentioned above. The 3D experiments used a problem size of 173 × 173 × 173 cells, limited by cache capacity rather than raw RAM.

3.2 Resource Footprint and Implications

Table 2 and Figure 2 summarize FPGA resource usage. Throughout this section, `On-chip Mem` refers to BRAM utilization only; as no MLABs were used in our implementation.

In 2D, the design occupies 37% of logic elements, 9% of DSP blocks, and 37% of on-chip memory, accounting for both static partition and kernel resources.

In 3D, resource usage rises sharply to 52% of logic, 13% of DSPs, and 61% of on-chip memory. The larger on-chip memory footprint is directly linked to the increased stencil halo depth and to the need for storing multiple 2D slabs concurrently in cache. This higher utilization constrains the placement and routing phases, contributing to the drop in operating frequency.

While the 2D and 3D designs differ in footprint, they share the same fundamental limitation: pipeline duplication is blocked by off-chip memory bandwidth. In principle, two or even three identical pipelines could be instantiated - up to three in 2D and two in 3D - given available resources. In practice, duplication is infeasible because a single pipeline already sustains ≈27.7 GB/s at a measured II=1.25(instead of the expected II=1 without memory constraints), nearly saturating the DDR4 channel (42.7 GB/s) in 2D; the 3D kernel further increases per-pipeline demand, leaving no headroom for replication. If bandwidth permitted, duplication would yield an almost perfectly linear 2–3× throughput increase, since each pipeline is fully independent and implemented as a deterministic hardware circuit operating at fixed frequency and initiation interval, without runtime scheduling overhead. By contrast, the A100's HBM2 bandwidth (1 500 gbps) is ∼35× higher, enabling massive parallelism without comparable restrictions. Moving to an HBM-enabled FPGA would directly address this bottleneck.

Moreover, available logic and DSPs could implement additional pipelined physics, such as viscosity or gravitational terms, on existing variables, without increasing memory use or affecting pipeline timing, provided that computations remain independent.

Table 2: Resource usage of the 2D and 3D designs

Design Name	2D	3D
Design frequency (MHz)	539	425
Logic (LUTs)	37%	52%
DSPs	9%	13%
On-chip Mem	37%	61%

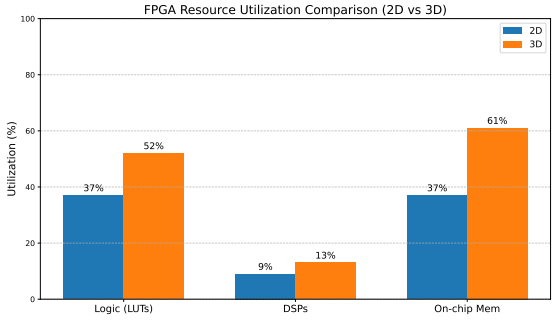


Figure 2: Single FPGA Resource utilization for 2D vs 3D hydrodynamics kernels (LUTs, DSPs, on-chip memory)

3.3 Energy Efficiency (2D and 3D; FPGA vs. GPU)

We benchmark the FPGA designs against the ARK code [12] on an NVIDIA A100, using identical numerical settings (blast wave initial condition, CFL 0.5).

In 2D (see tab. 3), the FPGA achieves 432 MC/s at 68.23 W, giving 6.3 MC/s/W, while the A100 delivers 2 126 MC/s at 263 W, i.e. 8.1 MC/s/W. Although the GPU's raw throughput is ~4.9× higher, the FPGA reaches 78% of its energy efficiency. This is noteworthy given the FPGA's older generation (Agilex 7 with DDR4 memory) and the absence of high-bandwidth memory. As discussed previously, off-chip bandwidth limits pipeline duplication on the FPGA; if extra pipelines could be instantiated, throughput would scale accordingly, with only a moderate increase in power draw, likely improving energy efficiency well beyond the current design.

