uc3m | Universidad Carlos III de Madrid

Master Degree in Aeronautical Engineering
2024-2025

Master Thesis

"GPU accelerated high order Hybrid Discontinuous Galerkin framework for elliptic equations."

Javier Urrutia Madrid

Ceren Gurkan

Madrid, Spain. September 2025

ABSTRACT

This work consists of the GPU-accelerated implementation of a high-order Hybrid Discontinuous Galerkin framework for elliptic equations into CFD Lab's in-house solver TUCANGPU. Currently, TUCANGPU is written in Python and relies on second-order finite difference (FD) schemes for spatial discretization. Aiming to avoid the limitations of the FD discretization and have higher fidelity simulations, this work will center around algorithm development for the 2D Poisson equation using HDG with Dirichlet boundary conditions. The paper outlines the numerical approach in detail and explains the GPU programming techniques used in Python to accelerate the solver. Performance is evaluated by comparing the GPU implementation with equivalent CPU-based versions written in MATLAB and Python. To better understand the efficiency gains, the study includes an analysis of computational cost, memory bandwidth usage, and operational intensity across the different GPU kernels. This helps identify bottlenecks and areas for future optimization.

Key words: CFD; Discontinous Galerkin; High Order; Hybridization; Accelerated GPU computing; Elliptic equations; Poisson equation; Performance analysis.

AKNOWLEDGEMENTS

First of all, I would like to express my most sincere gratitude to my supervisor for this master's thesis, Ceren Gurkan, for being such a dedicated and attentive professor, and for giving me the opportunity to carry out this interesting project. I feel proud of contributing to a long-term project that is novel in the CFD industry, and which I believe will have a meaningful impact on the cost optimization of important problems.

As could not be otherwise, I would like to thank my classmates and friends from the master's degree for always supporting each other through the difficult times and enjoying the good ones together. In particular, I will be eternally grateful for having met Lucia and Daniel.

Lastly, I want to thank my family for their unconditional support, which has allowed me to be writing this paragraph today and to achieve one of my greatest dreams: studying what I love the most.

Contents

List of Figures

List of Tables

Acronyms

AI Artificial Intelligence

API Application Programming Interface

CFD Computational Fluid Dynamics

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

CuPy CUDA-accelerated Python array library

DG Discontinuous Galerkin

DOF Degree of Freedom

DRAM Dynamic Random-Access Memory

FEM Finite Element Method

FDM Finite Difference Method

FVM Finite Volume Method

GDDR Graphics Double Data Rate (memory)

GPGPU General-Purpose computing on Graphics Processing Units

GPU Graphics Processing Unit

HBM High Bandwidth Memory

HDG Hybridizable Discontinuous Galerkin

HPC High-Performance Computing

LES Large Eddy Simulation

NASA National Aeronautics and Space Administration

NumPy Numerical Python

PDE Partial Differential Equation

PID Proportional–Integral–Derivative (controller)

RAM Random-Access Memory

RANS Reynolds-Averaged Navier–Stokes

RHS Right-Hand Side

SIMD Single Instruction, Multiple Data

SIMT Single Instruction, Multiple Threads

SM Streaming Multiprocessor

SpMV Sparse Matrix–Vector multiplication

SRAM Static Random-Access Memory

VRAM Video Random-Access Memory

X-HDG Extended Hybridizable Discontinuous Galerkin

Nomenclature

$\boldsymbol{A}$ Generic local block matrix (e.g., $A_{\ell q}$, $A_{\ell u}$, A_{qq})

$\boldsymbol{A}_{\mathbf{big}}$ Per–element dense local system (condensed unknowns)

$\boldsymbol{A}_{\boldsymbol{\ell\ell}}$ Local face–face block

$\boldsymbol{A}_{\boldsymbol{\ell q}}$ Local face–flux block

$\boldsymbol{A}_{\boldsymbol{\ell u}}$ Local face–solution block

$\boldsymbol{A}_{\boldsymbol{qu}}$ Local flux–solution block

$\boldsymbol{A}_{\boldsymbol{qq}}$ Local flux–flux block

$\boldsymbol{A}_{\boldsymbol{uq}}$ Local solution–flux block

$\boldsymbol{B}$ Generic vector/matrix (e.g., B_q in postprocess)

BW Effective memory bandwidth [GB/s]

CG Conjugate Gradient (iterative solver)

COO Coordinate sparse format (triplets: row, column, value)

CSR Compressed Sparse Row sparse format

cuBLAS NVIDIA CUDA Basic Linear Algebra Subprograms

cuFFT NVIDIA CUDA Fast Fourier Transform

$\mathbf{det}\,\boldsymbol{J}$ Determinant of the element mapping Jacobian

$\boldsymbol{d\Omega}$ Volume quadrature weight [area L^2]

`extFaces` List of boundary faces (e, f)

`faceNodes` Local node indices on each element face (ordered)

$\boldsymbol{F}$ Global face–ID matrix; $F[e, f]$ is the global face index of element e's local face f

$\boldsymbol{f}$ Source term (load), consistent with $-\nabla\cdot(\mu\nabla u) = f$ [problem-dependent]

FLOP Floating–point operation [op]

FP Floating point (number representation)

GB Gigabyte [10^9 bytes]

GFLOP 10^9 floating–point operations [FLOP]

`intFaces` List of interior faces $(e_1, f_1, e_2, f_2, \cdot)$

IPw 2D quadrature weights on the reference triangle [unitless]

$\mathbf{IPw^{1D}}$ 1D quadrature weights on edges [unitless]

$\boldsymbol{J}$ Jacobian matrix of the reference→physical element mapping

$\boldsymbol{K}$ Global (condensed) stiffness matrix for HDG trace unknowns

KB Kilobyte [10^3 bytes]

$\boldsymbol{k}$ Polynomial degree (per element)

$\boldsymbol{\lambda}$ Trace (hybrid) unknown on mesh faces

$\boldsymbol{\mu}$ Diffusion coefficient [e.g., L^2/T]

$\mathbf{n}$ Outward unit normal vector on a face

$\boldsymbol{N}$ Nodal basis function (tabulated at quadrature points)

$\boldsymbol{N^{1D}}$ 1D edge basis functions

$\boldsymbol{\partial_\xi N^{1D}}$ Derivative of 1D edge basis

$\boldsymbol{N_e}$ Number of mesh elements

$\boldsymbol{N_\eta}$ Reference derivative of N in η direction

$\boldsymbol{n_{\mathbf{fn}}}$ Nodes per face

$\boldsymbol{n_{\mathbf{loc}}}$ Local (volume) basis functions per element

`nnz` Number of nonzero entries of a sparse matrix

$\boldsymbol{N_p}$ Number of basis functions per element

Numba Python JIT compiler (CUDA kernels with `@cuda.jit`)

$\boldsymbol{N_\xi}$ Reference derivative of N in ξ direction

OI Operational intensity [FLOP/byte]

$\boldsymbol{Q}$ Local response tensor mapping trace to flux ($q = Q_e\,\lambda + Q_f$)

$\boldsymbol{Q_f}$ Particular (source–driven) flux contribution

$\boldsymbol{QQ}$ Stacked flux response tensor over faces

$\mathbf{q}$ Diffusive flux, $\mathbf{q} = -\mu\nabla u$ [units of $\mu\nabla u$]

RNG Random Number Generator (utility/library)

$\boldsymbol{T}$ Element–to–node connectivity (each row lists the triangle's 3 vertex IDs)

$\boldsymbol{\tau}$ Stabilization parameter on faces [units $\sim \mu/L$]

$\boldsymbol{t_x}$ CUDA thread index within a 1D block

$\boldsymbol{U}$ Local response tensor mapping trace to solution ($u = U_e\,\lambda + U_f$)

$\boldsymbol{U_f}$ Particular (source–driven) solution contribution

$\boldsymbol{UQ}$ Stacked local responses (u, q) to unit face traces

$\boldsymbol{UU}$ Stacked solution response tensor over faces

$\boldsymbol{u}$ Scalar solution field

$\boldsymbol{\hat{u}}$ Numerical trace of u on faces (HDG)

$\boldsymbol{X}$ Nodal coordinates array (mesh geometry)

$\mathbf{x}$ Physical point (x, y) in the domain

$\boldsymbol{\eta}$ Reference coordinate (barycentric axis 2)

$\boldsymbol{\xi}$ Reference coordinate (barycentric axis 1)

$\widehat{\boldsymbol{K}}$ Reference (master) triangle in parametric space

$\widehat{\boldsymbol{K}}^{\star}$ Enriched reference element (postprocess)

$\boldsymbol{X_e}$ Coordinates of the nodes of element e

CHAPTER 1

Introduction

Contents

1.1 Evolution and Current Trends in Computational Fluid Dynamics (CFD)

Computational Fluid Dynamics (CFD) has become an essential tool for both engineering design and scientific research. Its development over the last five decades has always followed the progress of computing hardware, as each generation of faster processors and larger memory has allowed engineers to simulate increasingly complex problems [39].

The first practical CFD methods appeared in the 1970s and 1980s. At that time, engineers mainly relied on potential flow solvers and panel methods, which were useful for low-speed aerodynamics and simple configurations. Later, the introduction of Euler solvers and the first Reynolds-Averaged Navier–Stokes (RANS) simulations represented a major step forward. With the arrival of second-order finite volume (FV) methods, it became possible to analyze complete three-dimensional configurations with reasonable accuracy. These methods became the backbone of industrial CFD, and they are still widely used today for applications where flows remain attached and relatively steady, such as the preliminary design of aircraft wings or car aerodynamics.

However, RANS solvers have reached a practical limit. While they perform well in the *cruise region of the flight envelope*, they cannot reliably predict *turbulent separation, vortex-dominated wakes, or strongly unsteady flows.* Phenomena such as *buffet, dynamic stall, rotorcraft aerodynamics, high-lift configurations, and landing gear noise* are still poorly captured [39]. As the NASA CFD Vision 2030 Study highlights, in these situations wind tunnel testing remains the most reliable source of data, because current CFD methods cannot fully replace experimental measurements in unsteady or separated regimes.

To overcome these limitations, research is moving toward scale-resolving simulations [39], mainly Large Eddy Simulation (LES) and, eventually, Direct Numerical Simulation (DNS). These methods can capture the *unsteady turbulent structures* that RANS cannot resolve, but they demand enormous computational resources. Figure 1.1 illustrates the historical and expected evolution of CFD in terms of computational speed (FLOPs) and memory usage. It shows how the community has progressed from non-linear inviscid solvers to RANS and wall-modeled LES (WMLES), with the ultimate goal of enabling fully transformative, scale-resolving simulations for design purposes. The position marked as "Today" highlights the current gap between the capabilities of industrial RANS and the computational requirements of high-fidelity LES.

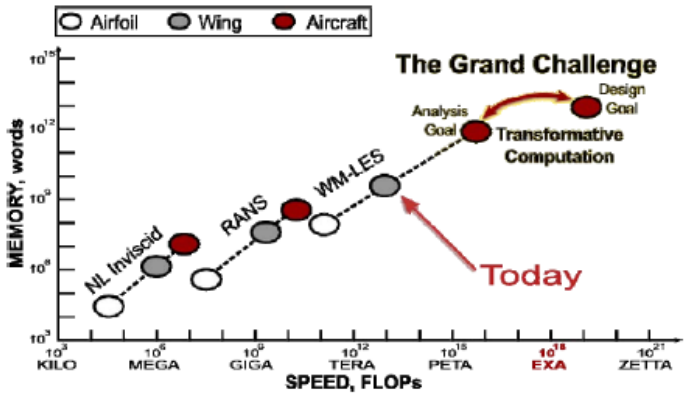


Figure 1.1: *Evolution of CFD methods in terms of computational speed (FLOPs) and memory. Adapted from Witherden & Jameson (2017) [39].*

At the same time, the landscape of high-performance computing (HPC) is undergoing a profound change. Modern supercomputers and even personal workstations increasingly rely on massive parallelism and heterogeneous architectures, which combine multi-core CPUs with GPUs or other accelerators [19]. Figure 1.2 shows an example of an Intel Haswell CPU die, where the integrated GPU occupies a substantial portion of the chip area compared to the CPU cores. This hardware trend forces numerical methods to be GPU-friendly, favoring local operations with high arithmetic intensity and minimal memory transfers, while classical low-order FV methods underutilize such architectures due to memory bandwidth limitations.

These hardware and modeling trends are driving a shift toward high-order methods. Techniques such as Discontinuous Galerkin (DG), Flux Reconstruction (FR), and Spectral Difference (SD) offer clear advantages for next-generation CFD [19]:

- High accuracy per degree of freedom, crucial for resolving *vortex-dominated and turbulent flows.*
- Very low numerical dissipation which allows *long-distance preservation of vortical structures.*
- Local element operations and compact stencils, ideal for GPU acceleration and large-scale HPC.
- Native compatibility with unstructured and curved meshes, necessary for realistic industrial geometries.

Building on these methods, Hybridizable Discontinuous Galerkin (HDG) schemes introduce face-based unknowns that allow the elimination of interior degrees of freedom, reducing the size of

the global system. This feature is highly beneficial for elliptic or mixed PDE systems, such as the pressure Poisson equation in incompressible flow, and it makes HDG particularly suitable for GPU-accelerated frameworks [32] like *TUCANGPU*, which is the focus of this thesis.

In summary, CFD has evolved from low-order, RANS-based methods toward high-order, scale-resolving simulations, driven by the need for accurate unsteady predictions and the emergence of heterogeneous computing hardware. The future of CFD is expected to combine high-order discontinuous and hybridized schemes with massively parallel architectures, bridging the gap between academic high-fidelity methods and industrial simulation needs.

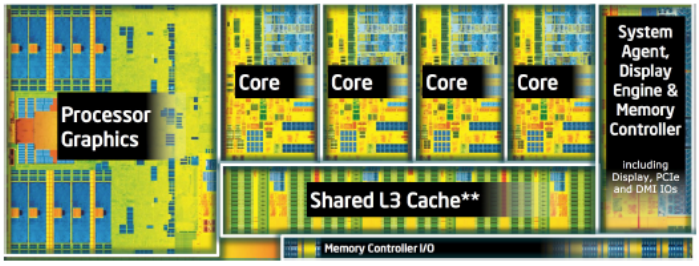


Figure 1.2: *False-color die image of a four-core Intel Haswell CPU. The large area of the integrated GPU highlights the hardware trend toward heterogeneous computing. Image courtesy of Intel Corporation [13].*

1.2 Introduction to Hybridizable Discontinuous Galerkin (HDG) Method

In recent years, the need for accurate and efficient numerical methods has grown dramatically, especially in fields dealing with complex geometries, multiphysics interactions, and interface-dominated phenomena [39]. Traditional low-order methods often struggle to capture fine-scale features without excessive mesh refinement, leading to high computational costs and limited scalability.

This work focuses on the Hybridizable Discontinuous Galerkin (HDG) method, a high-order discretization technique designed to address these challenges in the context of elliptic partial differential equations. HDG belongs to the family of discontinuous Galerkin methods but introduces a hybridization strategy that makes it particularly attractive for large-scale simulations and parallel computing environments.

The proposed **HDG** method combines three key elements:

1. **Hybridization** reduces the size of the global system by condensing element-interior degrees of freedom.
2. **Mixed formulation stability** allows the use of the same polynomial degree k for both the primal variables and their fluxes.
3. **Superconvergence properties**: the solution and its derivatives converge with order $k+1$ in the L^2 norm, and a simple local postprocessing step achieves order $k+2$.

To sum up, HDG offers a robust framework for high-fidelity simulations, especially in scenarios where traditional methods fall short due to mesh constraints or low-order limitations.

1.3 Introduction to GPU Computing

As this work aims to integrate high-order Hybridizable Discontinuous Galerkin (HDG) discretization with GPU acceleration, it is essential to understand the underlying architecture that makes GPUs well-suited for such computationally intensive tasks. The HDG method involves dense numerical operations and high data parallelism—characteristics that align naturally with the strengths of modern GPU systems. Therefore, a general overview of GPU computing is assessed in this section.

Although both the CPU (Central Processing Unit) and GPU (Graphics Processing Unit) are designed to process data, they do so based on fundamentally different architectural principles:

- A CPU typically consists of a few powerful cores optimized for sequential task execution or modest parallelism. This makes it particularly well-suited for general-purpose tasks such as operating system management, logic control, or running standard applications.
- In contrast, a GPU is composed of thousands of smaller, more specialized cores designed to execute many operations concurrently. This highly parallel structure enables GPUs to excel in tasks that involve large-scale, repetitive computations, such as image rendering, scientific simulations, and the training of artificial intelligence models. In essence, while CPUs prioritize versatility and control, GPUs are engineered for throughput and efficiency in massively parallel workloads [36].

Graphics Processing Units (GPUs) are devices that act as **co-processors** to traditional Central Processing Units (CPUs), providing a significant boost in computational capability (see Figure 1.3). Their high performance comes from the large number of computational resources integrated into the chip. On a GPU, most of the silicon area is dedicated to arithmetic operations, while the space devoted to control logic is minimized. This design philosophy allows GPUs to execute a vast number of operations in parallel, making them ideal for tasks with a high degree of data parallelism [36].

According to Flynn's taxonomy, GPUs are close to Single Instruction, Multiple Data (SIMD) systems [12]. In practice, they follow a Single Instruction, Multiple Threads (SIMT) programming model, where thousands of lightweight threads collaborate to solve a problem simultaneously. In this model, the CPU governs the main execution thread and launches GPU kernels, which are parallel routines executed across the GPU cores. This paradigm, known as General-Purpose computing on Graphics Processing Units (GPGPU), has transformed the use of GPUs from purely graphical applications to a broad range of scientific and engineering computations [12].

In recent years, the adoption of GPUs for general-purpose computing has grown rapidly, mainly due to the availability of programming frameworks such as CUDA and OpenCL [14] [9]. These Application Programming Interfaces (APIs) simplify GPU programming, allowing researchers and engineers to accelerate applications that involve dense and sparse algebraic operations, as in the case of the Poisson solver developed in this work.

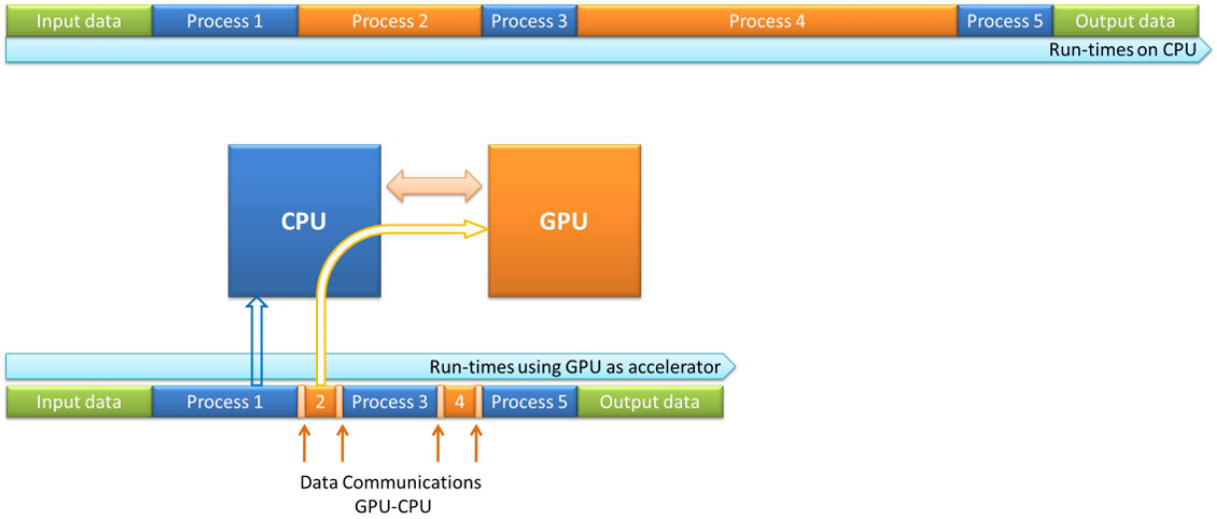


Figure 1.3: *Conceptual view of a GPU acting as a co-processor of a CPU. GPUs devote most of their area to arithmetic units to exploit massive parallelism [18].*

1.4 Literature review

The HDG method was developed by Cockburn [4], who showed how second-order elliptic problems could be reformulated so that globally coupled unknowns live only on the mesh skeleton while element interiors are condensed locally. Building on that foundation, the formulation was further developed beyond Poisson to Stokes, both at the level of practical HDG discretizations [23] and with a rigorous analysis of stability and error estimates [5]. It was then generalized to incompressible Navier–Stokes, initially presented in a conference [24] and subsequently as an implicit high-order solver that exploits static condensation [26]. After this, related developments for compressible Euler/Navier–Stokes further increased the scope [33]. Along the way, the now-standard local post-processing that recovers a superconvergent scalar field (and improves gradient accuracy) became part of the HDG toolkit, with clear recipes and implementation details consolidated in accessible overviews and lectures given to public [25].

On the hardware side, [16] showed that a nodal DG implementation on GPUs can accelerate simulations by a factor of $40-60\times$ compared to a serial CPU code. In the specific context of HDG, Roca demonstrated that a GPU-optimized sparse matrix–vector product (SpMV) for the global trace system can reach about $20-25$ GFLOP/s in double precision, sustaining roughly 40% of peak bandwidth. This performance was shown to be approximately $2\times$ faster than cuSPARSE and around $30\times$ faster than MATLAB on CPU for the same system size [34]. These studies confirm that combining dense local kernels with a reduced global trace system allows GPU-HDG to outperform equivalent DG implementations.

Methodologically, performance comparisons help position HDG against CG and DG for elliptic problems. Matrix-free CG/DG with multigrid often attains excellent time-to-solution by minimizing memory-bound SpMV and maximizing on-chip arithmetic [17]. HDG, in turn, reduces the size of the globally coupled system by hybridizing, shifting work toward dense, element-local kernels and leaving a smaller trace system on faces. Comparative studies ("To CG or to HDG") report regimes, especially at higher orders or when static condensation is efficiently batched, where HDG is competitive or advantageous [15, 40]. Specifically, DG can involve up to $5\times$ more global unknowns

than CG, whereas HDG achieves a clear reduction in the dimension of the global system, placing it much closer to CG than to DG [15, 40]. From a GPU viewpoint, these properties are attractive: localized dense linear algebra maps naturally to batched kernels, and the reduced global DoF count reduces pressure on GPU SpMV. Finally, it is worth highlighting that prior HDG GPU work on the trace operator reinforces this architectural fit [34].

1.5 Motivation, Project Development and Objectives

Modern Computational Fluid Dynamics (CFD) increasingly demands methods that can handle flows with strong unsteadiness and the presence of coherent vortical structures, specially for interface-dominated and multiphysics simulations, as seen before. Many engineering problems fall into this category, including rotating machinery and wind energy applications. Accurately resolving these phenomena often requires solving elliptic problems, such as the pressure Poisson equation in incompressible formulations of the Navier–Stokes equations. In practice, this step frequently becomes the computational bottleneck of the simulation, as it involves the global coupling of the flow field.

The Hybridizable Discontinuous Galerkin (HDG) method offers a promising framework for these problems. By introducing face-based unknowns and condensing interior degrees of freedom, HDG reduces the size of the global system while maintaining high-order accuracy. However, even with this reduction, the global Poisson solve can still dominate the computational cost, especially for large three-dimensional problems.

In this context, GPU acceleration represents a natural step forward. GPUs provide massive parallelism and high arithmetic intensity, which are well suited to the local element operations of HDG. Implementing the Poisson equation on GPUs is therefore a crucial first milestone within the long-term vision of this work, which is to build a complete high-order HDG solver capable of simulating complex aerodynamic configurations with high fidelity, such as wind turbine flows. Accurately capturing tip vortices, wake interactions, and unsteady blade aerodynamics requires a method that combines low numerical dissipation with efficient parallel performance—qualities that HDG can offer once its core components, starting with the Poisson solver, are fully GPU-enabled.

Motivated by the above, the **development of this project** has focused on the implementation and analysis of the Hybridizable Discontinuous Galerkin (HDG) method for the solution of the Poisson equation, both on CPU and GPU. This work represents the first step toward a complete high-order HDG solver suitable for incompressible CFD applications on modern parallel architectures.

The project was carried out in several stages. First, a *reference CPU implementation* was developed in both MATLAB and Python, with the goal of verifying the correctness of the numerical formulation and establishing a baseline for performance. These implementations allowed testing the assembly of local element matrices, face contributions, and the global condensed Poisson system, following the HDG methodology.

Once the CPU version was validated, the *GPU implementation* was developed. The core tasks involved:

- Translating element-local computations to massively parallel GPU kernels.
- Implementing the static condensation process efficiently to reduce the size of the global system.
- Exploring different GPU programming strategies to optimize memory access, arithmetic intensity, and thread parallelization.

Therefore, the **main objective of this thesis** is to develop a GPU-accelerated implementation of the Poisson equation using the Hybridizable Discontinuous Galerkin (HDG) method and to compare its performance against CPU-based implementations. This serves as a first step toward the development of high-order HDG solvers on GPUs, which in the future could tackle complex incompressible flow simulations, including applications such as wind turbine aerodynamics.

This general goal can be divided into the following sub-objectives:

- Implementation and optimization of GPU-accelerated HDG solver for Poisson problem.
- Validation of accuracy and performance through comparison against MATLAB and Python CPU reference implementations.
- Evaluation of how problem size and polynomial degree affect computational efficiency of HDG-GPU architecture.
- Establishment of a scalable computational framework that can later be extended to solve the full Navier–Stokes system with high-order HDG on GPUs.

These objectives are motivated by the need for high-fidelity CFD solvers capable of exploiting the computational power of modern heterogeneous architectures. By addressing this Poisson equation problem first, the present work lays the foundation for future GPU-based HDG solvers that can tackle large-scale, unsteady, and vortex-dominated flows efficiently. Ultimately, this step is crucial for the long-term goal of enabling high-order HDG simulations for applications such as wind turbine aerodynamics and other challenging engineering flows, unifying both academic high-order methods and practical high-performance CFD.

1.6 Structure of this work

The contents of this thesis are organized according to the following structure:

- **Chapter 1 – Introduction:** This first chapter presents an overview of the work, including a brief introduction to high-order Computational Fluid Dynamics (CFD), the Hybridizable Discontinuous Galerkin (HDG) method, and the motivation for developing GPU-based solvers. It also describes the project development process, reviews the relevant literature, and concludes with the justification and objectives of the thesis.
- **Chapter 2 – Theoretical background:** This chapter will introduce the fundamental theoretical concepts required to understand the methodology, focusing on the mathematical foundations of HDG and its role within high-order numerical schemes for incompressible flow simulations.

- **Chapter 3 – HDG discretization for the Poisson problem:** This section will detail the formulation of the Poisson problem, the HDG discretization, and the numerical strategies followed in the development of the solver.
- **Chapter 4 – GPU-HDG implementation:** This chapter will describe the main aspects of the GPU programming approach, including kernel design, memory management, and strategies to exploit parallelism in the Poisson solver.
- **Chapter 5 – Results:** This section will present the validation of the CPU and GPU implementations, performance comparisons, and scalability analyses.
- **Chapter 6 – Conclusions:** The final chapter will summarize the key outcomes of the thesis and propose future directions for extending the solver to full high-order HDG Navier–Stokes simulations.

CHAPTER 2

Theoretical background

Contents

Computational Fluid Dynamics (CFD) is a branch of fluid mechanics focused on the numerical simulation of problems involving fluid flows. Through CFD, it is possible to accurately predict the behavior of a fluid within a given domain, supporting the design and analysis of engineering systems by providing faster and more cost-effective solutions than traditional physical experimentation, thus avoiding the need for multiple prototypes and expensive testing. In both industrial and scientific contexts, CFD simulations have become an essential tool for investigating and optimizing flow phenomena, as they deliver comprehensive information on pressure, velocity, and temperature fields that would otherwise be difficult to obtain. However, achieving these predictions requires the numerical approximation of the governing fluid equations, which poses significant theoretical and computational challenges.

2.1 Mathematical Foundations of Flow: Governing Equations

The starting point of any fluid dynamics simulation lies in the governing equations that mathematically describe the physics of the flow. In particular, a continuous Newtonian fluid is governed by the Navier-Stokes equations, accompanied by the mass conservation equation (continuity) and, when relevant, the energy conservation equation.

These equations represent the fundamental conservation laws of **mass**, **linear momentum**, and **energy** in the fluid. For a viscous fluid (governed by Navier–Stokes) or an ideal inviscid fluid

(governed by the simplified Euler equations), these laws take specific forms but share the same foundation in conservation principles.
In simplified form, for example for an incompressible fluid with constant viscosity, the Navier–Stokes equations can be written as follows.

Continuity Equation (Incompressible Flow)

The incompressible continuity equation enforces mass conservation:

$$\nabla \cdot \mathbf{u} = 0,$$

where $\mathbf{u}(x, t)$ is the velocity field of the fluid.

Momentum Equation (Navier–Stokes)

The linear momentum equation, which is a form of Newton's second law for a fluid, is expressed as:

$$\rho \frac{D\mathbf{u}}{Dt} = -\nabla p + \mu \nabla^2 \mathbf{u} + \mathbf{f},$$

where:

- $\mathbf{u}(x, t)$ is the velocity vector field,
- $p(x, t)$ is the pressure,
- ρ is the density,
- μ is the dynamic viscosity,
- $\mathbf{f}$ represents body forces (e.g., gravity).

The operator

$$\frac{D}{Dt} = \frac{\partial}{\partial t} + (\mathbf{u} \cdot \nabla)$$

is the material derivative, which follows the motion of the fluid.

Energy Equation (Compressible or Thermal Flows)

Additionally, for compressible flows or flows with heat transfer, an energy equation (for example, in terms of temperature or enthalpy) is considered. This equation expresses the first law of thermodynamics applied to the fluid and includes convective, thermal diffusion, and heat source terms.

Nature of the Governing Equations

The full Navier–Stokes system forms a set of nonlinear, coupled partial differential equations (PDEs). The nonlinearity arises primarily from the convective term $u \cdot \nabla u$.

Except for very simplified cases (e.g., simple laminar flows, steady-state regimes with simple geometries), these equations do not admit closed-form analytical solutions. Therefore, to predict the behavior of practical fluid flows, it is essential to rely on numerical methods that approximate the

solution of these equations on a computer.

This is where numerical discretization becomes fundamental, serving as the basis of Computational Fluid Dynamics (CFD) techniques.

2.2 Classification of Partial Differential Equations in CFD

In computational fluid dynamics (CFD), partial differential equations (PDEs) can be categorized into three main types according to their mathematical properties and the way information propagates through the domain: **elliptic**, **parabolic**, and **hyperbolic**. Each type requires specific boundary or initial conditions and reflects different physical phenomena.

Elliptic PDEs

Elliptic equations typically describe time-independent problems. A canonical example is the Poisson or Laplace equation:

$$\nabla^2 \phi = f(\mathbf{x}),$$

which, in Cartesian coordinates, expands to

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} \left(+ \frac{\partial^2 \phi}{\partial z^2} \right) = f(x, y, z).$$

Elliptic equations require boundary conditions specified over the entire domain boundary, and any local change in boundary data influences the solution everywhere in the domain . There are no real characteristic directions or wave propagation; disturbances are felt instantaneously throughout the domain (e.g. a pressure change at one point affects the whole field immediately) [8]. Solutions tend to be smooth if data are smooth. A steady-state heat conduction or potential flow problem is elliptic.

Parabolic PDEs

Parabolic equations involve one time derivative combined with spatial diffusion terms. A classical example is the heat conduction equation:

$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T,$$

where α is the thermal diffusivity. Parabolic PDEs can be interpreted as "elliptic plus time": they require initial conditions in time and boundary conditions in space. Disturbances diffuse as time progresses, affecting only future states; information does not propagate backward in time. Solutions tend to smooth out irregularities, and no sharp wave fronts appear. When a parabolic problem reaches steady state, it reduces to an elliptic one [8].

Hyperbolic PDEs

These often represent wave propagation or advection and usually have second order time derivatives or first-order time coupled with convection (e.g. the wave equation or inviscid Euler equations). They require initial conditions and appropriate inflow/outflow boundary conditions, but not every boundary needs a condition for well-posedness – only as many as the number of characteristics entering the domain . Hyperbolic equations support finite-speed wave propagation along characteristic lines. Information travels along these characteristics, and disturbances propagate with finite

speed (and can cause wave fronts or discontinuities like shocks) . Waves can reflect off boundaries, and changes in one part of the domain affect others only after some time (no instantaneous global influence).

A simple hyperbolic PDE is the pure advection equation:

$$\frac{\partial w}{\partial t} + c\frac{\partial w}{\partial x} = 0,$$

where c is the wave speed. In fluid dynamics, the compressible flow equations (Euler) are hyperbolic systems: they have real eigenvalues and distinct characteristic directions (Mach waves) along which information travels.

Table 2.1 highlights how the *type of PDE* directly dictates the conditions needed for a well-posed problem, the domain configuration, and the expected smoothness of the solution. Elliptic PDEs exhibit global influence of boundary data, parabolic PDEs diffuse disturbances over time, and hyperbolic PDEs propagate information along finite-speed characteristics, allowing discontinuities.

Type	**Problem**	**Prototype**	**BC/IC**	**Domain**	**Solution**
Elliptic	Equilibrium	$\nabla^2 f = 0$	Boundary	Closed	Smooth
Parabolic	Dissipative march	$\frac{\partial f}{\partial x} = \alpha\nabla^2 f$	B + I	Open	Smooth
Hyperbolic	Wave / no dissipation	$\frac{\partial f}{\partial x} = c^2\nabla^2 f$	B + I	Open	May be discontinuous

Table 2.1: *Compact classification of PDEs by type, prototype, conditions, domain, and expected solution behavior.*

The Poisson Equation

The Poisson equation is a fundamental second-order elliptic partial differential equation (PDE) that appears in numerous areas of physics and engineering and plays a central role in Computational Fluid Dynamics (CFD). Its canonical form is expressed as:

$$-\Delta u(\mathbf{x}) = f(\mathbf{x}), \quad \mathbf{x} \in \Omega, \tag{2.1}$$

where $\Delta = \nabla^2$ is the Laplace operator, $u(\mathbf{x})$ is the unknown scalar field (such as pressure, temperature, or potential), and $f(\mathbf{x})$ is a known source term defined in the domain $\Omega \subset \mathbb{R}^n$. The homogeneous case, where $f(\mathbf{x}) = 0$, reduces to Laplace's equation. As seen before, the Poisson equation belongs to the class of elliptic PDEs [8].

Variational (Weak) Formulation of the Poisson Equation: To analyze and solve the Poisson equation on complex domains, it is often advantageous to rewrite it in variational form (also known as the weak formulation).

The idea is to multiply the PDE by a test function and integrate by parts, transferring derivatives from the unknown solution u to the test function v. This leads to a formulation requiring weaker differentiability of u, which is particularly suitable for the Ritz–Galerkin approach, the foundation of the finite element method (FEM) [8].

Boundary Conditions in Elliptic Problems

The Poisson equation is elliptic and therefore requires boundary conditions (BCs) on the entire boundary $\partial\Omega$ to ensure a unique solution. Let Γ_D, Γ_N, and Γ_R denote the portions of the boundary where Dirichlet, Neumann, and Robin conditions are applied, respectively, with

$$\Gamma_D \cup \Gamma_N \cup \Gamma_R = \partial\Omega.$$

Dirichlet Boundary Conditions: Dirichlet conditions fix the value of the solution on the boundary:

$$u(x) = g(x), \quad x \in \Gamma_D.$$

They are essential conditions in the weak form, as trial functions must satisfy $u|_{\Gamma_D} = g$. Physically, Dirichlet BCs prescribe the state of the field, e.g., a fixed temperature in heat transfer or a reference pressure in fluid flow. A full Dirichlet boundary guarantees a unique solution to the Poisson problem.

Neumann Boundary Conditions: Neumann conditions prescribe the normal derivative (flux) of the solution:

$$\frac{\partial u}{\partial n}(x) = h(x), \quad x \in \Gamma_N,$$

where $\partial u/\partial n = \nabla u \cdot \mathbf{n}$.

A purely Neumann problem determines u only up to an additive constant and requires the compatibility condition

$$\int_\Omega f \, dx = \int_{\Gamma_N} h \, ds$$

to ensure solvability.

Robin (Mixed) Boundary Conditions: Robin conditions combine Dirichlet and Neumann information:

$$\alpha(x)\, u(x) + \frac{\partial u}{\partial n}(x) = q(x), \quad x \in \Gamma_R.$$

They model realistic exchanges, such as convective heat loss (*Newton's cooling law*) or surface reactions in diffusion problems. In the weak form, they contribute both to the bilinear and linear terms and ensure uniqueness under mild conditions.

The choice of boundary conditions is dictated by the physics of the problem and directly affects the well-posedness and numerical treatment of the Poisson equation [8].

2.3 Overview of alternative spatial discretizations

As discussed, the starting point is to perform a spatial discretization of the physical domain. Naturally, if the governing equations exhibit time-dependent behavior, a temporal discretization will also be required. For the moment, we will consider a steady-state regime, focusing on the characteristics of the spatial variation of the equations.

There are several methodologies for performing this type of discretization. The most common are the Finite Difference Method (FDM), the Finite Element Method (FEM), and the Finite Volume Method (FVM). Below, the finite difference method is introduced, in order to maintain simple details and clearly illustrate the fundamental concepts.

2.3.1 Finite Difference Method (FDM)

The finite difference method approximates derivatives in the governing equations using their truncated Taylor series expansions. To illustrate the concept, consider the following one-dimensional partial differential equation:

$$\frac{d\phi}{dx} + \phi = 0, \quad 0 \leq x \leq 1, \quad \phi(0) = 1 \tag{2.2}$$

If we discretize the equation over a one-dimensional domain, the grid is composed of five equally spaced nodes, with Δx representing the distance between consecutive nodes. Since the equation is valid at every point in the domain, it can be expressed as:

$$\left(\frac{d\phi}{dx}\right)_i + \phi_i = 0 \tag{2.3}$$

where the subscript i represents the value at node x_i. To obtain an expression for the derivative in terms of the values of ϕ at the grid points, we expand ϕ using a Taylor series as follows:

$$\phi_{i-1} = \phi_i - \Delta x \left(\frac{d\phi}{dx}\right)_i + \frac{\Delta x^2}{2!}\left(\frac{d^2\phi}{dx^2}\right)_i - \frac{\Delta x^3}{3!}\left(\frac{d^3\phi}{dx^3}\right)_i + \dots \tag{2.4}$$

In this case, a backward difference scheme has been employed. Alternatively, one could also use a forward difference or a centered difference scheme for this example.

By neglecting the higher-order terms of the Taylor series and rearranging, the derivative can be approximated as:

$$\left(\frac{d\phi}{dx}\right)_i = \frac{\phi_i - \phi_{i-1}}{\Delta x} + \mathcal{O}(\Delta x) \tag{2.5}$$

The error introduced by neglecting the higher-order terms of the Taylor series is known as the truncation error. In this case, since the truncation error is of order $\mathcal{O}(\Delta x)$, the discrete representation is said to have first-order accuracy. Finally, substituting Equation 2.5 into the discrete form (Equation 2.3), we obtain a fully discrete algebraic equation:

$$\frac{\phi_i - \phi_{i-1}}{\Delta x} + \phi_i = 0 \tag{2.6}$$

In this way, we transition from a differential equation defined over the entire continuous domain to an algebraic equation defined at each discrete node. Solving the approximate solution to the original differential equation now requires resolving a system of algebraic equations, with one equation per node in the mesh.

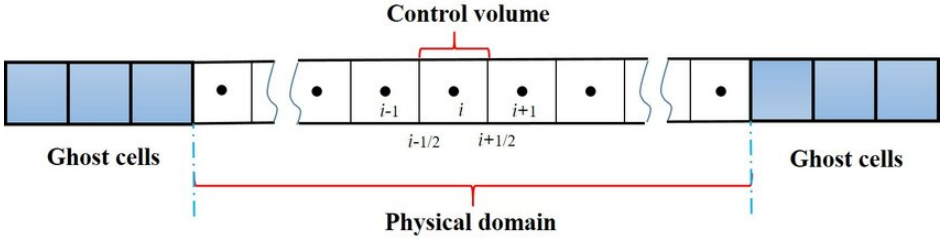


Figure 2.1: *Schematic of a one-dimensional discretization with 5 equally spaced nodes [11].*

FDM is conceptually simple and allows for high accuracy with compact stencils. However, it is generally restricted to structured grids, as its standard formulation assumes an ordered Cartesian arrangement of nodes. Applications to highly irregular or curved geometries are cumbersome and often require complex coordinate transformations.