In 3D (see tab. 4), the FPGA delivers 365 MC/s at 67.91 W (5.4 MC/s/W), compared to 2 728 MC/s at 297 W (9.2 MC/s/W) for the A100. The efficiency gap widens in 3D due to reduced per-pipeline throughput and nearly unchanged FPGA power draw between 2D and 3D. Nevertheless, the FPGA still achieves more than half (59%) of the GPU's energy efficiency in a more demanding stencil regime, without the benefit of HBM.

The results in Figure 3 show that FPGA efficiency advantages are maximized in memory-bound workloads with high cache reuse and simple stencil patterns (e.g. our 2D hydrodynamics case). As stencil dimensionality and data dependencies grow, architectural limits (frequency, on-chip storage, and memory bandwidth) become the primary determinants of performance-efficiency trade-offs.

Table 3: Comparison of Hardware Performance Metrics for 2D Hydrodynamics Simulation

Metric	Agilex 7 FPGA	A100 GPU
Performance (MC/s)	432 MC/s	2 126 MC/s
Mean Power Draw (W)	68.23 W	263 W
Efficiency (MC/s/W)	6.3 MC/s/W	8.1 MC/s/W

Table 4: Comparison of Hardware Performance Metrics for 3D Hydrodynamics Simulation

Metric	Agilex 7 FPGA	A100 GPU
Performance (MC/s)	365 MC/s	2 728 MC/s
Power Draw (W)	67.91 W	297 W
Efficiency (MC/s/W)	5.4 MC/s/W	9.2 MC/s/W

3.4 Multiple FPGAs executions

Multi-device experiments were conducted on two FPGAs using MPI and a host CPU. Each configuration was executed for 2000 iterations. In 2D, the code achieved 844 MC/s on a 40 000 × 20 000 domain, while in 3D it reached 639 MC/s on a 4 000 × 170 × 170 domain, including both kernel execution and MPI communication. Excluding communication overhead, the aggregated kernel performance was 859 MC/s (2D) and 688 MC/s (3D). These results constitute a proof of concept for distributed FPGA execution.

Table 5 summarizes the performance for 1 and 2 devices. Overall strong-scaling efficiency from 1 to 2 FPGAs was 97.86% (2D) and 93.58% (3D). Kernel-only efficiency was higher, at 99.42% (2D) and 99.95% (3D). Figure 4 illustrate these results.

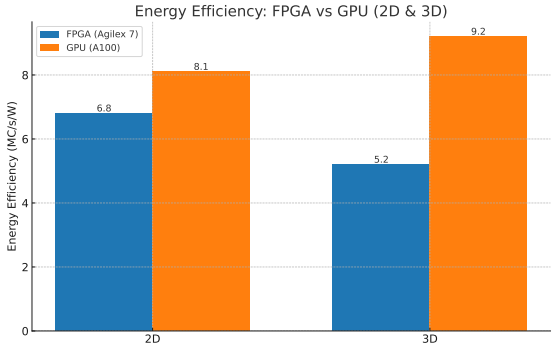


Figure 3: Energy efficiency (MC/s/W) comparison: FPGA (Agilex 7) vs. GPU (A100) in 2D and 3D.

Table 5: Performance results for 2D and 3D hydrodynamics simulation.

Version	FPGAs	Overall (MC/s)	Kernel (MC/s)
2D Hydro	1	431.246	431.998
2D Hydro	2	844.066	858.987
3D Hydro	1	341.374	343.943
3D Hydro	2	638.921	687.517

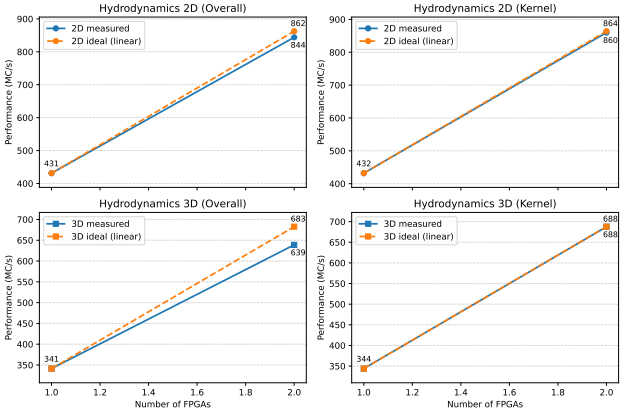


Figure 4: Strong scaling from 1 to 2 FPGAs. Left column: Overall performance (kernel + MPI), right column: Kernel execution only. Top row: 2D, bottom row: 3D. Measured vs ideal performance is shown in all panels.