2.3.2 Finite Element Method (FEM)

The finite element method is based on a functional representation of the numerical solution. Instead of obtaining the solution as a discrete set of points, the unknown field $\phi(x)$ is expressed as a combination of previously defined basis functions $\psi_j(x)$:

$$\phi(x) \approx \sum_{j=1}^{N} \phi_j \, \psi_j(x) \tag{2.7}$$

In this approach, the solution is not exact; it generally produces a residual, which is minimized using specific criteria (such as the Galerkin method) to characterize the numerical approximation. In practice, this method applies a functional representation similar to the concept of the finite difference method. The parameters of the representation correspond to discrete points that divide the domain into a set of elements.

The basis functions are usually defined as polynomial interpolations restricted to contiguous elements (commonly triangles or quadrilaterals in two dimensions). These interpolations can be linear, quadratic, or of higher order. For fluid mechanics applications, it is common to use at least quadratic interpolation functions to accurately approximate second derivatives. As in the finite difference method, the representation parameters are nodal values associated with the mesh nodes.

In this case, the points are located on curves that subdivide the domain into non-overlapping regions. Each node is also associated with a polynomial interpolation function over the surrounding elements. In this way, a compact algebraic system is obtained, where each equation relates only a few nodal values.

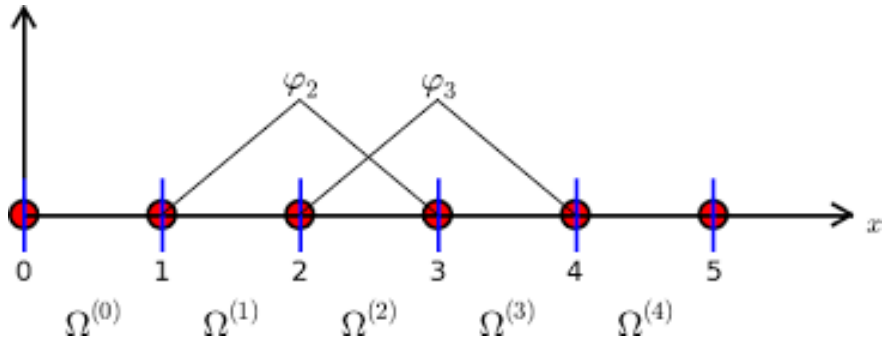


Figure 2.2: *One-dimensional domain with basis functions defined [30].*

FEM naturally handles unstructured meshes and complex geometries, and it provides a solid mathematical framework for error analysis and convergence. Its main limitations in CFD are the higher computational cost and the need for stabilization techniques to handle convection-dominated flows or ensure local conservation. Despite these drawbacks, FEM is the basis for modern high-order and hybridized methods such as DG and HDG.

2.3.3 Finite Volume Method (FVM)

The finite volume method (also known as the *control volume method*) divides the domain into a finite number of non-overlapping control volumes. The conservation of the variable ϕ is imposed in a discrete manner over each control volume. Starting from the one-dimensional differential equation introduced in Section 2.3 and integrating over a control volume P (see Figure 2.3), we obtain:

$$\int_w^e \frac{d\phi}{dx}\, dx + \int_w^e \phi\, dx = 0 \tag{2.8}$$

Assuming a linear variation of ϕ between the cell centers (where the variable is stored) and taking the average value of ϕ in cell P as $\bar{\phi}_P$, the discrete equation for the 1D finite volume method can be written as:

$$\frac{\phi_E - \phi_P}{\Delta x} - \frac{\phi_P - \phi_W}{\Delta x} + \bar{\phi}_P\, \Delta x = 0 \tag{2.9}$$

Here, P denotes the current node (or control volume), while E and W denote the nodes to the right (east) and left (west), respectively. The previous approximation assumes that the variable ϕ varies linearly between the mesh nodes. Rearranging the expression, the discrete algebraic form is:

$$a_P \phi_P = a_E \phi_E + a_W \phi_W + b \tag{2.10}$$

where a_P, a_E, and a_W are the coefficients associated with the variables in each node, and b represents a source term. Equations analogous to (2.10) can be obtained for every control volume in the domain, resulting in a coupled algebraic system that must be solved for all unknowns simultaneously.

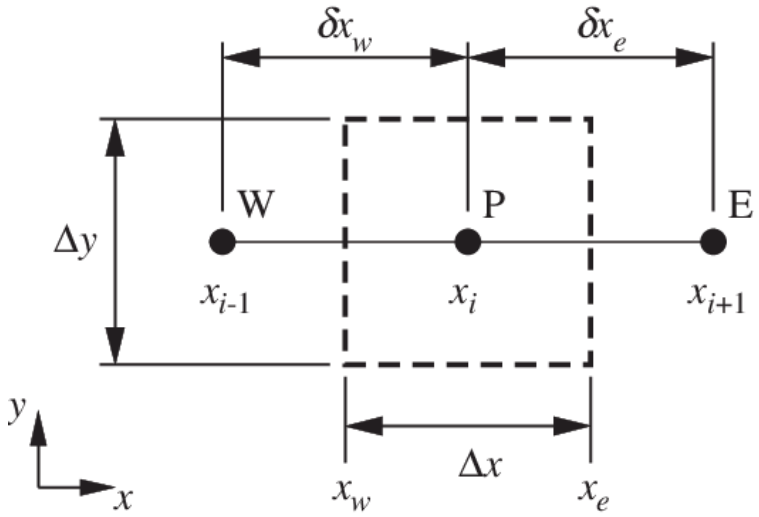


Figure 2.3: *Schematic of a 1D finite volume discretization [30].*

Its key advantage is geometric flexibility, as it can be applied naturally to unstructured meshes of arbitrary polyhedral shape, which makes it the most widely used method in industrial CFD codes. However, while FVM guarantees local conservation, its accuracy depends strongly on the interpolation of face fluxes, and coarse meshes may produce conservative but low-accuracy solutions.

2.3.4 HDG Formulation for the Poisson Equation

In this subsection, we assess the HDG formulation for the Poisson equation and briefly contrast CG and DG to motivate HDG. At the end, we summarize the properties that make HDG especially attractive compared to DG.

The **continuous Galerkin (CG)** method, also referred to as the standard finite element method (FEM), is a widely used approach for the numerical solution of partial differential equations, particularly elliptic problems such as the Poisson equation. It is based on a variational (weak) formulation in which both the trial and test functions are drawn from a continuous finite-dimensional space, typically composed of piecewise polynomials that are globally continuous across element boundaries. This continuity ensures a symmetric and well-posed linear system and leads to accurate solutions with relatively low computational cost. However, CG methods do not guarantee local conservation of fluxes and may struggle with problems involving strong advection or discontinuities. These limitations have motivated the development of discontinuous Galerkin (DG) methods, which relax the inter-element continuity constraint to provide improved flexibility, better handling of complex physics, and enhanced suitability for parallel computation.

Discontinuous Galerkin (DG) methods are a class of finite element methods in which the solution and test functions are allowed to be discontinuous across element interfaces. Unlike traditional continuous Galerkin methods, where continuity is enforced strongly (i.e., solutions are continuous across elements), DG formulations enforce inter-element conditions weakly via interface integrals and numerical fluxes. This approach offers several advantages: high-order accuracy, local conservation, geometric flexibility, and ease of parallelization [3]. However, one of the main drawbacks is the large number of globally coupled degrees of freedom due to the duplication of unknowns across interfaces.

The **Hybridizable Discontinuous Galerkin (HDG)** method is a discontinuous finite element approach developed to reduce the number of globally coupled unknowns while maintaining the benefits of classical DG methods, such as high-order accuracy and local conservation [4, 23].

The key idea is to introduce additional unknowns on the element interfaces, often called *trace* or *hybrid* variables. These act as Lagrange multipliers that weakly enforce continuity between elements. By doing so, the interior degrees of freedom can be eliminated locally, leaving a much smaller global system involving only the interface variables.

This process, known as *static condensation* or *hybridization*, allows each element problem to be solved independently using the trace variables as boundary data. The global system then only couples the interface unknowns, which significantly reduces computational cost without sacrificing accuracy or conservation.

2.3.4.1 Discontinuous Galerkin Formulation

To appreciate the advantages of HDG, we first recall the formulation of the classical Discontinuous Galerkin (DG) method applied to the Poisson equation.

Given the strong form:

$$-\nabla \cdot (\kappa \nabla u) = f \quad \text{in } \Omega, \qquad u = 0 \quad \text{on } \partial\Omega,$$

the DG method seeks an approximate solution u_h in a discontinuous finite element space. The weak formulation over a mesh of elements $\mathcal{T}_h$ becomes:

$$\sum_{K \in \mathcal{T}_h} \Big[\int_K \kappa \nabla u_h \cdot \nabla v_h \, dx - \int_{\partial K} \kappa (\nabla u_h \cdot n) v_h \, ds + \int_{\partial K} \kappa \widehat{\nabla u_h \cdot n} \, v_h \, ds \Big] = \sum_{K \in \mathcal{T}_h} \int_K f v_h \, dx, \quad (2.11)$$

for all test functions v_h in the same discontinuous space. The term $\kappa \widehat{\nabla u_h \cdot n}$ is the numerical flux, introduced to weakly enforce inter-element coupling. A typical choice is:

$$\kappa \widehat{\nabla u_h \cdot n} = \{\!\{ \kappa \nabla u_h \cdot n \}\!\} + \sigma u_h,$$

where:

- $\{\!\{\cdot\}\!\}$ denotes an average across faces,
- u_h is the jump of u_h,
- $\sigma > 0$ is a stabilization parameter.

Each element shares interface integrals with its neighbors. Since u_h is discontinuous, there are duplicated degrees of freedom on every interface, and all element unknowns are globally coupled. This leads to a large global system, which scales poorly for high polynomial degrees or fine meshes.

2.3.4.2 Hybridizable Galerkin Discontinuous Formulation

Consider the Poisson equation in a domain $\Omega \subset \mathbb{R}^d$ with a diffusion coefficient $\kappa > 0$ and homogeneous Dirichlet boundary conditions. The strong form is

$$-\nabla \cdot (\kappa \nabla u) = f \quad \text{in } \Omega, \qquad u = 0 \quad \text{on } \partial\Omega.$$

To facilitate the HDG discretization, we rewrite it as a first-order mixed system by introducing the flux variable $\mathbf{q}$:

$$\mathbf{q} + \kappa \nabla u = 0, \qquad \nabla \cdot \mathbf{q} = f.$$

This formulation is equivalent to the original PDE but is more convenient for DG methods because it allows flux continuity to be imposed weakly.

The computational mesh $\mathcal{T}_h$ is divided into elements K. HDG introduces three sets of unknowns:

1. u_h, the scalar field inside each element,
2. $\mathbf{q}_h$, the flux inside each element,
3. $\hat{u}_h$, the *trace variable* on the mesh skeleton.

The first two are discontinuous between elements, while the trace variable $\hat{u}_h$ is single-valued on each interface. This variable couples local element problems.

The local weak formulation on an element K is

$$\int_K \mathbf{q}_h \cdot \mathbf{v}_h - \int_K \kappa u_h (\nabla \cdot \mathbf{v}_h) + \int_{\partial K} \kappa \hat{u}_h (\mathbf{v}_h \cdot n) = 0,$$
$$- \int_K \mathbf{q}_h \cdot \nabla w_h + \int_{\partial K} \hat{\mathbf{q}}_h \cdot n \, w_h = \int_K f \, w_h,$$

for all test functions $\mathbf{v}_h$ and w_h. Boundary terms involve the numerical trace $\hat{u}_h$ and the numerical flux $\hat{\mathbf{q}}_h \cdot n$. Notice that this development will be further explained in future sections.

Numerical Fluxes and Weak Continuity

The numerical flux on each face is usually defined as

$$\hat{\mathbf{q}}_h \cdot n = \mathbf{q}_h \cdot n + \tau\big(u_h - \hat{u}_h\big),$$

where $\tau > 0$ is a stabilization parameter.

If $u_h = \hat{u}_h$, the flux reduces to the physical flux $\mathbf{q}_h \cdot n$. If there is a mismatch, the penalty term enforces weak continuity. This definition ensures local conservation because the flux leaving one element is equal to the negative of the flux entering its neighbour.

Global Hybridization

After expressing local problems in terms of $\hat{u}_h$, all interior unknowns can be eliminated.

1. Each element problem is solved locally using $\hat{u}_h$ as boundary data.
2. Local solutions are assembled into a global system involving only $\hat{u}_h$.
3. Solving this *Schur-complement system* for $\hat{u}_h$ allows the recovery of u_h and $\mathbf{q}_h$ locally and in parallel.

This procedure drastically reduces the size of the global system, particularly for high-order or 3D problems, and enables efficient parallelization.

2.3.4.3 Advantages and Theoretical Insights of the HDG Method

The Hybridizable Discontinuous Galerkin (HDG) method combines the local accuracy and flexibility of classical Discontinuous Galerkin (DG) methods with the reduced global cost of continuous Galerkin formulations. Several key advantages make HDG especially effective for elliptic problems like the Poisson equation.

- First, **hybridization** enables the elimination of element-interior unknowns via static condensation, resulting in a global system that involves only the trace variable $\hat{u}_h$, defined on the mesh skeleton. This dramatically reduces the number of globally coupled degrees of freedom. The size of the global HDG system becomes comparable to that of standard FEM for the same polynomial degree k, despite retaining a DG-like local structure.

- Second, HDG methods enforce **local conservation**. The formulation ensures that the normal flux $\mathbf{q}_h \cdot \mathbf{n}$ is continuous across element boundaries in a weak sense, which guarantees elementwise conservation of quantities like mass or energy—something not always preserved in continuous FEM.

- In terms of accuracy, HDG offers **high-order convergence**, with both u_h and $\mathbf{q}_h$ achieving optimal $\mathcal{O}(h^{k+1})$ convergence in the L^2 norm. Moreover, a simple *local postprocessing* step can further enhance the accuracy of u_h to $\mathcal{O}(h^{k+2})$, a phenomenon known as *superconvergence*.

- The method also allows **flexible discretizations**, supporting non-matching meshes, variable polynomial degrees (p-adaptivity), and complex geometries. Since continuity is imposed weakly via $\hat{u}_h$, HDG handles discontinuities naturally, while still producing globally smooth approximations.

- From a numerical perspective, the elimination of local variables improves the **conditioning** of the global system relative to standard DG methods. The global matrix involves only face variables and tends to have conditioning comparable to continuous FEM. Furthermore, the stabilization parameter τ used in the numerical fluxes can be chosen quite flexibly without compromising stability.

- Finally, HDG is well-suited for **parallel computing**. Local problems can be solved independently and simultaneously, which is ideal for modern multi-core and distributed memory systems. Communication is only required for the assembly and solution of the global trace system, which is significantly smaller.

From a theoretical standpoint, HDG can be seen as a hybridized mixed FEM. Its formulation can be rigorously analyzed using saddle-point theory, and optimal convergence results hold even under limited regularity assumptions due to the trace-based formulation.

In summary, HDG delivers the local accuracy, conservation, and flexibility of DG methods, while maintaining a global complexity similar to FEM. This balance makes it particularly attractive for large-scale and high-order simulations of elliptic PDEs, such as the Poisson equation.

Hybridizable Discontinuous Galerkin (HDG) method use justification

Figure 5 from [31] illustrates a **decision framework for selecting numerical schemes** to solve partial differential equations (PDEs) or multiphysics systems in an efficient and stable manner. The motivation behind this approach is the increasing complexity in modern computational methods: while a wide variety of high-performance discretizations exist, from classical finite volumes to advanced discontinuous Galerkin variants, selecting the most appropriate method for a given problem is not straightforward.

In the context of this work, the decision framework supports the selection of the **Hybridizable Discontinuous Galerkin (HDG)** method because:

1. It is highly suited for GPU execution due to its element-local computations and condensed global solve.

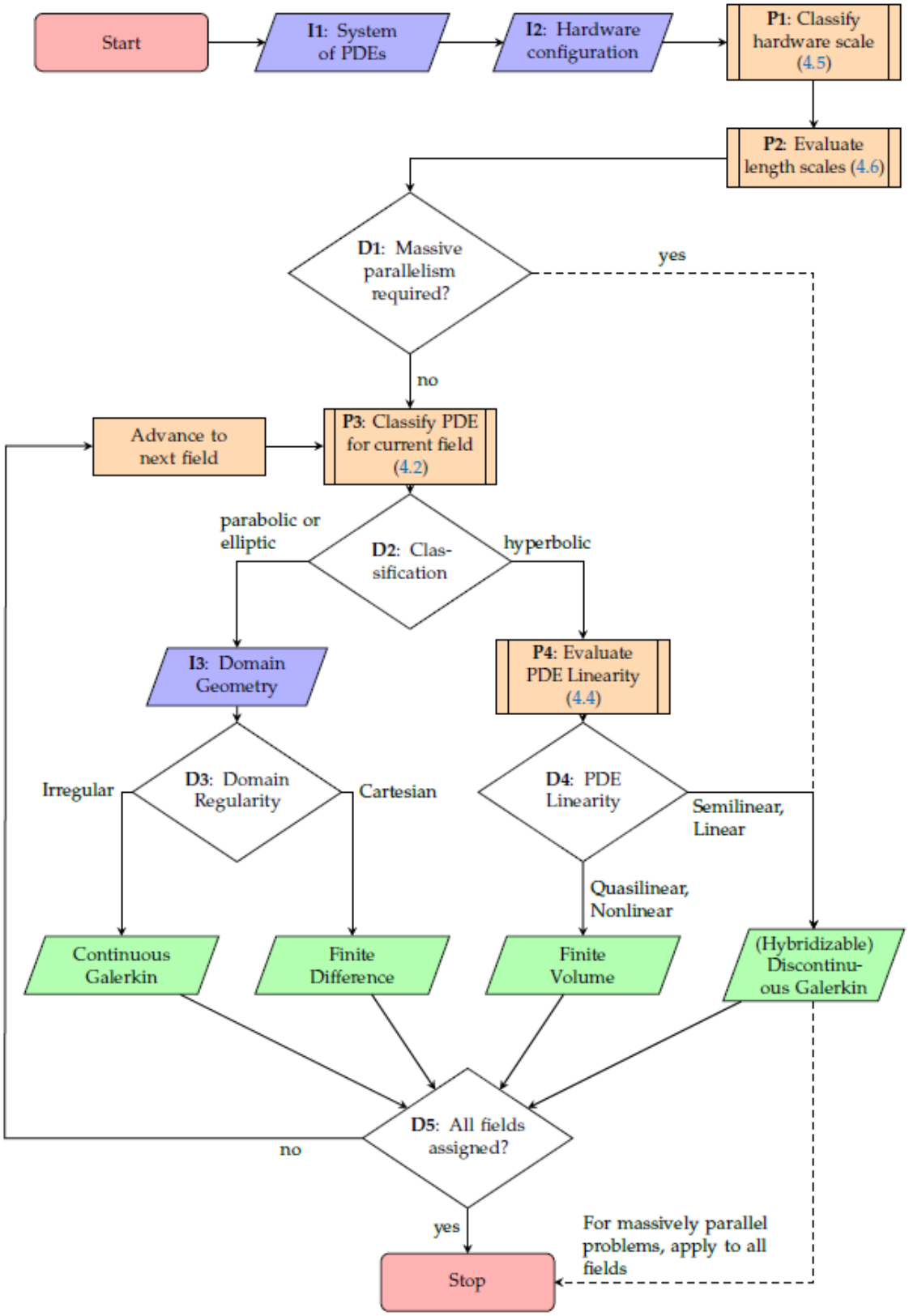


Figure 2.4: *Graphic summary of the proposed process for choosing appropriate numerical schemes. Inputs (I) are given by purple trapezoids, decision points (D) by white diamonds and processes (P) by orange rectangles. Processes with additional vertical bars denote more complex processes and have references to their respective sections. Results are shown in green trapezoids. [31].*

2. It provides high-order accuracy and robustness for hyperbolic and vortex-dominated flows, which are common in unsteady aerodynamics and wind energy applications.