Communication overhead explains the gap between overall and kernel efficiency. In 2D, efficiency loss is moderate, caused by ghost-cell exchanges across domains. In 3D, the effect is amplified due to smaller subdomain sizes, making communication relatively more expensive. By contrast, kernel performance scales nearly ideally, reflecting the deterministic mapping of computation onto FPGA pipelines. Since kernel execution is independent of inter-device communication, larger deployments should maintain high efficiency, limited only by host-side communication cost.

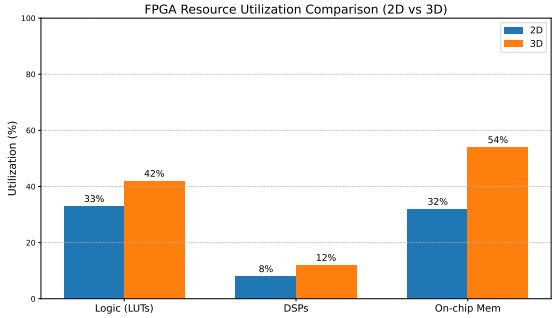


Figure 5: Per-FPGA Resource Utilization of 2D/3D Kernel in Two-FPGA Domain Decomposed Execution

4 Conclusions and perspectives

This work demonstrates the potential and current limitations of using Agilex FPGAs with limited bandwidths for large-scale hydrodynamics simulations. The current state of OneAPI and the Bittware ASP allows one to develop straightforwardly in C++ and achieve good performance, although some features such as mixed precision using `ap_float` are not properly supported yet.

Great performance comparable to A100 level with a single pipeline was achieved in 2D hydrodynamics thanks to leveraging the on-chip memory by caching the data needed by the stencil, with domain decomposition enabling further scaling and improved energy efficiency. Nevertheless, the design was bandwidth-bound with that single pipeline, highlighting that future platforms with High Bandwidth Memory (HBM) could substantially increase performance.

The 3D implementation delivered good single-pipeline performance but fell off compared to A100. The transition from 3D hydrodynamics to 3D MHD is already underway, and initial tests indicate that a full double-precision pipeline is feasible within available resources. A promising starting point for more complex physics.

Overall, these results confirm that FPGAs, even without HBM, can provide competitive performance-per-watt for memory-bound HPC workloads, while offering architectural flexibility to tailor designs to specific applications. Future work will focus on extending the solver to full 3D MHD, exploring HBM-enabled devices and refining mixed-precision strategies once the software stack supports it.

Acknowledgments

The authors gratefully acknowledge the Paderborn Supercomputing Center (PC2) and Dr. Tobias Kenter for support during the FPGA hackathon held in August 2024.

References

[1] AMD. 2019. AMD Instinct MI60 GPU. https://en.wikipedia.org/wiki/AMD_Instinct Accessed: 2025-04-10.

[2] Rémi Bourgeois, Pascal Tremblin, Samuel Kokh, and Thomas Padioleau. 2024. Recasting an operator splitting solver into a standard finite volume flux-based algorithm. The case of a Lagrange-projection-type method for gas dynamics. *J. Comput. Phys.* 496 (2024), 112594. doi:10.1016/j.jcp.2023.112594

[3] Daouda Diakite and Nicolas Gac. 2023. X-ray Tomography Reconstruction Accelerated on FPGA through High-Level Synthesis Tools. *IEEE Transactions on Biomedical Circuits and Systems* 17, 2 (2023), 1–14. doi:10.1109/TBCAS.2023.3258879

[4] Changdao Du and Yoshiki Yamaguchi. 2019. A High-Level Synthesis Design for a Scalable Hydrodynamic Simulation on OpenCL FPGA Platform. In *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies* (Nagasaki, Japan) *(HEART '19)*. Association for Computing Machinery, New York, NY, USA, Article 19, 4 pages. doi:10.1145/3337801.3337807

[5] Jennifer Faj, Tobias Kenter, Sara Faghih-Naini, Christian Plessl, and Vadym Aizinger. 2023. Scalable Multi-FPGA Design of a Discontinuous Galerkin Shallow-Water Model on Unstructured Meshes. In *Proceedings of the Platform for Advanced Scientific Computing Conference (PASC '23)* (Davos, Switzerland). Association for Computing Machinery, New York, NY, USA, Article 8, 12 pages. doi:10.1145/3592979.3593407

[6] IBM Corporation. 2025. FPGA vs. GPU Performance for Deep Learning Applications. https://www.ibm.com/think/topics/fpga-vs-gpu Accessed: 2025-04-10.

[7] Intel. 2023. Intel Agilex 7 M-Series. https://www.intel.fr/content/www/fr/fr/products/details/fpga/agilex/7/m-series.html Accessed: 2025-04-10.

[8] Intel Corporation. 2025. Comparing CPUs, GPUs, and FPGAs for oneAPI Workloads. https://www.intel.com/content/www/us/en/developer/articles/technical/comparing-cpus-gpus-and-fpgas-for-oneapi.html Accessed: 2025-04-10.

[9] Intel Corporation. 2025. oneAPI: Unified Programming Model for CPUs, GPUs, FPGAs, and AI Accelerators. https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html Accessed: 2025-04-10.

[10] Kamalavasan Kamalakkannan, Gihan R. Mudalige, Istvan Z. Reguly, and Suhaib A. Fahmy. 2021. High-Level FPGA Accelerator Design for Structured-Mesh-Based Explicit Numerical Solvers. arXiv:2101.01177 [cs.AR] https://arxiv.org/abs/2101.01177

[11] NVIDIA. 2021. NVIDIA Tesla A100 Data Sheet. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf Accessed: 2025-08-01.

[12] Thomas Padioleau, Pascal Tremblin, Edouard Audit, Pierre Kestener, and Samuel Kokh. 2019. A High-performance and Portable All-Mach Regime Flow Solver Code with Well-balanced Gravity. Application to Compressible Convection. *The Astrophysical Journal* 875, 2 (April 2019), 128. doi:10.3847/1538-4357/ab0f2c Accessed from HAL archive: https://cea.hal.science/cea-04381512v1/document.

[13] Charles Prouveur, Matthieu Haefele, Tobias Kenter, and Nils Voss. 2023. FPGA Acceleration for HPC Supercapacitor Simulations. In *Proceedings of the Platform for Advanced Scientific Computing Conference (PASC '23)*. Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/3592979.3593419

[14] Kentaro Sano, Yoshiaki Hatsuda, and Satoru Yamamoto. 2014. Multi-FPGA Accelerator for Scalable Stencil Computation with Constant Memory Bandwidth. *IEEE Transactions on Parallel and Distributed Systems* 25, 3 (2014), 695–705. doi:10.1109/TPDS.2013.51

[15] Kentaro Sano and Satoru Yamamoto. 2017. FPGA-Based Scalable and Power-Efficient Fluid Simulation using Floating-Point DSP Blocks. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (2017), 2823–2837. doi:10.1109/TPDS.2017.2691770

[16] J. M. Stone, T. A. Gardiner, P. Teuben, J. F. Hawley, and J. B. Simon. 2008. Athena: A new code for astrophysical MHD. *The Astrophysical Journal Supplement Series* 178, 1 (2008), 137–177. doi:10.1086/588755

[17] Jian Sun, Zhiming Gao, David Grant, Kashif Nawaz, Pengtao Wang, Cheng-Min Yang, Philip Boudreaux, Stephen Kowalski, and Shean Huff. 2024. Energy dataset of Frontier supercomputer for waste heat recovery. *Scientific Data* 11, Article 1077 (2024), 7 pages. doi:10.1038/s41597-024-03913-w

[18] Xilinx. 2019. Virtex UltraScale+ VU19P Product Brief. https://www.xilinx.com/publications/product-briefs/virtex-ultrascale-plus-vu19p-product-brief.pdf Accessed: 2025-04-10.