3. It reduces the cost of elliptic solves, which is valuable in multiphysics problems or incompressible formulations where pressure Poisson equations appear.

2.4 GPU Memory Hierarchy and Architecture

Graphics Processing Units (GPUs) are designed for highly parallel computation and incorporate a sophisticated memory hierarchy that plays a central role in their performance. Unlike traditional CPU memory architectures, which prioritize latency and general-purpose flexibility, the GPU memory system is structured to support the execution of thousands of lightweight threads simultaneously. Understanding this memory layout is essential for efficiently designing algorithms for high-performance computing tasks such as artificial intelligence (AI), scientific simulation, and real-time rendering [38].

2.4.1 On-Chip Memory: Registers and Cache

At the top of the memory hierarchy are fast on-chip memory resources collectively known as *static RAM* (SRAM) [13]. These include the following:

- **Registers**: These are the fastest memory components available to a GPU core. Each core has access to a small number of registers that hold immediate values being operated on. Because they reside inside the core itself, register access is virtually instantaneous, enabling high-speed arithmetic operations.

- **L1 Cache**: Located within each Streaming Multiprocessor (SM), the Level-1 cache temporarily stores data that is accessed frequently by threads within that SM. This cache reduces the need for repeated access to slower global memory and improves throughput, particularly for localized computations.

- **L2 Cache**: This larger, shared cache spans across multiple SMs. It stores data that does not fit into the L1 cache but is still accessed often. By acting as a buffer before accessing global memory, the L2 cache contributes to reducing memory latency and traffic to off-chip DRAM.

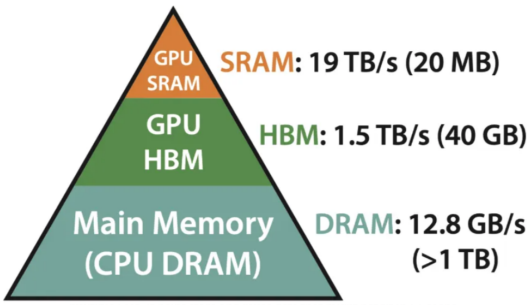


Figure 2.5: *GPU memory scheme [13].*

These memory units dramatically improve efficiency by reducing the number of accesses to the global memory (commonly referred to as VRAM), which is significantly slower. Efficient use of registers and cache is a critical factor in optimizing kernel performance on a GPU.

2.4.2 Global Memory: DRAM and HBM

Beyond the on-chip memory, GPUs use larger and slower memory types to store application data such as model parameters, textures, and simulation state [13].

- **DRAM (Dynamic RAM)**: Typically implemented as GDDR (Graphics Double Data Rate), this is the main memory on the GPU card. While much slower than SRAM, it offers much larger storage capacity. Data frequently moves between DRAM and the faster caches or registers during program execution.
- **High Bandwidth Memory (HBM)**: Found in high-end GPUs, especially those used for scientific computing or AI model training, HBM stacks memory vertically to increase bandwidth and reduce latency. It outperforms GDDR in both throughput and power efficiency but comes at a higher cost and is therefore used selectively in professional-grade hardware.

Due to the performance cost of transferring data between DRAM and on-chip memory, modern GPU programming models emphasize reducing the frequency and volume of such memory transfers to maximize computational efficiency.

2.4.3 Streaming Multiprocessors (SMs)

At the heart of GPU architecture lies the *Streaming Multiprocessor* (SM), which is the fundamental unit of execution. Each SM includes multiple GPU cores (also called CUDA cores in NVIDIA hardware), vector processing units, shared memory (a type of fast, user-manageable SRAM), and control logic [38].

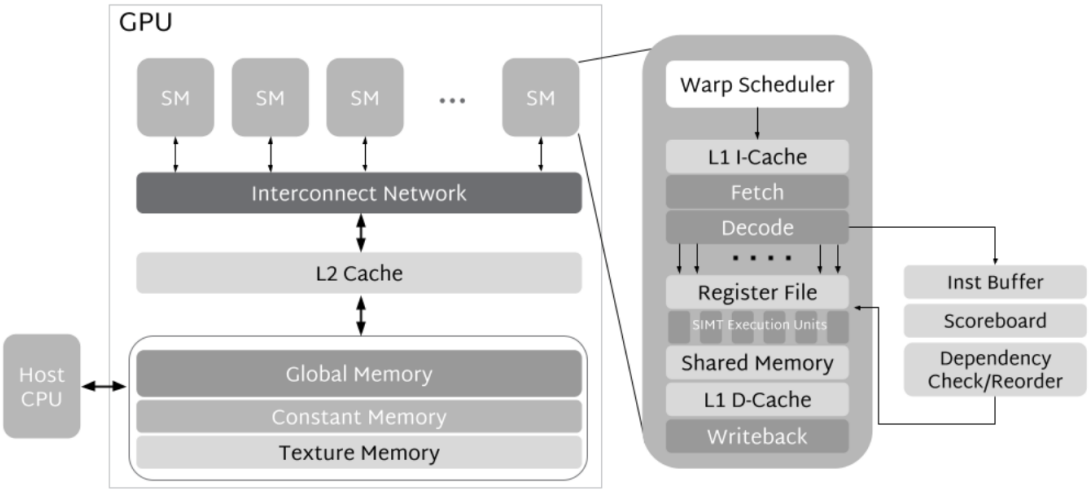


Figure 2.6: *GPU architecture scheme [38].*

When a kernel is launched, the GPU scheduler assigns blocks of threads to available SMs. Each SM then processes its portion of the workload independently, leveraging its internal cores and memory to execute instructions concurrently. The number of SMs in a GPU directly correlates with its ability to execute larger or more complex workloads in parallel.

2.4.4 Program Execution and PIDs

Each parallel unit of computation within the GPU corresponds to a thread executing a specific instance of a kernel. Threads are grouped into blocks, and each thread is assigned a unique Program ID (PID). This PID determines the data segment the thread will operate on.

Threads within a block can cooperate by sharing data via the SM's shared memory and synchronizing their execution. The maximum number of active PIDs per SM depends on the amount of shared memory each thread requires. If a kernel consumes too much shared memory per thread, fewer threads can run concurrently, leading to underutilization of the available computational resources.

2.4.5 Cores, Warps, and Parallelism

Each GPU core is a minimal computational unit capable of executing floating-point operations. However, individual cores do not work alone, they are grouped into execution units called *warps*.

- **Warp Structure**: On NVIDIA GPUs, a warp consists of 32 threads. All threads in a warp execute the same instruction simultaneously but operate on different data, a concept known as Single Instruction, Multiple Threads (SIMT).
- **Warp Divergence**: If the threads within a warp encounter divergent control flow (e.g., some threads enter a conditional branch while others do not), execution becomes serialized for the diverging paths, reducing efficiency. Therefore, aligning workloads to warp-friendly data shapes, such as using multiples of 32 or 64, helps ensure optimal performance.

Warp-level parallelism is a defining feature of GPU architectures and is key to achieving high throughput in data-parallel applications [27].

2.4.6 Best Practices for GPU Optimization

To fully exploit the computational power of GPUs, several optimization principles are commonly applied [27]:

1. **Minimize Global Memory Access**: Reducing the frequency of data transfers between global memory (VRAM) and local caches or registers is essential. Data reuse through shared memory or cache significantly enhances speed.
2. **Maximize Occupancy**: Design kernels such that the required shared memory and registers per thread are low enough to enable many threads to execute simultaneously on each SM.
3. **Align Workloads with Warp Size**: Ensure that computational grids are defined using dimensions that are multiples of the warp size (32 for NVIDIA, 64 for AMD) to avoid idle threads and underutilization of resources.

Conclusion

In summary, GPUs offer a hierarchical memory and execution structure optimized for parallelism at multiple levels. From registers and caches to SMs and warps, each architectural component contributes to the GPU's ability to execute large-scale, data-parallel computations efficiently. Mastery of these details enables developers to harness the full potential of modern GPUs.

CHAPTER 3

HDG discretization for the Poisson problem

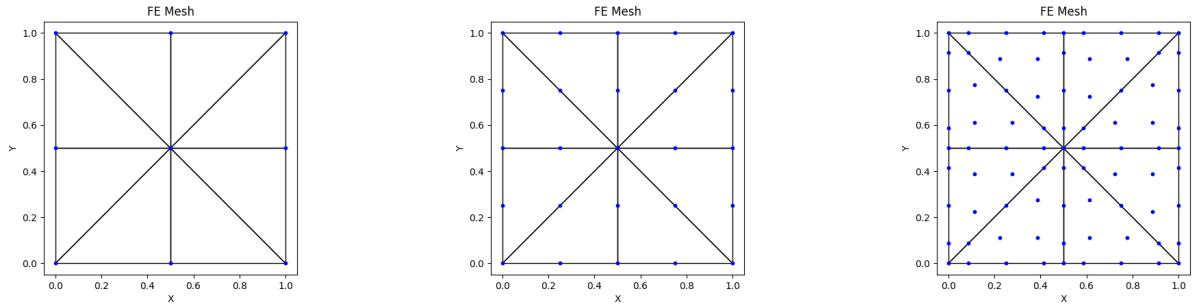
Contents

The problem is to approximate the solution u of the Poisson equation with Dirichlet boundary conditions on a mesh of triangles that will be refined and possess faces' polynomial degrees (k) varying across the mesh. The methodology below explains the mathematical model, data, the particular discretization, and implementation details.

3.1 Mesh Initialization and Loading

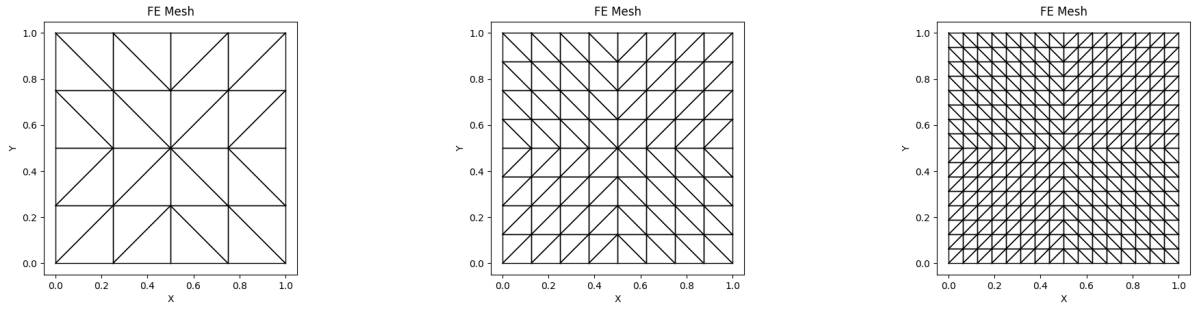
We control the test case with two integers: *matrix number* selects the mesh file and *degree* sets the polynomial order k of the reference-element basis. Together they determine the geometric resolution (elements) and the approximation order on each element.

Firstly, loading the adecuated mesh returns the nodal coordinates $\mathbf{X} \in \mathbb{R}^{n_{\text{nodes}} \times 2}$ with $X[i] = (x_i, y_i)$, the connectivity $\mathbf{T} \in \mathbb{Z}^{n_{\text{elems}} \times 3}$ whose row $T[e] = [n_{e,0}, n_{e,1}, n_{e,2}]$ lists the triangle's global node indices, and the Dirichlet boundary map *Tb_Dirichlet* used to fix the corresponding degrees of freedom. These three objects fully specify geometry, topology, and boundary data for the subsequent HDG assembly. Moreover, the grids with different refinements and polynomial degrees are represented in Figure 3.1 and Figure 3.2 to see the look of the problem. Notice that blue points in Figure 3.1 are the Gauss points, and as the degree is increased, the fulfillment of the domain is better.

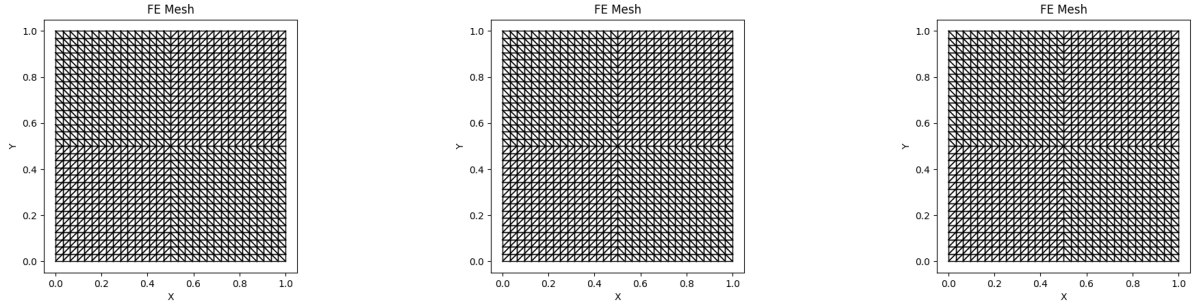


(a) Mesh 1, $k = 1$ (b) Mesh 1, $k = 2$ (c) Mesh 1, $k = 4$

Figure 3.1: *Coarser mesh (mesh 1), increasing polynomial degree. As k grows the mesh remains the same, but the number of Gauss points (blue) is higher.*



(a) Mesh 2 (b) Mesh 3 (c) Mesh 4



(d) Mesh 5 (e) Mesh 6 (f) Mesh 7

Figure 3.2: *Multiple meshes from coarse to finer discretizations.*

3.2 HDG Preprocessing

We turn the element–to–node table T (each row lists the three vertex IDs of a triangle) into a complete, unique catalog of mesh faces. First, we scan every triangle edge and classify it as *interior* (shared by two elements) or *boundary* (owned by one element). Next, we assign a consecutive global face ID to each unique edge: the same ID is written into both neighboring elements for interior faces, and a new ID is given to each boundary face. The result is:

- $F \in \mathbb{Z}^{n_{\text{elems}} \times 3}$: for element e and local face f, $F[e, f]$ is the global face ID.

- *infoFaces*: raw listings of interior faces (e_1, f_1, e_2, f_2, e) and exterior faces (e, f), kept for later use.

3.2.1 The Reference Element concept

HDG discretization has the flexibility of being implemented on unstructured meshes (unlike classical finite-difference and many finite-volume schemes) thanks to the *reference element* concept: all algebra is formulated on a single, simple prototype cell, and geometry enters only through an elementwise mapping and its Jacobian. In practice, basis functions, quadrature rules, and numerical fluxes are evaluated once on the reference element and then reused across all physical elements, which simplifies the code and preserves high-order accuracy on complex meshes.

Therefore, in the present HDG case, for triangles, this reference element is represented in

$$\widehat{K} = \{(\xi, \eta) \in \mathbb{R}^2 : \ \xi \geq 0, \ \eta \geq 0, \ \xi + \eta \leq 1\}.$$

Every physical triangle K_e in the mesh is obtained from $\widehat{K}$ by an element–wise affine map

$$\boldsymbol{x}(\xi, \eta) = \mathbf{A}_e \begin{bmatrix} \xi \\ \eta \end{bmatrix} + \boldsymbol{b}_e, \qquad \mathbf{J}_e = \nabla_{\widehat{K}} \boldsymbol{x} = \mathbf{A}_e,$$

so that all volume integrals on K_e are pulled back to $\widehat{K}$ and scaled by $|\det \mathbf{J}_e|$. This simple idea (compute once on $\widehat{K}$, reuse everywhere through $\mathbf{J}_e$)is what makes the method efficient and clean.

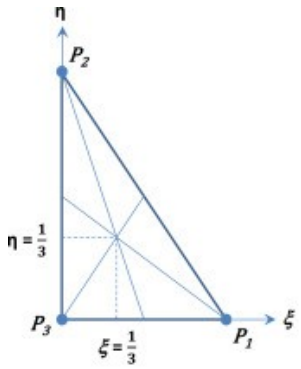


Figure 3.3: The reference triangle $\widehat{K}$ with vertices numbered counter–clockwise [10].

The unknowns inside each element are expanded in a polynomial basis defined on $\widehat{K}$. For a degree p triangle, the number of local basis functions is $n_{\text{loc}} = \frac{(p+1)(p+2)}{2}$. We place their interpolation nodes at the vertices, along the edges, and (for $p \geq 3$) in the interior using Fekete points (observable in Figure 3.1). Evaluating basis values and their reference gradients $\nabla_{\widehat{K}} N_i$ at quadrature points on $\widehat{K}$ gives everything needed to build the local stiffness and mass matrices. Physical gradients follow from

$$\nabla_{K_e} \varphi_i(\boldsymbol{x}) = \mathbf{J}_e^{-T} \nabla_{\widehat{K}} N_i(\xi, \eta).$$

HDG couples elements only through their faces. Each physical edge of K_e is the image of a 1D reference segment $\widehat{E} = [-1, 1]$ via an edge map $\boldsymbol{x}_E(\xi)$ with 1D Jacobian $J_e^{(1d)} = \|\partial_\xi \boldsymbol{x}_E\|$. Numerical fluxes and trace unknowns live on these edges, so we also fix a 1D basis and Gauss points on $\widehat{E}$. A consistent local edge ordering (the *faceNodes* and *faceNodes1d* maps) guarantees that the same physical edge is seen with the same orientation by its two neighboring elements, crucial to assemble

skeleton terms with the correct sign.

Volume integrals $\int_{K_e}(\cdot)\,d\boldsymbol{x}$ become $\int_{\widehat{K}}(\cdot)\,|\det \mathbf{J}_e|\,d\xi\,d\eta$, and edge integrals $\int_{\partial K_e}(\cdot)\,ds$ become $\int_{\widehat{E}}(\cdot)\,J_e^{(1d)}\,d\xi$. Thus, a single set of 2D cubature points/weights on $\widehat{K}$ and 1D Gauss points/weights on $\widehat{E}$ is sufficient for the whole mesh, as geometry only changes the Jacobian factors.

Finally, by collecting all of the above into a single *referenceElement* struct (or Python dictionary), we **provide every HDG kernel** with immediate access to:

- Quadrature nodes and weights (2D and 1D),
- Shape function values and gradients,
- Local face and interior node maps,
- Nodal coordinate lists for element-to-physical-domain mappings.

This modular design decouples reference-element setup from mesh traversal and linear-system assembly, greatly simplifying code maintainability and extension to higher polynomial degrees or alternative element shapes.

3.3 Hybridizable Discontinuous Galerkin (HDG) System Resolution

The resolution of the Poisson problem with Dirichlet boundary conditions using the HDG method proceeds through several stages, from the weak formulation to the final global algebraic system. This section provides a detailed derivation of the process.

Poisson Problem and Mixed Formulation

We consider the Poisson problem with Dirichlet boundary conditions:

$$\nabla \cdot \mathbf{q} = f \qquad \text{in } \Omega, \tag{3.1}$$

$$\mathbf{q} + \kappa \nabla u = 0 \qquad \text{in } \Omega, \tag{3.2}$$

$$u = 0 \qquad \text{on } \partial\Omega, \tag{3.3}$$

where:

- $\mathbf{q}$ is an auxiliary flux field,
- u is the scalar unknown,
- $\kappa > 0$ is the diffusion coefficient,
- f is the volumetric source term.

The HDG method rewrites the system in **mixed form**, introducing $\mathbf{q} = -\kappa \nabla u$. This allows us to define $\mathbf{q} \in H(\mathrm{div}, \Omega)$ as a new vector unknown while keeping $u \in L^2(\Omega)$.

Weak Form of the Divergence Equation

Multiplying the divergence equation $\nabla \cdot \mathbf{q} = f$ by a scalar test function w and integrating over an element K gives

$$\int_K w \nabla \cdot \mathbf{q} \, dx = \int_K w f \, dx. \tag{3.4}$$

Applying integration by parts yields

$$-\int_K \nabla w \cdot \mathbf{q} \, dx + \int_{\partial K} w \, \mathbf{q} \cdot \mathbf{n} \, ds = \int_K w f \, dx, \tag{3.5}$$

where $\mathbf{n}$ is the outward normal to ∂K.

To close the system, we replace the flux at the boundary with the **numerical flux** $\hat{\mathbf{q}} \cdot \mathbf{n}$, which will later depend on the hybrid unknown $\hat{u}$.