Appendix A: Research Methods

A.1 Reproducibility

All experiments reported in this work were designed to be fully reproducible. Below we detail the hardware, software, compilation, execution, measurement methodology, and artefact availability.

Hardware. Experiments were performed on:

- **FPGA:** BittWare IA-840F, Intel Agilex AGF027 FPGA (AGFB027R25A2E2V), 2 × 16 GB DDR4 (72-bit bus), 35.9 MB on-chip SRAM, 8,528 DSP blocks.
- **Host CPU:** Dual Intel Xeon Platinum 8352Y @ ∼ 2.2 GHz, 32 logical cores, ∼ 512 GB DDR4 RAM.
- **Host OS:** Ubuntu 22.04 LTS (FPGA hardware build additionally tested on Rocky Linux 8, kernel 4.18.0-513.24.1.el8_9.x86_64).
- **GPU:** NVIDIA A100 80 GB HBM2 (Jean Zay supercomputer) for reference implementations.

Software Stack.

- Intel oneAPI Base Toolkit 2024.1.
- Intel oneAPI HPC Toolkit 2024.1.
- Quartus 23.1 patch 0.02iofs (final place-and-route).
- BittWare IA-840F oneAPI Accelerator Support Package (ASP 2024).
- Intel MPI Library 2021.12.
- Optional Python 3 stack (plotting): `numpy, scipy, pandas, matplotlib, seaborn, plotly, kaleido`

Environment initialization for compilation:

```
source /opt/intel/oneapi/setvars.sh
export PATH=/opt/intel/oneapi/compiler/2024.1.0/bin:$PATH
```

FPGA Build and Compilation.

- CPU reference build: `make -Bj cpu` produces `hydro.cpu`.
- FPGA emulator: `make -Bj fpga_emu` produces `hydro.fpga_emu`.
- FPGA hardware build: `make -Bij fpga` produces `hydro.fpga`. Hardware link uses fixed seed (`-Xsseed=2`) and parallelism (`-Xsparallel=8`) for deterministic builds.
- Host-only relink: `make recompile_fpga` if kernel unchanged.
- Optional float precision: `USE_FLOAT=1`.
- Board selection via `Makefile`: `BOARD_NAME := ia840f:ofs_ia840f`.

Execution. All FPGA experiments require exactly 2 MPI ranks (`mpirun -np 2`), supporting a 2×1 subdomain decomposition.

Emulator example in 2D:

```
mpirun -np 2 ./hydro.fpga_emu --sdx 2 --sdy 1
     -t 2 -i 200 -w 10  -x 60 -y 60
```

Hardware example in 2D:

```
mpirun -np 2 ./hydro.fpga --sdx 2 --sdy 1 -t 2
      -i 200 -w 10 -x 60 -y 60
```

Runtime constraints.

- Cache constraint: $2 \times (NB\_Y + 2) + 1 \leq 65536$ (in 2D), $2 \times (NB\_Y + 2) \times (NB\_Z + 2) + 1 \leq 65536$ (in 3D); enforced at startup.
- Only X-direction decomposition implemented; Y (and Z) remains undecomposed.
- Ghost cell exchange: non-blocking MPI between two ranks.
- CFL-limited timestep with deterministic initial conditions for reproducible outputs.