Weak Form of the Constitutive Equation

The constitutive equation $\mathbf{q} + \kappa \nabla u = 0$ is tested with a vector test function $\mathbf{v}$:

$$\int_K \mathbf{v} \cdot \mathbf{q} \, dx + \kappa \int_K \mathbf{v} \cdot \nabla u \, dx = 0. \tag{3.6}$$

Integration by parts of the second term gives

$$\int_K \mathbf{v} \cdot \mathbf{q} \, dx - \kappa \int_K (\nabla \cdot \mathbf{v}) u \, dx + \kappa \int_{\partial K} (\mathbf{v} \cdot \mathbf{n}) \hat{u} \, ds = 0, \tag{3.7}$$

where the **hybrid trace** $\hat{u}$ mediates information exchange between elements in HDG.

Numerical Flux Definition

To ensure inter-element communication and stability, we define the **numerical flux** as

$$\hat{\mathbf{q}} \cdot \mathbf{n} = \mathbf{q} \cdot \mathbf{n} + \tau (u - \hat{u}), \tag{3.8}$$

where $\tau > 0$ is the stabilization parameter.
This definition enforces a Robin-like coupling at the faces, allowing the local problem to be expressed entirely in terms of $\mathbf{u}$ and $\mathbf{q}$.

Local Discrete System

After discretization with shape functions N (volume) and N_{1d} (faces), the local system for an element K is expressed as a block system:

$$\begin{bmatrix} A_{uu} & A_{uq} \\ A_{qu} & A_{qq} \end{bmatrix} \begin{bmatrix} u \\ \mathbf{q} \end{bmatrix} = - \begin{bmatrix} A_{u\hat{u}} \\ A_{q\hat{u}} \end{bmatrix} \hat{u} + \begin{bmatrix} f_u \\ f_q \end{bmatrix}, \tag{3.9}$$

where:

- A_{uu}, A_{uq}, A_{qu}, A_{qq} arise from volume integrals,
- $A_{u\hat{u}}$, $A_{q\hat{u}}$ come from face contributions,
- f_u and f_q are the local source term contributions.

This block system corresponds to the **elemental HDG system**.

Static Condensation (Local Elimination)

Elemental unknowns u_e and q_e can be written explicitly in terms of the face unknown $\hat{u}$, hence, static condensation is achieved at the element level. Defining

$$A = \begin{bmatrix} A_{uu} & A_{uq} \\ A_{qu} & A_{qq} \end{bmatrix}, \quad B = \begin{bmatrix} A_{u\hat{u}} \\ A_{q\hat{u}} \end{bmatrix}, \quad F = \begin{bmatrix} f_u \\ f_q \end{bmatrix}, \tag{3.10}$$

we obtain the **local mapping**:

$$\begin{bmatrix} u \\ \mathbf{q} \end{bmatrix} = -A^{-1} B\, \hat{u} + A^{-1} F. \tag{3.11}$$

This defines the **elemental matrices**:

$$U = -A^{-1} A_{u\hat{u}}, \quad Q = -A^{-1} A_{q\hat{u}}, \quad U_f = (A^{-1}F)_u, \quad Q_f = (A^{-1}F)_q. \tag{3.12}$$

These matrices allow expressing the interior fields in terms of the trace $\hat{u}$.

Assembly of the Global System

The global unknowns at the end are the trace values $\hat{u}$ at all faces. The **global HDG system** is assembled as:

$$K\, \hat{u} = f, \tag{3.13}$$

where the global matrix K and vector f are obtained by summing contributions from each element:

$$K_e = A_{lq} Q_e + A_{lu} U_e + A_{ll}, \tag{3.14}$$

$$f_e = -\big(A_{lq} Q_f + A_{lu} U_f\big). \tag{3.15}$$

Here:

- A_{lq}, A_{lu}, A_{ll} are the local face matrices,
- Proper orientation handling (*flipFace*) ensures interior faces have coherent normals.

Each element adds its submatrix to the global sparse system, connecting the DOFs corresponding to its three faces.

Final Remarks

The final sparse system only involves $\hat{u}$ as global DOFs, while the interior variables u and $\mathbf{q}$ are reconstructed locally after solving the global system.

This procedure allows HDG to achieve:

- Reduced global DOF count compared to classical DG,
- Local static condensation for efficiency,
- Easy post-processing of high-order fluxes.

3.4 Boundary Conditions in the HDG discreatization

The proper imposition of boundary conditions is a fundamental step in any Computational Fluid Dynamics (CFD) simulation, particularly when using high-order finite element or Hybridizable Discontinuous Galerkin (HDG) methods. In this work, Dirichlet boundary conditions are applied on the exterior faces of the mesh by projecting the analytical solution onto the face nodes using an L^2 projection. This section details the theoretical formulation, the computation of degrees of freedom (DoFs), the projection procedure, and the system reduction applied to incorporate Dirichlet conditions.

Degrees of Freedom on Faces

In the HDG formulation, the unknowns are defined on the **faces** of the mesh elements rather than in the interior of the elements. For a polynomial approximation of degree p, each face has:

$$n_{\text{face nodes}} = p + 1$$

in one dimension.

- **Interior faces** are shared by two elements and carry unknown degrees of freedom λ.
- **Exterior faces** lie on the domain boundary $\partial\Omega$ and are subjected to Dirichlet boundary conditions.

The global vector of unknowns $\hat{u}$ is then split into two groups:

$$\hat{u} = \begin{bmatrix} \lambda_{\text{interior}} \\ u_{\text{Dirichlet (boundary)}} \end{bmatrix}$$

where λ are the face-based unknowns on interior faces and $u_{\text{Dirichlet}}$ are known values on exterior faces.

L^2 Projection on Boundary Faces

The Dirichlet boundary values are imposed by projecting a given analytical solution $u(x, y)$ onto the polynomial space defined by the face nodes. This is achieved through an L^2 projection, ensuring that the numerical approximation u_h satisfies:

$$\int_{\Gamma_f} (u(x, y) - u_h(\xi))\, \phi_i(\xi)\, ds = 0 \quad \forall\, i$$

where $\phi_i(\xi)$ are the face shape functions in the local reference coordinate $\xi \in [-1, 1]$. Expanding $u_h(\xi)$ in the local basis:

$$u_h(\xi) = \sum_{j=1}^{n_f} u_j\, \phi_j(\xi)$$

leads to a local linear system for each face f:

$$M_f\, \mathbf{u}_f = \mathbf{b}_f$$

where:

$$M_{ij} = \int_{\Gamma_f} \phi_i(\xi)\phi_j(\xi)\, ds, \qquad b_i = \int_{\Gamma_f} u(x(\xi), y(\xi))\phi_i(\xi)\, ds$$

To compute these integrals in the reference coordinate ξ, the line element ds is mapped from the reference segment $[-1, 1]$ to the physical face as:

$$ds = \|\mathbf{x}'(\xi)\|\, d\xi$$

$$\mathbf{x}'(\xi) = \begin{bmatrix} x'(\xi) \\ y'(\xi) \end{bmatrix} = \frac{d}{d\xi} \sum_k \mathbf{x}_k \phi_k(\xi)$$

The discrete approximation using Gauss quadrature is:

$$M_{ij} \approx \sum_{q=1}^{n_g} \phi_i(\xi_q)\phi_j(\xi_q) w_q \|\mathbf{x}'(\xi_q)\|$$

$$b_i \approx \sum_{q=1}^{n_g} \phi_i(\xi_q)\, u(x(\xi_q), y(\xi_q))\, w_q \|\mathbf{x}'(\xi_q)\|$$

This projection computes the nodal boundary values $\mathbf{u}_f$ for each exterior face f.

Dirichlet Boundary Condition Application

Once the boundary nodal values $\mathbf{u}_{\text{Dirichlet}}$ are computed, they are incorporated into the global system:

$$K\hat{u} = f$$

where K is the global face-face stiffness matrix and f is the right-hand side vector.
The global system can be partitioned as:

$$\begin{bmatrix} K_{UU} & K_{UD} \\ K_{DU} & K_{DD} \end{bmatrix} \begin{bmatrix} \lambda \\ u_{\text{Dirichlet}} \end{bmatrix} = \begin{bmatrix} f_U \\ f_D \end{bmatrix}$$

Here, U refers to unknown (interior) DoFs and D refers to Dirichlet (boundary) DoFs.

System Reduction

Since the boundary values are known, the system can be reduced to solve only for λ:

$$f_{\text{reduced}} = f_U - K_{UD} u_{\text{Dirichlet}}$$

$$K_{\text{reduced}} = K_{UU}$$

$$K_{\text{reduced}} \lambda = f_{\text{reduced}}$$

After solving for λ, the full global vector is reconstructed:

$$\hat{u} = \begin{bmatrix} \lambda \\ u_{\text{Dirichlet}} \end{bmatrix}$$

Therefore, this approach ensures that the Dirichlet conditions are imposed in a weak form consistent with the HDG method, improving stability and accuracy.

3.5 Element Solution in the HDG discretization

In the Hybridizable Discontinuous Galerkin (HDG) method, the numerical solution within each finite element can be reconstructed after solving the global system for the *hybrid unknown* defined on the mesh skeleton. This section details the methodology for computing the **elemental solution** (u, q), including the theoretical formulation, input/output description.

HDG methods introduce an additional unknown, denoted as λ, representing the solution trace on the element interfaces (faces). This hybrid variable is the only globally coupled degree of freedom (DOF), while the internal unknowns u (scalar solution, e.g. pressure or temperature) and q (flux vector field) are computed locally at the element level.

The local system for an element K can be written in block form as:

$$\begin{bmatrix} \mathbf{A}_{uu} & \mathbf{A}_{uq} \\ \mathbf{A}_{qu} & \mathbf{A}_{qq} \end{bmatrix} \begin{bmatrix} \mathbf{u}_K \\ \mathbf{q}_K \end{bmatrix} = \begin{bmatrix} \mathbf{f}_u \\ \mathbf{f}_q \end{bmatrix} + \begin{bmatrix} \mathbf{B}_u \\ \mathbf{B}_q \end{bmatrix} \lambda_{\partial K}, \tag{3.16}$$

where:

- $\mathbf{u}_K \in \mathbb{R}^{N_e}$ is the vector of elemental scalar DOFs.
- $\mathbf{q}_K \in \mathbb{R}^{dN_e}$ is the flux vector (with d components per node in d dimensions).
- $\lambda_{\partial K}$ are the face unknowns (trace DOFs) associated with the element boundary ∂K.
- $\mathbf{A}_{**}$ and $\mathbf{B}_*$ arise from the HDG weak formulation and numerical fluxes.
- $\mathbf{f}_u$, $\mathbf{f}_q$ are local source contributions.

Solving (3.16) for $\mathbf{u}_K$ and $\mathbf{q}_K$ leads to the classical HDG elemental solution formula:

$$\mathbf{u}_K = \mathbf{U}_U \, \lambda_{\partial K} + \mathbf{U}_f, \tag{3.17}$$

$$\mathbf{q}_K = \mathbf{Q}_U \, \lambda_{\partial K} + \mathbf{Q}_f, \tag{3.18}$$

where the matrices $(\mathbf{U}_U, \mathbf{Q}_U)$ and vectors $(\mathbf{U}_f, \mathbf{Q}_f)$ are precomputed during the *local static condensation* stage:

$$\mathbf{U}_U = -\mathbf{A}_{uu}^{-1}\mathbf{B}_u, \quad \mathbf{U}_f = \mathbf{A}_{uu}^{-1}\mathbf{f}_u,$$

and similarly for $\mathbf{Q}_U$ and $\mathbf{Q}_f$.

Once λ is known from the global solve, the interior solution of all elements is reconstructed without any additional global communication.

Final Remarks

This postprocessing step is essential for:

- Visualizing the reconstructed physical fields inside elements.
- Evaluating derived quantities, such as vorticity or divergence of flux.
- Comparing with other CFD solvers or experimental data on a nodal basis.

This methodology forms the bridge between the hybrid unknown λ and the physically meaningful field variables in the domain, completing the HDG pipeline from global solve to local field recovery.

3.6 Local Postprocessing and Superconvergence in HDG

A key advantage of the Hybridizable Discontinuous Galerkin (HDG) method is its ability to produce a *superconvergent* solution through local postprocessing. Given a polynomial degree k, the standard HDG solution $u_h \in \mathbb{P}^k$ achieves $\mathcal{O}(h^{k+1})$ accuracy in the L^2-norm. A postprocessed field $u^* \in \mathbb{P}^{k+1}$ can be computed on each element, yielding improved convergence of order $\mathcal{O}(h^{k+2})$. This is precisely a key feature that standard DG cannot achieve.

Mathematical Formulation

After solving the HDG global system, we have:

- $u_h \in \mathbb{P}^k$: the scalar approximation,
- $\mathbf{q}_h = \nabla u_h \in \mathbb{P}^k$: the local gradient.

The postprocessed solution u^* satisfies the local problem:

$$\int_K \nabla w \cdot \nabla u^* \, d\mathbf{x} = \int_K \nabla w \cdot \mathbf{q}_h \, d\mathbf{x}, \quad \forall w \in \mathbb{P}^{k+1}(K), \tag{3.19}$$

subject to the mean-value constraint

$$\int_K u^* \, d\mathbf{x} = \int_K u_h \, d\mathbf{x}, \tag{3.20}$$

which enforces uniqueness.

Reference Element and Geometric Mapping

To implement Eq. (3.19) numerically, integrals are evaluated using an enriched reference element. Let $\widehat{K}$ be the standard master triangle with vertices $(0,0)$, $(1,0)$, $(0,1)$. Two types of shape functions are involved:

- **Geometric functions:** $N_{\text{Geo},i}(\xi,\eta) \in \mathbb{P}^k$ are used to define the mapping

$$\mathbf{x}(\xi,\eta) = \sum_i N_{\text{Geo},i}(\xi,\eta)\, \mathbf{x}_i,$$

  where $\mathbf{x}_i$ are the physical coordinates of the element.

- **Solution basis functions:** $\{\phi_j\} \in \mathbb{P}^{k+1}(\widehat{K})$ span the enriched solution space for u^*.

These bases are evaluated at a suitable quadrature rule over $\widehat{K}$, allowing accurate computation of stiffness matrices and right-hand side integrals.

Local Element Assembly

For each physical element K, the following terms are computed:

- The local stiffness matrix:

$$K_{ij} = \int_K \nabla \phi_i \cdot \nabla \phi_j \, d\mathbf{x},$$

- The Neumann-type source vector:

$$B_i = \int_K \nabla \phi_i \cdot \mathbf{q}_h \, d\mathbf{x},$$

- The integrals of shape functions:
$$c_i = \int_K \phi_i \, d\mathbf{x},$$
- The scalar integral $\int_K u_h \, d\mathbf{x}$.

These terms are assembled into a local saddle-point system with a Lagrange multiplier λ:

$$\begin{bmatrix} \mathbf{K} & \mathbf{c} \\ \mathbf{c}^T & 0 \end{bmatrix} \begin{bmatrix} \mathbf{u}^* \\ \lambda \end{bmatrix} = \begin{bmatrix} \mathbf{B} \\ \int_K u_h \end{bmatrix}.$$

This system enforces both the weak form in Equation 3.19 and the constraint Equation 3.20.

High-Order Basis Functions for $\mathbb{P}^{k+1}$

To represent u^*, we use high-order nodal shape functions built from an orthogonal Dubiner basis on the reference triangle. Starting from modal Jacobi polynomials $\psi_{i,j}(\xi, \eta)$, we form a nodal basis $\ell_n(\xi, \eta)$ via the Vandermonde matrix V:

$$\ell_n(\xi, \eta) = \sum_{m=1}^{N_p} \left(V^{-1}\right)_{nm} \psi_m(\xi, \eta), \quad N_p = \frac{(k+2)(k+3)}{2}.$$

These basis functions and their derivatives are evaluated at quadrature points to perform the integrals required in the local system.

Advantages of HDG Postprocessing

- **Superconvergence:** Improves the convergence rate of the scalar field to $\mathcal{O}(h^{k+2})$.
- **Locality:** All operations are local to each element, so no global solve is needed.
- **Accuracy:** Higher-degree reconstruction offers improved resolution, especially for visualization or error estimation.

Thus, local postprocessing in HDG provides an efficient path to high-accuracy solutions with minimal computational overhead.

CHAPTER 4

GPU IMPLEMENTATION

Contents

4.1 GPU Parallel Programming

This section explains the GPU programming model, the memory hierarchy, and the practices we applied so that a reader understands the rationale behind the decisions taken in the different GPU kernels. Those strategies referenced here come from the Python notebooks developed by Miguel Encinar [7] and Numba/Cupy/NVIDIA guides available on the internet [29] [27] [28] [6].

GPUs implement a massively parallel architecture that differs significantly from CPUs. A GPU kernel (a function marked to run on the device) is executed by thousands of threads in parallel. Threads are organized hierarchically: adjacent threads form a warp (typically 32 threads on NVIDIA hardware), warps form a thread block, and blocks form a grid. All threads in a warp execute in lockstep, SIMT (Single instruction, multiple threads) execution, so branch divergence within a warp causes serialization and efficiency loss. Using CUDA's (Compute Unified Device Architecture) built-in 3D index variables (threadIdx, blockIdx, blockDim), each thread computes a unique global index. Therefore, a kernel is executed as a grid of blocks of threads [28]. Each block runs on one streaming multiprocessor (SM) and can synchronize internally (*syncthreads()*); blocks cannot easily synchronize with each other except by ending a kernel launch.

NVIDIA GPUs support extremely fine-grained parallelism: each SM can have up to 2048 active threads (64 warps) concurrently, so a GPU with many SMs can execute on the order of 10^5–10^6 threads at once. To exploit this [27], many threads should be launched (tens of thousands or more) such that almost all SMs are busy. A common guideline is to use 128 - 256 threads per block and launch enough blocks so that every SM has several active blocks [27]. This large number of lightweight GPU threads hides memory and operation latency: if one warp stalls (e.g., waiting for memory), the SM can quickly switch to another ready warp. By contrast, CPU threading is coarser and cannot hide latency at this scale [27].

4.1.1 CUDA memory

- **Per thread.** Each thread uses private registers (the fastest storage), allocated automatically by the compiler. If working data exceeds the available registers, the overflow spills to local memory, which lives in global DRAM and is much slower. Write kernels with modest per-thread footprints to limit register pressure and spills [27].
- **Per block.** Threads in a block share on-chip shared memory (tens of KB per SM) with low latency. They can stage data there, synchronize with __syncthreads(), and reuse it cooperatively—acting as a user-managed cache [27].
- **Higher levels.** There are read-only and constant caches for immutable data, plus an L2 cache shared by all SMs. Global memory (GPU DRAM) is the largest but slowest [27].
- **Host vs. device.** CPU (host) and GPU (device) memories are separate: copy inputs host→device, launch kernels on device data, then copy results device→host. PCIe/NVLink transfers are relatively slow, so minimize them and keep as much work on the GPU as possible before copying back [27].

4.1.2 CuPy and Numba

- **CuPy.** A NumPy-like array library on the GPU: *cupy.ndarray* lives in device memory and vectorized ops call CUDA libraries (cuBLAS, cuFFT, RNG, etc.) [6]. Examples: *cp.random.random((N,N))* allocates on GPU, or $C = A + B$ launches an elementwise kernel. It manages a memory pool and copies via *cp.asarray*/*cp.asnumpy*. Best when the code is already vectorized and math-heavy [6].
- **Numba.** A JIT that turns Python functions into CUDA kernels with *@cuda.jit*. You write Python, use CUDA indexing (*cuda.grid()*, *cuda.syncthreads()*), and choose blocks/threads (see Algorithm 1). First call pays compile time and subsequent calls reuse the kernel. Ideal for custom per-element logic or loop-heavy algorithms [29] [27].
- **Interoperation & workflow.** CuPy and Numba interoperate (zero-copy views between *cupy.ndarray* and Numba's *DeviceNDArray*). A practical pattern: CuPy for high-level arrays/linear algebra (e.g., *cp.linalg.solve*) and Numba for fine-grained kernels (e.g., assembling per-element contributions). This mixes CuPy's tuned libraries with Numba's flexibility [27].

Algorithm 1: SampleFcn with Numba CUDA

```
from numba import cuda

@cuda.jit
def SampleFcn(n, a, x, y, out):
    i = cuda.grid(1)
    if i < n:
        out[i] = a * x[i] + y[i]

# launch like: SampleFcn[blocks, threads_per_block](n, a, x_d, y_d, out_d)
```

4.1.3 Memory Access and Optimization

- **Coalesced global accesses.** First, because global memory is comparatively slow, we should help the hardware merge requests whenever possible. In practice, that means arranging data so that thread t in a warp reads or writes element t of a linear array. With this layout, adjacent threads touch adjacent addresses, and the warp can service many lanes with a single aligned 32–128 B transaction instead of many smaller, scattered ones. Conversely, irregular patterns waste bandwidth and inflate latency [27]. Therefore, when assembling CSR/COO buffers or right-hand sides, we map indices so neighboring threads land on neighboring slots.

- **Shared-memory tiling for reuse.** Moreover, alignment alone is not enough; we also want to reduce how often we hit DRAM. A common remedy is to stage frequently reused values into on-chip shared memory and then reuse them cooperatively across the block, synchronizing with *syncthreads()*. For example, when accumulating element stiffness matrices, we first load basis-function values or geometric factors into a shared "tile" and then iterate over quadrature points without reloading from global memory. Consequently, we cut redundant transactions and raise effective throughput [27].

- **Occupancy and launch configuration.** To hide remaining memory latency, we want many warps ready to run on each SM. Consequently, we choose moderate block sizes (typically 128–256 threads) so multiple blocks can reside concurrently, launch enough blocks so that all SMs stay busy—rule of thumb: blocks $\gtrsim 2 \times$ SM and limit per-thread register usage and per-block shared memory so the scheduler can keep more warps resident. Otherwise, even a well-coalesced kernel may stall simply because too few warps are available to cover latency [27].

- **Minimizing warp divergence.** However, even with good memory behavior, divergent control flow can serialize execution inside a warp. Therefore, we structure kernels so threads in the same warp follow similar paths: we separate boundary-heavy cases into their own kernels when feasible, refactor *if/else* patterns to be predication-friendly, or reorder work so that warps handle homogeneous tasks. As a result, the hardware issues uniform instructions more often, improving utilization [27].

- **Choosing precision** Finally, precision has a direct performance and bandwidth impact. Since FP32 typically offers substantially higher throughput than FP64 (and halves the bytes moved per value), using *float32* where accuracy allows can reduce runtime and memory pressure [27].

4.1.4 Memory Transfers & Data Layout

- **Transfer once, then stay on device.** First, copy each large array to the GPU (*cp.asarray* or *cuda.to_device*) and keep it resident for the whole solve [27]; only bring back the final result and small outputs. In iterative or multi-stage workflows, avoid host↔device round trips inside inner loops [27].
- **Lay out data for efficient access.** Next, store connectivity and neighbor info in contiguous arrays so threads handling element e read a contiguous offset $\propto e$ [6] [27].
- **Accumulate in shared memory, write once globally.** Finally, load each element matrix into shared memory, sum contributions across threads (sync with *syncthreads()*), and let a designated thread issue a single coalesced global write. Use atomics only within shared memory when needed; avoid global atomics via this block-wise accumulation [27].

4.1.5 Final guideline

To summarize the general strategy in GPU programming, [7] python notes are summarized with bullet points, gathering the main conclusions and best practices learnt:

- Keep data on the GPU; avoid host round-trips in loops.
- Map work so neighboring threads touch neighboring memory (coalescing); precompute contiguous write offsets.
- Use shared memory as a cooperative cache; synchronize once, write once; avoid global atomics (limit atomics to shared if needed).
- Prefer 128–256 threads per block and launch many more blocks than SMs; measure and adjust.
- Minimize warp divergence; push special cases to separate kernels or restructure conditions.
- Use float32 unless accuracy requires float64; expect substantial speed/footprint gains.
- Batch small solves/reductions; avoid Python loops; rely on CuPy BLAS/FFT where applicable.
- Always profile, then optimize the true bottleneck (layout/coalescing vs arithmetic vs occupancy).

4.2 HDG Poisson problem kernels

The GPU implementation of the HDG discretization for the Poisson equation is organized into a sequence of specialized kernels, each targeting one stage of the algorithm. In the following subsections, we describe the assembly and solution kernels in detail, exposing their data layout, parallel strategy, and performance considerations.

4.2.1 HDG Poisson assembly kernels

Firstly, to give an overview, one must know that we split the assembly pipeline so that each stage maximizes locality and shared-memory reuse, avoids global atomics and inter-block synchronization, and produces compact device buffers consumed directly by the next stage, all without host transfers. The code stages are:

1. **Stage 1: *volumetric_integration_kernel.*** For each element, evaluate volume integrals and write to small per-element arrays (*Aqq*, *Auq*, *fe*). Where helpful, stage intermediates in shared memory to increase reuse and cut global traffic. This kernel will be explained in detail to show the general strategy taken in the GPU code.

2. **Stage 2: *face_integration_kernel.*** Integrate over faces per element, filling (*Alq*, *Alu*, *All*, *Auu*). Data are laid out so neighboring threads touch neighboring addresses, promoting coalesced accesses. Notice that this kernel is not explained as it uses the same strategy as the Volumetric Integration kernel.

3. **Stage 3: Batched local dense solve on GPU.** Use *cp.linalg.solve* to eliminate local unknowns entirely on the device, producing *UQ* and *fU*.

4. **Stage 4: *flip_and_build_coo_kernel.*** Assemble each element's global contributions into COO triplets and RHS tuples. The kernel handles face orientation ("flip") consistently and emits compact, coalesced *(row, col, val)* records.

Note that the wrapper *hdgMatrixPoisson_gpu(. . .)* orchestrates the launches, keeps all data resident on the device throughout, and converts the accumulated COO structure to CSR at the end.

4.2.1.1 Volumetric integration kernel

Below is depicted the pseudocode for the volumetric integration kernel (Algorithm 4), together with explanations and two small CPU reference boxes (Algorithms 2 and 3) to better see some differences between CPU and GPU implementations:

Memory footprint & safety. All device arrays use *float32* to reduce bandwidth and footprint.

Blocks & threads. The grid launches n_{elem} blocks (one per element). Each block uses n_{gauss} threads (one per Gauss point), so thread *tx* owns exactly one quadrature point within its element.

Shared-memory accumulators. Element-local tiles (*fe_local*, *Me_local*, *Aqq_local*, *Auq_local*) live in *shared memory*. Per-thread contributions are added via *block-scope atomics*, avoiding slow global atomics and scattered global stores.

Thread 0 responsibilities. *tx==0* zero-initializes shared tiles before use and performs a single *coalesced* flush to global memory using precomputed base offsets. Centralizing these steps in one thread ensures contiguous writes and predictable memory traffic.

Synchronization. Two *syncthreads()* barriers fence the phases: one after initialization (guarantees clean shared tiles) and one after accumulation (guarantees all contributions are visible before flushing). A final barrier is optional for symmetry and debugging convenience.

Boundary-thread guard. Extra threads (if any) are disabled with *if tx < nGauss*, preventing out-of-bounds accesses while still allowing all threads to participate in barriers.

Physical gradients: CPU vs GPU. On the CPU, mapping reference gradients to physical space is best expressed as compact BLAS-like vector operations (Algorithm 2). A few dense vector calls exploit cache locality and tuned libraries, so it is natural to do the arithmetic in large operations.

On the GPU, we change the perspective: *one thread = one Gauss point.* Each thread computes its Jacobian and fills small per-thread arrays *Nx[a]*, *Ny[a]* with simple fused multiply–adds over nodes. Keeping *Nx*, *Ny* in thread-local registers avoids shared/global traffic, and fusing all steps (Jacobian, inverse, gradient transform) into a single per-thread routine minimizes launches and maximizes occupancy.

We use a *1D thread layout* per element with $blockDim.x = n_{\text{Gauss}}$. Thread *tx* owns one Gauss point and selects the corresponding *row* in the reference derivative tables via two-index addressing *[tx,a]*, observable in Algorithm 4 and in Figure 4.1. The first index *tx* fixes the Gauss row (per-thread), while the loop index a runs over nodes.

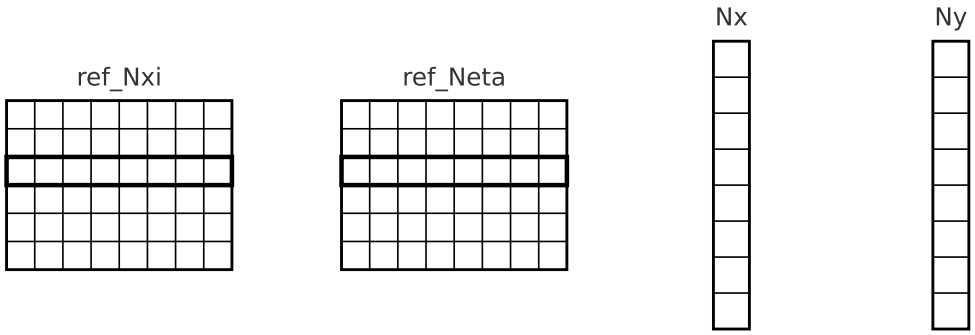


Figure 4.1: *GPU physical gradients diagram*

A 1D block maps the natural parallel grain (Gauss points) to threads without introducing an extra dimension that would require cross-thread reductions:

- simpler indexing (*[tx,a]* vs. *[tx,ty]*)
- no inter-thread coordination to assemble per-point gradients
- fewer barriers and lower register/shared-memory pressure
- straightforward bounds guard (*if tx ¡ nGauss*) with minimal divergence

Using a 2D block (e.g., Gauss×node) would either duplicate work or force extra reductions/atomics to combine per-node contributions, offering no advantage for this stage.

Local assembly: CPU vs GPU. On the CPU, local matrices come from dense products (see Algorithm 3). Here, M_e is duplicated into the two diagonal blocks of A_{qq}. These expressions are compact, cache-friendly, and let BLAS do the heavy part.

On the GPU, the same math is reorganized for the memory hierarchy. Each thread contributes its Gauss point via atomic adds into shared tiles: *fe_local[i]*, *Me_local[i,j]*, *Auq_local[i,2*j]*, *Auq_local[i,2*j+1]*. Shared memory is fast and block-scoped, so these atomics are cheap and avoid global contention.

The access *ref_N[tx,i]* again uses *[tx,i]*: the thread's Gauss point *tx* picks the row; i selects the basis function. The following atomic adds show exactly which entries are updated, represented in Figure 4.2:

- *cuda.atomic.add(fe_local, i, Ni * dvol * src)* updates the i-th entry of the local load vector `fe_local`.
- *cuda.atomic.add(Me_local, (i,j), ...)* updates the (i, j) entry of the scalar mass tile M_e.
- *cuda.atomic.add(Auq_local, (i,2*j), ...)* and *(i,2*j+1)* pack the q_x and q_y couplings side-by-side. Then, column $2j$ stores the q_x contribution of node j and column $2j+1$ stores the q_y one.

This turns the CPU's dense products into per-Gauss incremental updates aggregated safely in shared memory.

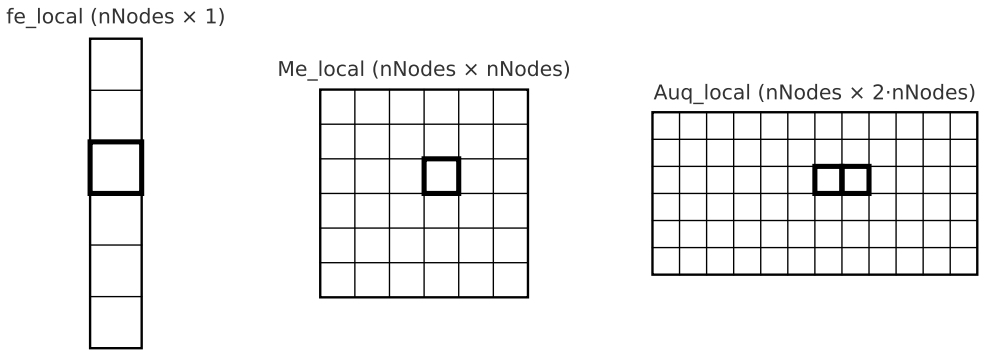


Figure 4.2: *Local assembly diagram*

After a barrier, **thread 0** duplicates *Me_local* into the two diagonal blocks of *Aqq_local* and performs a *single coalesced* flush to global memory using per-element base offsets. This "accumulate locally, write once" pattern is crucial on GPUs to reduce scattered global stores.

Data flow. A final diagram (Figure 4.4) is included in the explanation to visualize the data flow among the different types of memory:

- **Global memory (GPU DRAM):** the block reads the Gauss row *ref_N[tx,*]*.
- **Per-thread registers:** each thread *tx* forms `Ni` and *dvol*.
- **Per-block shared memory:** threads atomically add into *Me_local[i,j]*.
- **Global output (GPU DRAM):** after a barrier, thread 0 duplicates *Me_local* into the q_x/q_y blocks and flushes the result to the element's contiguous slice of *Aqq_data* (coalesced, no global atomics).

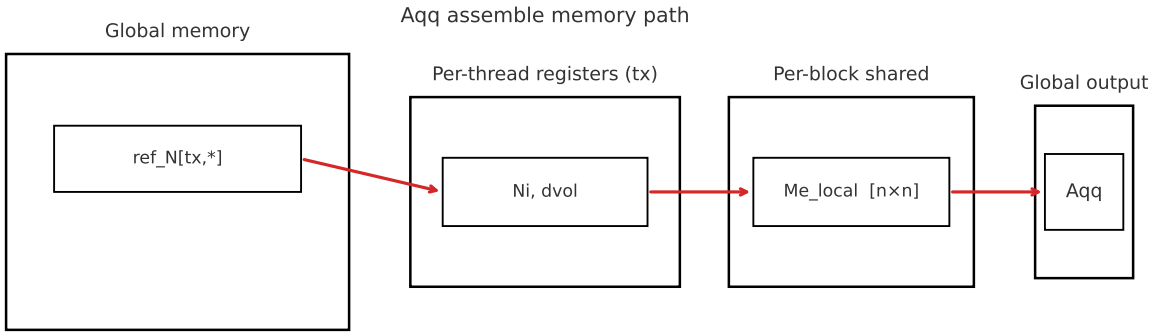


Figure 4.3: *GPU memory flow in the volumetric integration kernel diagram*

Algorithm 2: CPU reference — physical gradients

```
# Given inverse Jacobian entries and reference gradients (Nxi, Neta):
Nx = invJ11 @ Nxi + invJ12 @ Neta
Ny = invJ21 @ Nxi + invJ22 @ Neta
```

Algorithm 3: CPU reference — local assembly

```
Me  = N.T @ (dvolu @ N)              # (nNodes x nNodes)
Aqq = np.zeros((2*nNodes, 2*nNodes))
Aqq[0::2, 0::2] = Me                 # qx block
Aqq[1::2, 1::2] = Me                 # qy block

Auq = np.zeros((nNodes, 2*nNodes))
Auq[:, 0::2] = N.T @ (dvolu @ Nx)    # qx coupling
Auq[:, 1::2] = N.T @ (dvolu @ Ny)    # qy coupling

fe  = N.T @ (dvolu @ src)            # load vector
```

Algorithm 4: Volumetric Integration Kernel pseudocode

```
from numba import cuda, float32

@cuda.jit
def volumetric_integration_kernel(
    Xe, ref_N, ref_Nxi, ref_Neta, ref_IPw,
    muElem, Aqq_data, Auq_data, fe_data
):
    # -- mapping
    e  = cuda.blockIdx.x          # one block per element
    tx = cuda.threadIdx.x         # one thread per Gauss point

    # -- sizes
    nNodes = ref_N.shape[1]
    nGauss = ref_IPw.shape[0]

    # -- shared tiles (per element)
    fe_local  = cuda.shared.array(32, float32)
    Me_local  = cuda.shared.array((32,32), float32)
    Aqq_local = cuda.shared.array((64,64), float32)
    Auq_local = cuda.shared.array((32,64), float32)

    # -- init shared once (thread 0) ...
    if tx == 0:
        for i in range(nNodes):
            fe_local[i] = 0.0
            for j in range(nNodes):
                Me_local[i,j] = 0.0
        # also for Aqq_local and Auq_local
    cuda.syncthreads()

    # -- per-Gauss work (guard boundary threads)
    if tx < nGauss:
        # (1) build Jacobian & inverse ...
        ...

        # physical gradients
        Nx = cuda.local.array(32, float32)
        Ny = cuda.local.array(32, float32)
        for a in range(nNodes):
            Nx[a] = invJ11*ref_Nxi[tx,a] + invJ12*ref_Neta[tx,a]
            Ny[a] = invJ21*ref_Nxi[tx,a] + invJ22*ref_Neta[tx,a]

        # (2) interpolate (xg, yg), compute src & dvol ...
        ...

        # local assembly
        for i in range(nNodes):
            Ni = ref_N[tx,i]
            cuda.atomic.add(fe_local, i, Ni * dvol * src)
            for j in range(nNodes):
                Nj = ref_N[tx,j]
                cuda.atomic.add(Me_local, (i,j),      Ni * dvol * Nj)
                cuda.atomic.add(Auq_local,(i,2*j),     Ni * dvol * Nx[j])
                cuda.atomic.add(Auq_local,(i,2*j+1),   Ni * dvol * Ny[j])

    cuda.syncthreads()

    # -- finalize & flush once (thread 0)
    if tx == 0:
        baseA = e*(2*nNodes)*(2*nNodes)
        for i in range(nNodes):
            for j in range(nNodes):
                m = Me_local[i,j]
                Aqq_local[2*i,2*j]     = m
                Aqq_local[2*i+1,2*j+1] = m
        for i in range(2*nNodes):
            for j in range(2*nNodes):
                Aqq_data[baseA + i*2*nNodes + j] = Aqq_local[i,j]

        # also for Auq_data and fe_data
    cuda.syncthreads()
```

4.2.1.2 Local dense solve on GPU

All dense on device. Local HDG blocks are reshaped and concatenated as *CuPy* arrays on the GPU. Operations like ttranspose and *concatenate* are device-side, avoiding host transfers and producing contiguous, coalesced layouts per element.

Shape of matrices. We form $A_{qu} = -\mu\, A_{uq}^{\top}$ and $A_{ql} = +\mu\, A_{lq}^{\top}$ on the device, matching the Poisson symmetry while keeping single precision (*float32*) consistent with the CUDA kernels.

Solve. *cp.linalg.solve* solves *one system per element* in batch using CuPy dense routines (cuSOLVER) [6]. This eliminates the need to write a custom kernel for small dense factorizations.

But, Why CuPy instead of a kernel? Small, dense linear solves are better handled by tuned libraries than by CUDA code: less code, fewer synchronization hazards, and typically higher throughput due to batched LU/solve implementations. We also keep the entire pipeline on GPU memory, minimizing latency.

Outputs of this stage. We obtain four device arrays: *UU_dev* and *QQ_dev* (interior responses to unit face dofs), and *Uf_dev*, *Qf_dev* (particular solutions due to the source). These feed the subsequent global assembly (COO) stage.