Program Structure. The solver follows the execution flow below (2 MPI ranks, 2 FPGA devices):

```
PROGRAM main
  INIT MPI (nprocs = 1 or 2)
  PARSE input: grid size, time horizon, iteration
      limit
  SETUP discretization: Δx, Δy, Δt
  DEFINE physical constants (C, γ, K)

  // Domain decomposition
  IF nprocs == 1: ASSIGN full domain to single
      rank
  ELSE IF nprocs == 2: SPLIT 2D domain into 2
      subdomains along X
  ELSE: ERROR (unsupported configuration)

  INITIALIZE fields (ρ, u, v, p) with blast
      perturbation
  COMPUTE initial mass
  t ← 0 ; it ← 0

  WHILE (t < T_max AND it < IT_max)
    // Ghost-cell management
    UPDATE ghost cells locally (periodic)
    IF nprocs == 2: EXCHANGE ghost cells between
        neighboring ranks (MPI)

    FOR each local subdomain owned by current rank
        :
      COPY ghost-cell boundary data host→device
      CALL kernel launcher (finite volume step) on
          FPGA
      SWAP old/new device arrays
      COPY back boundary strips device→host
      UPDATE local min Δt
    END FOR

    REDUCE global Δt across ranks
    t ← t + Δt ; it ← it + 1

    IF output interval:
      WRITE fields to file
      CHECK mass conservation
    END IF
  END WHILE

  FINAL mass conservation check
  FINALIZE MPI
END PROGRAM
```

Kernel Structure. The hydrodynamics update is implemented as a single-task SYCL kernel submitted to the FPGA. It performs conservative-primitive conversion, flux computation with a Riemann solver, finite-volume update, and local timestep estimation:

```
KERNEL FVM(U_old, U_new, Δt):
  next_Δt <- ∞
  FOR each interior cell:
    // Convert conservative → primitive variables
    Q_c ← ConvertToPrimitives(U_old[cell])

    // Loop over spatial directions (2D or 3D)
    FOR each direction d:
      Q_L ← neighbor state in -d
      Q_R ← neighbor state in +d
      IF neighbor near boundary: APPLY reflective
          boundary condition
      F_in  ← ComputeFlux(Q_L, Q_c, d)
      F_out ← ComputeFlux(Q_c, Q_R, d)
    END FOR

    // Finite volume update
    UPDATE U_new[cell] with all (F_in - F_out),
        scaled by Δt / Δd

    // Local CFL condition, next Δt computation
    Q_next ← ConvertToPrimitives(U_new[cell])
    next_Δt ← min(next_Δt, CFL(Q_next))
  END FOR
  RETURN next_Δt

FUNC Launcher(U_old, U_new, params):
  SUBMIT FVM as single_task (Intel oneAPI SYCL)
  WAIT for completion
  RETURN (execution_time, next_Δt)
boundary: APPLY reflective
```

Energy Measurement. Energy is measured via BittWare's `bw_card_monitor` using `energy_record.sh`. We report both *Total Input Power* and *Total FPGA Power*, using Total Input Power as the primary metric, reflecting full device cost (static, dynamic, I/O, auxiliary). No warm-up is needed; the 2000 iterations used for reference runs suffice to neglect the first iteration, which underperforms.

```
it=2000; y=20000; x=$((y*2))
./energy_record.sh start "record_${it}_${x}_${y}.record"
mpirun -np 2 ./hydro.fpga -t 2 -i $it -x $x -y $y &> "run_
     ${it}_${x}_${y}.res"
./energy_record.sh stop
python3 plot_power_consumption.py "record_${it}_${x}_${y}.record"
```

Outputs include total input power, FPGA chip power, and mean values.

Output and Verification.

- CSV: `<out>_<iter>_<time>.csv` with `x, y, rho, p, u, v, (w)`.
- Mass conservation verified at each write interval and simulation end.
- Performance metrics printed on rank 0 (timing breakdown, throughput, Mc/s, estimated initiation interval).
- Reference run: `reference_run/` contains sample outputs, logs, synthesis reports, and verification scripts (`compare_outputs.py`).

Artefact Availability. All source files, build scripts, measurement utilities, and reference runs are available at:

https://github.com/fm16191/scalable-fpga-hydrodynamics

Compilation commands, runtime parameters, job submission scripts, and FPGA synthesis reports are included. Reference outputs reproduce best-performing results reported in this work.

Summary. Explicit hardware, software, compilation, execution, measurement, and verification steps enable independent replication of results. Deterministic seeds, cache constraints, and MPI layout ensure identical outputs on identical setups.