Algorithm 5: Local Schur solve with CuPy pseudocode

```
# --- Dense blocks on GPU (CuPy) ---
Auu3 = Auu.reshape(nElem, nNodes, nNodes)
Auq3 = ...  # reshape to (nElem, nNodes, 2*nNodes)
Aqq3 = ...  # reshape to (nElem, 2*nNodes, 2*nNodes)

# Poisson-consistent transposes/scalings
Aqu  = -mu_d[:,None,None] * Auq3.transpose(0,2,1)
Aul  = -Alu.reshape(nElem, Nloc, nNodes).transpose(0,2,1)
Aql  =  mu_d[:,None,None] * Alq.reshape(nElem, Nloc, 2*nNodes).transpose(0,2,1)

# Big per-element system Abig (3n x 3n)
top  = cp.concatenate([Auu3, Auq3], axis=2)
bot  = cp.concatenate([Aqu,  Aqq3], axis=2)
Abig = cp.concatenate([top,  bot ], axis=1)

# Right-hand sides
rhsU = cp.concatenate([Aul, Aql], axis=1)     # shape (nElem, 3n, Nloc)
rhsf = ...  # stack fe and zeros to shape (nElem, 3n)

# Batched dense solves on GPU
UQ = -cp.linalg.solve(Abig, rhsU)  # (nElem, 3n, Nloc)
fU =  cp.linalg.solve(Abig, rhsf)  # (nElem, 3n)

# Partition results (examples; others analogous)
UQ_big = UQ.reshape(nElem, 3*nNodes, Nloc)
UU_dev = UQ_big[:, :nNodes, :].copy()
QQ_dev = UQ_big[:,  nNodes:, :].copy()
# Uf_dev, Qf_dev obtained analogously from fU
```

4.2.1.3 Kernel: Flip face orientation and build COO triplets and RHS tuples

Algorithm 6: flip_and_build_coo_kernel pseudocode

```
@cuda.jit
def flip_and_build_coo_kernel(
    F_dev, intFaces_dev, nFaceNodes, nIntFaces, nNodes, Nloc,
    Alq_data, Alu_data, All_data, UQ_data, fU_data,
    coo_rows, coo_cols, coo_vals,  # matrix COO triplets
    f_rows, f_vals,                # vector entries
    debug_e
):
    e0 = cuda.blockIdx.x
    e  = debug_e if debug_e >= 0 else e0
    tx = cuda.threadIdx.x
    if e < 0: return

    dimU = nNodes + 2*nNodes  # 3*nNodes

    # ---- (1) Face-orientation FLIP (permute only; no signs) ----
    flip = cuda.local.array(3, int32)
    for iface in range(3):
        flip[iface] = 0
        fidx = F_dev[e, iface]
        if fidx < nIntFaces:
            rE = intFaces_dev[fidx, 2]     # "right" element for that interior face
            if e == rE:
                flip[iface] = 1            # reverse local order on this face

    # ---- (2) Matrix entries: 1 thread -> 1 (irow,jcol) ----
    if tx < Nloc*Nloc:
        irow = tx // Nloc;  jcol = tx % Nloc

        iface_i = irow // nFaceNodes;  pos_i = irow - iface_i*nFaceNodes
        iface_j = jcol // nFaceNodes;  pos_j = jcol - iface_j*nFaceNodes

        # flip local indices if this element is the "right" neighbor
        iL = iface_i*nFaceNodes + (nFaceNodes-1-pos_i) if flip[iface_i] == 1 else irow
        jL = iface_j*nFaceNodes + (nFaceNodes-1-pos_j) if flip[iface_j] == 1 else jcol

        # global rows/cols (no flip; orientation handled in iL/jL)
        fidx_i = F_dev[e, iface_i];  fidx_j = F_dev[e, iface_j]
        row = fidx_i*nFaceNodes + pos_i
        col = fidx_j*nFaceNodes + pos_j

        # local Schur contribution sK = All + Alu*UQ_u + Alq*UQ_q  (omitted loops)
        sK = 0.0
        # ... accumulate using Alq_data/Alu_data/All_data and UQ_data at (e, iL, jL) ...

        # COO write (race-free): unique slot per (e,irow,jcol)
        out_idx = e*(Nloc*Nloc) + irow*Nloc + jcol
        coo_rows[out_idx] = row
        coo_cols[out_idx] = col
        coo_vals[out_idx] = sK

    # ---- (3) Vector entries: first Nloc threads ----
    if tx < Nloc:
        irow = tx
        iface_i = irow // nFaceNodes;  pos_i = irow - iface_i*nFaceNodes
        rowL = iface_i*nFaceNodes + (nFaceNodes-1-pos_i) if flip[iface_i] == 1 else irow

        # sf = Alu*fU_u + Alq*fU_q  (omitted loops)
        sf = 0.0
        # ...

        out_f = e*Nloc + irow
        f_rows[out_f] = F_dev[e, iface_i]*nFaceNodes + pos_i
        f_vals[out_f] = -sf
```

Face flip (orientation). Each interior face has two owners, their local node orders are opposite. We detect if the current element is the designated right neighbor of that face. If so, we reverse the local face index when reading element-local blocks (iL, jL).

The global *row*/*col* mapping uses the unflipped (*pos_i*, *pos_j*); only the position inside the element blocks is flipped. This aligns both elements' contributions for the shared face without introducing sign changes.

COO assembly on GPU. First of all, the strategy of *One thread = one entry, zero races* is followed: no sums are done at the same time, so we avoid synchronization among threads. Each thread writes a single triplet (*row*, *col*, *val*) into a preassigned, unique slot: $out\_idx = e \cdot N_{\text{loc}}^2 + i_{\text{row}} \cdot N_{\text{loc}} + j_{\text{col}}$. No two threads ever collide, so *no global atomics are needed*.

We **allow duplicates** across neighboring elements (both contribute to the same global entry) and leave the sum task to a COO→CSR conversion (handled outside the kernel). Regarding the RHS reduction, *cp.bincount* performs a parallel histogram with weights to sum contributions that target the same global row, yielding *fgpu*:

```
# COO -> CSR (duplicates are summed automatically)
K_coo = cpsp.coo_matrix((coo_vals, (coo_rows, coo_cols)),
                        shape=(nGlobalDofs, nGlobalDofs))
K_gpu = K_coo.tocsr()

# RHS vector: reduction by row (sum repeated rows)
f_gpu = cp.bincount(f_rows, weights=f_vals, minlength=nGlobalDofs).astype(dtype)
```

There are other techniques in literature [27] like atomic adding or graph coloring. On the one hand, memory traffic is far more regular than random atomic updates into a global CSR structure, which is complex. On the other hand, graph coloring removes write conflicts by splitting work into multiple color passes (extra preprocessing, multiple kernel launches, potential load imbalance) [20].

Therefore, the use of COO assembly is justified, being the main cost the temporary storage for triplets and the CSR conversion step, which is typically small compared to the savings from eliminating global conflicts. Furthermore, one could see the visual explanation of how duplicates are handled in the COO→CSR conversion.

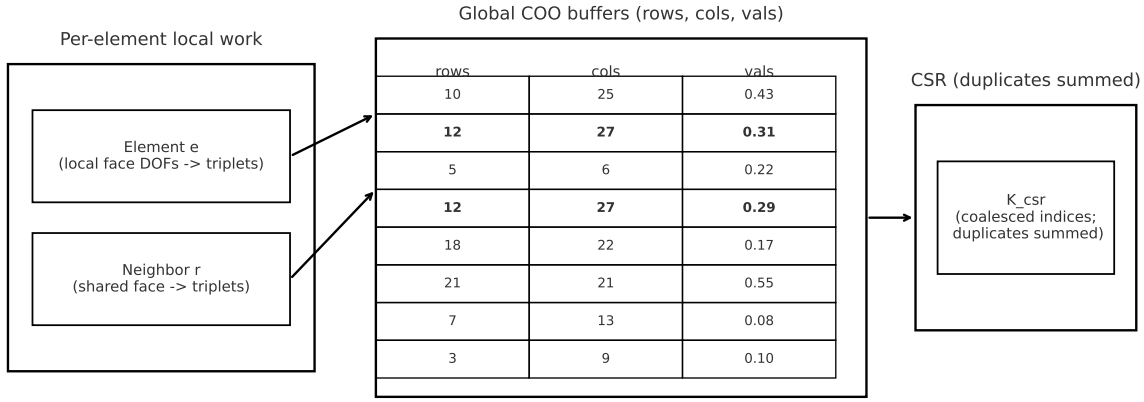


Figure 4.4: *GPU COO and CSR data conversion diagram. The COO→CSR conversion sorts indices and* adds *repeated* (i, j) *entries, producing the correct sparse matrix* K_{gpu} *with no global atomics or graph coloring.*

4.2.1.4 Development Challenges and Solutions

During the GPU implementation of the HDG Poisson assembly we faced several practical issues. This section records the main problems and the design choices that resolved them.

Monolithic kernel. Our first attempt packed volume/face integration, local Schur solve, and global assembly into a single mega–kernel. That design suffered from high register pressure and spills, oversized shared–memory tiles (different sizes for volume vs. face work), warp divergence from mixing Gauss loops with face loops, and the impossibility of inter–block synchronization needed for safe global assembly. The result was low occupancy, fragile code for higher polynomial orders, and poor performance. **Solution**. Each stage now fits comfortably in registers/shared memory, launches with a natural 1D thread layout, and avoids cross–concerns (no local solve inside integration, no global atomics during assembly). Furthermore, occupancy improves, and the code is easier to debug.

Race conditions in the global assembly. Early prototypes let each element kernel add directly into the global sparse matrix. On shared faces, two neighboring elements attempted to update the same entry simultaneously, creating race conditions. Global floating–point atomics removed the races but introduced severe serialization overhead. **Solution.** We switched to a two–phase strategy in *COO* format: each element writes its own triplets (row, col, val) into a private slice of preallocated arrays; duplicates are *intended* and later summed during the COO→CSR conversion. This eliminated inter–block synchronization and avoided global atomics entirely while keeping writes contiguous and predictable.

Memory bandwidth and atomic overhead. Accumulating Gauss–point contributions directly in global memory produced many uncoalesced stores and atomics, causing a slow performance, and problems with bandwidth. Apart from performance problems, we also had memory consumption errors, especifically we had large per-thread local arrays inside the kernel. Since their indices were not compile-time constants, the compiler placed them in *local memory* (which is actually device

DRAM). Scaled by device occupancy, these per-thread buffers exceeded the GPU's total memory and triggered a *CUDA_OUT_OF_MEMORY* error. **Solution.** We introduced per–block shared tiles (*fe_local*, *Me_local*, *Auq_local*, *Aqq_local*). Threads accumulate with cheap block–scope atomics, then a single thread flushes the final tiles to global memory once per element. With careful sizing (sufficient for the target polynomial orders), this moved the hot path to on–chip memory and reduced global traffic dramatically.

Face–orientation mismatch. Two elements see a common interior face with opposite local orderings. Without correction, contributions are misaligned across the interface. **Solution.** A lightweight *flip* rule chooses a canonical orientation per interior face. If the current element is the designated "right" owner, its local face indices are reversed for reads/writes against local blocks; global (*row*, *col*) indices remain unflipped. This guarantees consistent assembly on shared faces.

Precision and memory usage. Double precision doubles memory and is often slower on GPUs with limited FP64 throughput. **Solution.** We assembled in single precision (*float32*), matching the CUDA kernels and maximizing throughput. Where needed, later stages (e.g., final solves) can be promoted to FP64 without changing the assembly path.

Thread and kernel configuration. We explored "one thread per element" (underutilizes the GPU) and "one thread per matrix entry" (complex integration loops). **Solution.** A hybrid mapping works best: one block per element, with threads mapped either to Gauss points (integration kernels) or to face–matrix positions (global assembly). Thread counts match the work per element (e.g., n_{Gauss} or N_{loc}^2), avoiding idle lanes and excess synchronization.

4.2.2 Compute projection faces kernel

The special aspect is that this kernel builds and solves (in shared memory) a tiny face-local system for boundary projection, only for exterior faces, with a 1D Gauss mapping, and then writes the face nodal values as output. Firstly, we assemble and solve a small 1D face system $M_f\, c = b_f$ only when $\mathit{extFaceIdx} \geq 0$; interior faces are skipped.

Then, the face mass matrix *sm_M* and RHS *sm_b* live in shared memory; many threads build them (Gauss loop), then a single thread (t=0) runs a short *Gaussian elimination (forward + back substitution)* without pivoting. We do not require an special CUDA kernel because this matrix M_f is symmetric positive definite (all eigenvalues of A are non-negative).

Algorithm 7: Face projection pseudocode

```
@cuda.jit
def project_faces_elem_kernel(...):
    e = cuda.blockIdx.x; t = cuda.threadIdx.x

    # Shared tiles: one tiny system per local face (3 faces)
    sm_M = cuda.shared.array((3, 32, 32), float32)  # face mass
    sm_b = cuda.shared.array((3, 32),      float32)  # RHS
    if t == 0:                        # zero once per block
        for f in range(3):
            for i in range(nFaceNodes): sm_b[f,i] = 0.0;
                for j in range(nFaceNodes): sm_M[f,i,j] = 0.0
    cuda.syncthreads()

    # (A) Assemble ONLY exterior faces (Gauss loop -> shared atomics)
    if t < ngf:
        for f in range(3):
            ext = extFaceIdx[e,f]
            if ext < 0: continue
            # ... compute w, ug ...
            for i in range(nFaceNodes):
                Ni = N1d[t,i]
                for j in range(nFaceNodes):
                    cuda.atomic.add(sm_M, (f,i,j), Ni * N1d[t,j] * w)
                cuda.atomic.add(sm_b, (f,i), Ni * ug * w)
    cuda.syncthreads()

    # (B) In-kernel Gaussian elimination (t==0) per exterior face
    if t == 0:
        for f in range(3):
            ext = extFaceIdx[e,f]
            if ext < 0: continue
            # forward elimination
            for k in range(nFaceNodes):
                inv = 1.0 / sm_M[f,k,k]
                for j in range(k+1, nFaceNodes):
                    fac = sm_M[f,j,k] * inv
                    for l in range(k, nFaceNodes):
                        sm_M[f,j,l] -= fac * sm_M[f,k,l]
                    sm_b[f,j] -= fac * sm_b[f,k]
            # back substitution + write
            base = ext * nFaceNodes
            for i in range(nFaceNodes-1, -1, -1):
                acc = sm_b[f,i]
                for j in range(i+1, nFaceNodes):
                    acc -= sm_M[f,i,j] * sm_b[f,j]
                sm_b[f,i] = acc / sm_M[f,i,i]
                u_out[base + i] = sm_b[f,i]
```

4.2.3 Global system solve kernel

Global solver (GPU). *cupyx.scipy.sparse.linalg.cg*, to solve the condensed HDG trace system $K\lambda = f$ entirely on the GPU. For $\tau > 0$, the matrix K is symmetric positive definite (SPD), making CG appropriate and efficient. Each iteration performs one CSR SpMV via cuSPARSE and a small set of BLAS-1 operations (dot, norm, axpy) via cuBLAS. Notice that this library–driven design follows prior GPU implementations (e.g., Wang et al [21]).

4.2.4 Elements solution kernel

This kernel performs the HDG reconstruction of element–interior fields directly on the GPU. For each element it computes

$$u_e = UU_e\,\lambda + Uf_e, \qquad q_e = QQ_e\,\lambda + Qf_e,$$

using the precomputed local response tensors UU_e, QQ_e and the particular terms Uf_e, Qf_e. The launch strategy differs from the earlier integration/assembly kernels: *one block per element* and $2\,n_{\text{elemNodes}}$ threads per block. Threads with $t < n_{\text{elemNodes}}$ accumulate the scalar field u; threads with $t < 2\,n_{\text{elemNodes}}$ accumulate the flux q (Algorithm 8). There is no shared memory and no atomics—partial sums stay in registers—and each thread performs a single coalesced write to *u_out* or *q_out*. Compared with the previous kernels (Gauss integration, shared-memory tiling, COO assembly), this stage is a pure local mat–vec on device: lightweight, deterministic, and bandwidth-friendly, with only a trivial reshape of q at the end.

Algorithm 8: Element reconstruction pseudocode

```
@cuda.jit
def compute_elem_solution_kernel(lambda_d, UU_d, QQ_d, Uf_d, Qf_d, F_d,
                                 u_out, q_out, nFaceNodes, nElemNodes):
    e = cuda.blockIdx.x; t = cuda.threadIdx.x   # 1 block = 1 element

    # u: threads 0..nElemNodes-1
    if t < nElemNodes:
        acc = 0.0
        for fi in range(3):                     # three faces
            face = F_d[e, fi]; base = face * nFaceNodes
            for j in range(nFaceNodes):         # gather lambda on face
                acc += UU_d[e, t, fi*nFaceNodes + j] * lambda_d[base + j]
        u_out[e*nElemNodes + t] = acc + Uf_d[e, t]

    # q: threads 0..(2*nElemNodes-1)
    if t < 2*nElemNodes:
        acc = 0.0
        for fi in range(3):
            face = F_d[e, fi]; base = face * nFaceNodes
            for j in range(nFaceNodes):
                acc += QQ_d[e, t, fi*nFaceNodes + j] * lambda_d[base + j]
        q_out[e*2*nElemNodes + t] = acc + Qf_d[e, t]
```

4.2.5 HDG Postprocess

Unlike the integration/assembly kernels that tile in shared memory and emit COO triplets, the postprocess kernel builds, *per element*, the dense operators for the "star" reconstruction directly on device: $K_e[i,j] = \int \nabla N_i \cdot \nabla N_j$, $Bq_e[i] = \int \nabla N_i \cdot q$, $Ius_e[i] = \int N_i$, and the scalar $Iu_e = \int u$. We switch the thread mapping to *one block per element, threads over basis rows* ($t < i < n_{\text{pts}}$), while each thread runs the *entire Gauss loop* for its row. This eliminates large shared tiles for a dense K_e and keeps all per–row accumulations in registers.

Row-wise contributions to K_e, Bq_e, Ius_e are aggregated with *element-scoped global atomics*; since blocks never collide across elements, no inter-block synchronization is needed. The scalar integral Iu_e is collected by *t==0* only, avoiding atomics entirely for that quantity. To improve the accuracy of the postprocess solution, the kernel and its outputs run in *float64*, while the earlier assembly path used float32. After the kernel, we form the small augmented system per element and call a *batched dense solve* (*cp.linalg.solve*) on the GPU to obtain the postprocessed field $u_\star$.

Algorithm 9: HDG postprocess pseudocode

```
@cuda.jit
def hdg_postprocess_kernel(Xe, N, Nxi, Neta, dNdxi, dNdeta, IPw,
                           ue, qx, qy, K_out, Bq_out, Ius_out, Iu_out):
    e = cuda.blockIdx.x; t = cuda.threadIdx.x
    # one block = one element; threads index basis rows

    acc_Iu = 0.0  # only t==0 updates this (no atomics)

    for g in range(ngauss):
        # Jacobian, det, inverse, dvol
        # ... (assemble J from dNdxi/dNdeta and Xe) ...
        # interpolate u_g, qx_g, qy_g at Gauss point
        # ...

        if t == 0:
            acc_Iu += u_g * dvol

        if t < npts:
            # gradient of row-basis Ni in physical coords
            Nx = invJ11*Nxi[g,t] + invJ12*Neta[g,t]
            Ny = invJ21*Nxi[g,t] + invJ22*Neta[g,t]

            # element-scoped global atomics (dense row i)
            cuda.atomic.add(Bq_out,  (e, t), (Nx*qx_g + Ny*qy_g) * dvol)
            cuda.atomic.add(Ius_out, (e, t),  N[g,t] * dvol)
            for j in range(npts):
                Nxj = invJ11*Nxi[g,j] + invJ12*Neta[g,j]
                Nyj = invJ21*Nxi[g,j] + invJ22*Neta[g,j]
                cuda.atomic.add(K_out, (e, t, j), (Nx*Nxj + Ny*Nyj) * dvol)

    if t == 0:
        Iu_out[e] = acc_Iu
```

4.2.6 L2 error kernels (solution field and postprocessed solution field)

Both routines avoid custom kernels and compute the L2 norm entirely with vectorized CuPy primitives. In both cases the design is the same: no Python loops, no atomics, minimal memory traffic, and a single host transfer at the very end. No further explanation is given due to the simplicity of these kernels.

4.3 Hardware summary and implications.

Table 4.1 summarizes the host CPU used for the MATLAB and Python–CPU baselines. The system runs Windows 10 on an *Intel® Core™ i7-1165G7* (reported as 8 logical and 8 physical cores) at a base/max clock of 2.80 GHz, with only ≈ 4.5 GB of RAM available to MATLAB/Python in the measured sessions. Dense linear algebra is backed by Intel oneAPI MKL (BLAS/LAPACK 2023.2), which is advantageous for dense kernels but less impactful for sparse, memory–bound operations [13].

Table 4.2 reports the characteristics of the *TUCANGPU* device used for the Python GPU runs. It is an NVIDIA TITAN V (compute capability 7.0) with 80 streaming multiprocessors (SMs), 64 FP32 cores per SM (total 5120 cores), a core clock of 1.455 GHz, and ≈ 11.77 GB of device memory. The memory subsystem is notably wide (bus 3072 bits, memory clock 0.85 GHz), yielding a peak bandwidth of 652.8 GB/s. The quoted peak single–precision throughput is 14 899.2 GF/s (14.9 TF/s). From these numbers the roofline crossover intensity is

$$OI_{\times} = \frac{\text{Peak GF/s}}{\text{Peak GB/s}} \approx \frac{14899.2}{652.8} \approx 22.8 \text{ FLOP/Byte},$$

so kernels with $OI \ll 22.8$ will be memory-bound on this GPU [29].

Table 4.1: *Host CPU characteristics gathered from the MATLAB/Python CPU environment.*

Characteristic	**Value**
Operating system	Microsoft Windows [Versión 10.0.19045.6093]
Machine (MATLAB `computer`)	`PCWIN64`
Hostname	`LAPTOP-5CSAJO2M`
CPU model	11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
Logical cores	8
Physical cores	8
Base/Max clock (GHz)	2.80
System memory (GB)	4.5
MATLAB version	24.1.0.2689473 (R2024a) Update 6
BLAS	Intel(R) 2023.2
LAPACK	Intel(R) 2023.2
Python version (CPU exp.)	—

Table 4.2: *Characteristics of the GPU platform gathered from the Python–GPU run (values rounded for readability).*

Characteristic	**Value**
GPU model	NVIDIA TITAN V
Compute capability	7.0
Device memory (GB)	11.77
Core clock (GHz)	1.455
Memory clock (GHz)	0.85
Memory bus width (bits)	3072
Memory bandwidth (GB/s)	652.8
Multiprocessors (SMs)	80
CUDA cores (total)	5120
Cores per SM	64
Peak FP32 performance (GFLOPS)	14 899.2
Peak FP64 performance (GFLOPS)	—

The TITAN V's bandwidth (652.8 GB/s) and massive thread parallelism (80 SMs, 5120 FP32 cores) make it well suited to the phases that dominate our code: sparse matrix–vector products (CG) and elementwise streaming kernels (local reconstruction, postprocess, error evaluation). Because their operational intensity is well below $OI_{\times}$, performance is limited by data movement, not peak GFLOP/s [29], which is exactly what the roofline plots will show in the next chapter.

On the CPU, the modest available memory and the nature of sparse kernels (poor cache reuse, irregular access) constrain throughput. MKL accelerates dense blocks, but the end-to-end HDG pipeline remains memory-bound [13].

4.4 Methodology for performance metrics and roofline analysis

Our goal is to compare the three implementations (MATLAB–CPU, Python–CPU, Python–GPU) with hardware–meaningful metrics and to place each kernel on a roofline chart. Instead of instrumenting every floating-point operation inside the kernels (which would be invasive and brittle), we estimate the operation counts from the algebraic sizes that the method dictates, and we measure time precisely on the device. From those two ingredients we derive sustained throughput and memory traffic.

From the measured device time t, we report sustained throughput, bandwidth, and intensity using the standard relations:

$$\text{GFLOP/s} = \frac{\text{FLOPs}}{t}, \qquad \text{GB/s} = \frac{\text{Bytes}}{t}, \qquad \text{OI} = \frac{\text{FLOPs}}{\text{Bytes}} \ \text{[FLOP/Byte]},$$

where FLOPs and Bytes come from the algebraic sizes and data layouts touched by each kernel. We then interpret these points in a Roofline chart as in [22], contrasting the observed GFLOP/s against the bandwidth roof $\text{BW}_{\text{peak}} \cdot \text{OI}$ and the compute roof FP_{peak}.

Our "bytes moved" model counts values, indices/coordinates, and vector/matrix operands read and written by a kernel. This effective bandwidth may differ from exact DRAM traffic if caches reuse part of the input (e.g., repeated vector reads), but—consistent with [22], it remains a faithful proxy for bandwidth utilization because caching amplifies bandwidth without reducing latency. The goal is to compare kernels and sizes consistently, not to reconstruct the microarchitectural bus traffic.

Finally, unless explicitly stated, we exclude host↔device transfers from kernel timings to isolate compute and memory behavior on the device; when transfers are part of the workflow (e.g., for setup or diagnostics), we time them separately. In iterative contexts, transfers are typically amortized over many iterations and are not the bottleneck for the steady-state inner kernels [22].

CHAPTER 5

Results

Contents

The previous chapters introduced the Hybridizable Discontinuous Galerkin (HDG) formulation for the Poisson problem and outlined the GPU-oriented implementation strategy. With that background in place, the reader is prepared to follow the results and the discussion that follows.

This chapter reports results for the three implementations developed in this work: *MATLAB–CPU*, *Python–CPU*, and *Python–GPU*. We compare them in terms of solution accuracy and computational cost, and we examine in detail the Python–GPU runs to identify the stages that most affect performance and where further optimization is warranted. The overarching aim is to provide clear evidence of when and why a GPU implementation delivers benefits for this HDG Poisson solver.

5.1 HDG advantage over DG

First of all, it is highly important to highlight the advantages of using HDG methods over DG ones, and one way to prove it is to check the number of degrees of freedom for each problem case.

Regarding the counts for triangular elements [35], for a scalar field with P_k polynomials on each triangle, the number of local DG basis functions is

$$n_{\text{loc}}^{\text{DG}}(k) = \frac{(k+1)(k+2)}{2}. \tag{5.1}$$

Thus,

$$\text{DG DoFs} = N_{\text{elem}} \, \frac{(k+1)(k+2)}{2}. \tag{5.2}$$

In HDG, after static condensation only the trace on edges remains globally coupled. With degree-k traces, each edge carries $(k+1)$ unknowns. Dirichlet edges are prescribed and do not contribute; interior edges do. Hence,

$$\text{HDG DoFs} = \left(N_{\text{edges,int}} + N_{\text{edges,Neu}}\right)(k+1). \tag{5.3}$$

(For vector fields, multiply both formulas by the number of components. The ratio is unchanged.) Figure 5.1 reports the ratio DG DoFs/HDG DoFs versus the polynomial degree k for meshes of increasing resolution (mesh1 to mesh7). The ratio grows almost linearly with k and shifts upward for finer meshes. This reflects that DG couples *volume* unknowns while HDG couples only *trace* unknowns on edges. As the mesh is refined the fraction of boundary edges becomes negligible, so the HDG trace system scales more favorably. In practice, once the mesh is moderately large, HDG already has fewer global unknowns for any $k > 0$, and the advantage strengthens with k.

Regarding the linear growth of the ratio, on large triangular meshes one has $N_{\text{edges,int}} \approx \frac{3}{2} N_{\text{elem}}$. Substituting into the previous expressions yields the rule of thumb:

$$\frac{\text{DG}}{\text{HDG}} \approx \frac{(k+1)(k+2)/2}{(k+1)\cdot(3/2)} = \frac{k+2}{3}, \tag{5.4}$$

which is linear in k and consistent with the nearly parallel lines in the plot. Coarser meshes deviate downward because boundary edges constitute a larger fraction of all edges. Overall, the figure confirms that HDG benefits from higher orders and finer meshes, yielding fewer global DoFs, and typically a sparser global matrix than DG.

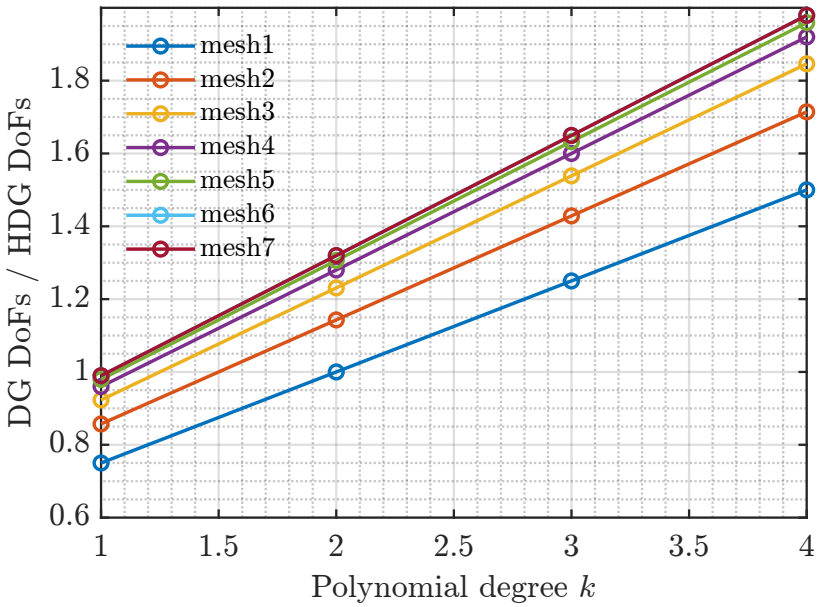


Figure 5.1: *Comparison of DG to HDG DoFs.*

5.2 Comparison of the implementation in MATLAB CPU, Python CPU, and Python GPU.

In this section, we compare the three implementations developed in this work— *MATLAB–CPU*, *Python–CPU*, and *Python–GPU*—using the same set of benchmark meshes (*mesh1* to *mesh7*) and polynomial degrees $k \in \{1, 2, 3, 4\}$. All runs solve the same manufactured Poisson problem with Dirichlet boundary conditions, so that accuracy can be verified through the L^2 error of the HDG solution u and its postprocessed counterpart $u^\star$. For timing, CPU codes use high–resolution wall clocks, while the GPU code uses CUDA events for kernel intervals and a dedicated counter for host

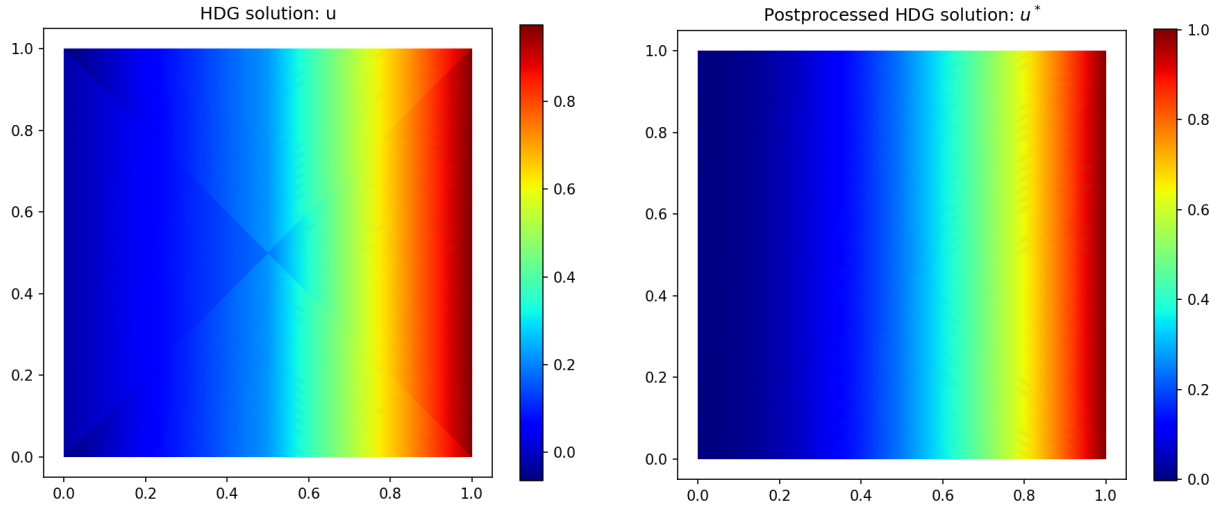
device transfers. Notice that the global time is then partitioned into *preprocess*,*assembly*, *Dirichlet projection/reduction*, *global solve (CG)*, *local element solver*, *postprocess*, and *transfers* between CPU and GPU.

The goal is to confirm that all implementations deliver comparable accuracy, and identify the regimes (mesh size, degree) where the GPU realization provides clear reductions in computational cost.

5.2.1 HDG solutions for the Poisson problem

Figure 5.2 compares the **scalar field** computed with HDG for a manufactured Poisson problem (Dirichlet data). Because the exact solution varies mainly along the x–direction, the maps show almost vertical color bands from the left to the right boundary, consistent with the imposed boundary condition.

Figure 5.4a plots the raw HDG solution u. Being piecewise polynomial, u can exhibit inaccuracies across interfaces, resulting in a visible X shape. On the other hand, Figure 5.4b shows the postprocessed field $u^\star$, obtained by the standard HDG local reconstruction in a higher-order space. This reconstruction smooths inter–element interfaces and typically improves the convergence rate from about k+1 for u to about k+2 for $u^\star$ in smooth cases, which reflects the visual disappearance of the X shape compared to Figure 5.2. This also explains the smaller L^2 errors observed in the results (reflected in Figure 5.4).



(a) HDG solution

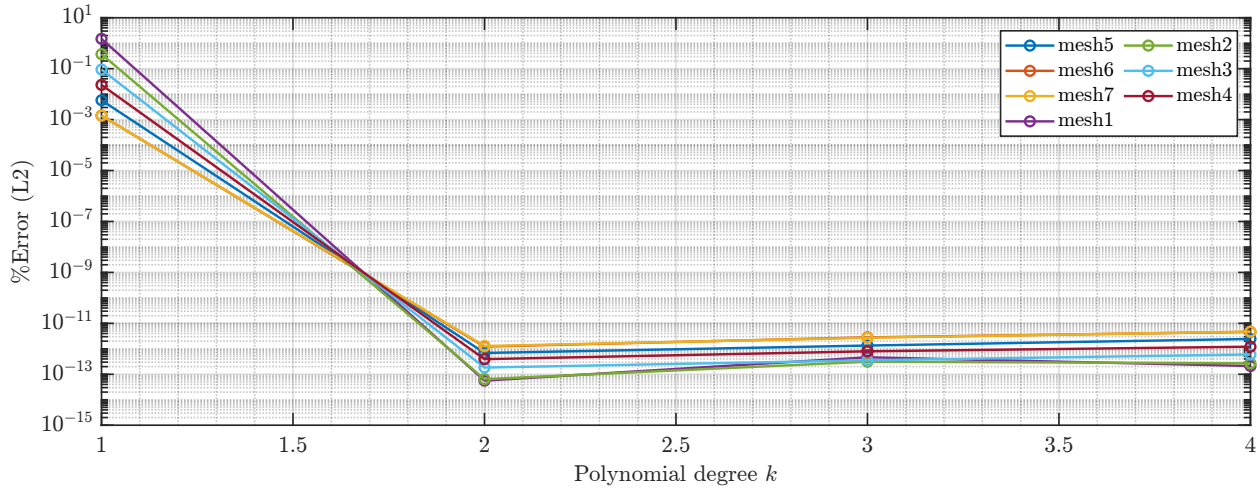
(b) HDG superconvergent postprocessed solution

Figure 5.2: *HDG solution and superconvergent postprocessed solution for the Poisson test case mesh 1,* $k = 1$.

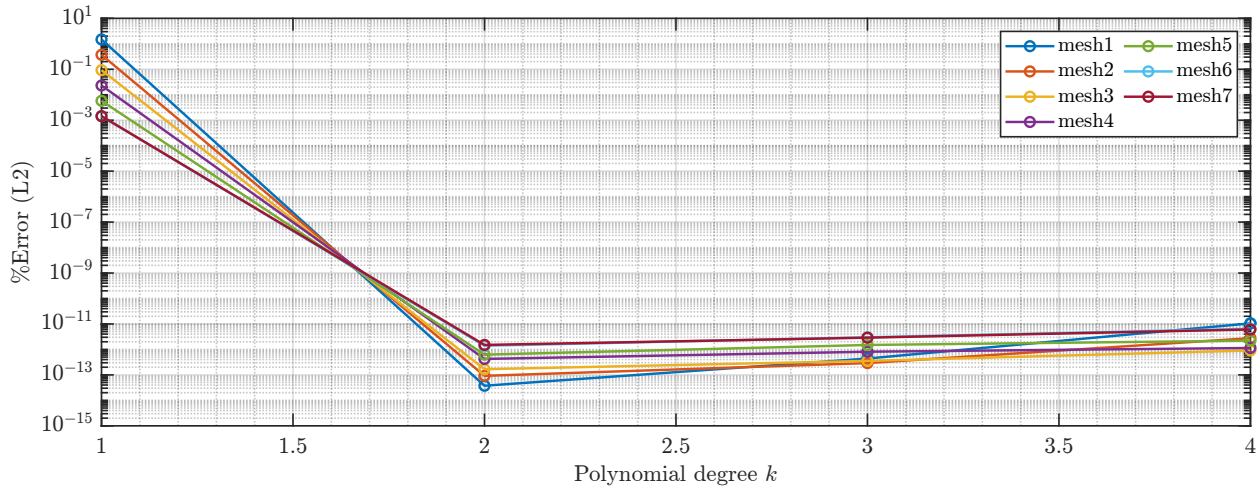
To account for the accuracy of the solution, Figure 5.3 compares the $\%L^2$ error versus polynomial degree k for the *same* manufactured Poisson problem, using our three implementations: (a) *MATLAB CPU*, (b) *Python CPU*, and (c) *Python GPU*.

The two CPU implementations deliver virtually the same accuracy for every mesh and degree, while the Python–GPU curves lie slightly higher. Regarding mesh dependence, at $k = 1$ the meshes are clearly separated (coarser meshes yield larger errors), but from $k \geq 2$ the curves group together: the error drops by orders of magnitude and then levels off as k increases. This reduction is less steep in the Python–GPU case, consistent with the use of single precision (*float32*). An interesting trend is that the mesh with the largest error at $k = 1$ often becomes one of the most accurate once k is increased, reflecting the shift from a mesh-dominated to a p-dominated regime.

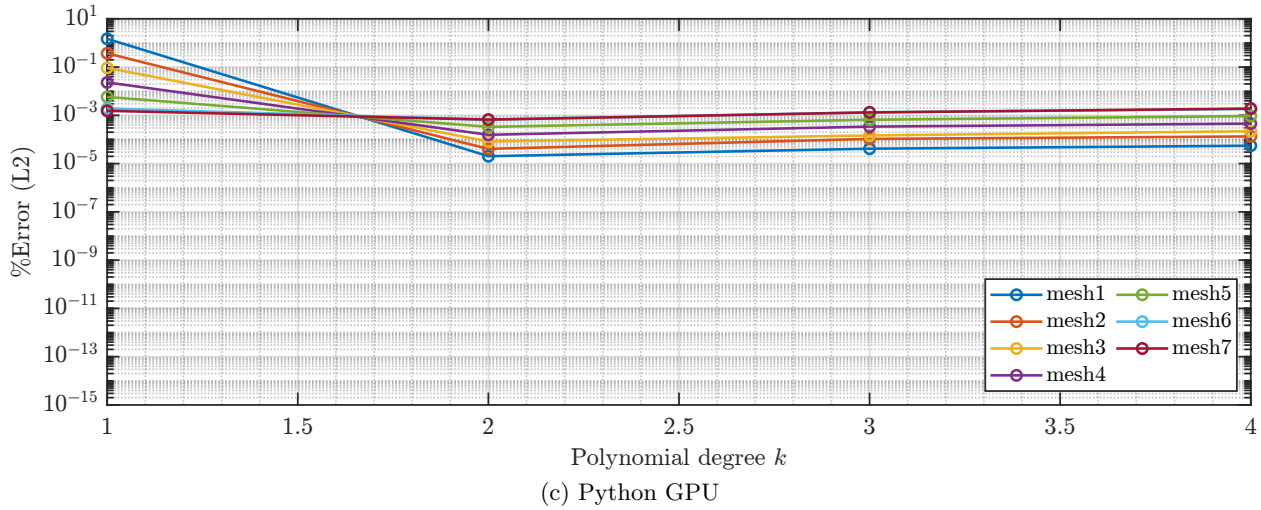
Figure 5.4 shows the L^2 errors obtained in both normal and superconvergent solutions among all meshes and polynomial degrees for MATLAB CPU and Python CPU implementations. One can see that across all meshes, both implementations exhibit the expected HDG behavior: the L_2 error decreases rapidly as the polynomial degree k increases, and the postprocessed solution u^* is consistently more accurate than the raw HDG solution u. The u^*–u gap is most visible for low k, then narrows as both curves approach numerical floors on the finest cases. Trends and magnitudes match closely between MATLAB–CPU and Python–CPU (even though is not represented). On the other hand, the GPU path preserves accuracy, (excepting higher k) and the superconvergent properties of the HDG postprocess, while enabling faster runs, as will be seen in future sections.



(a) MATLAB CPU

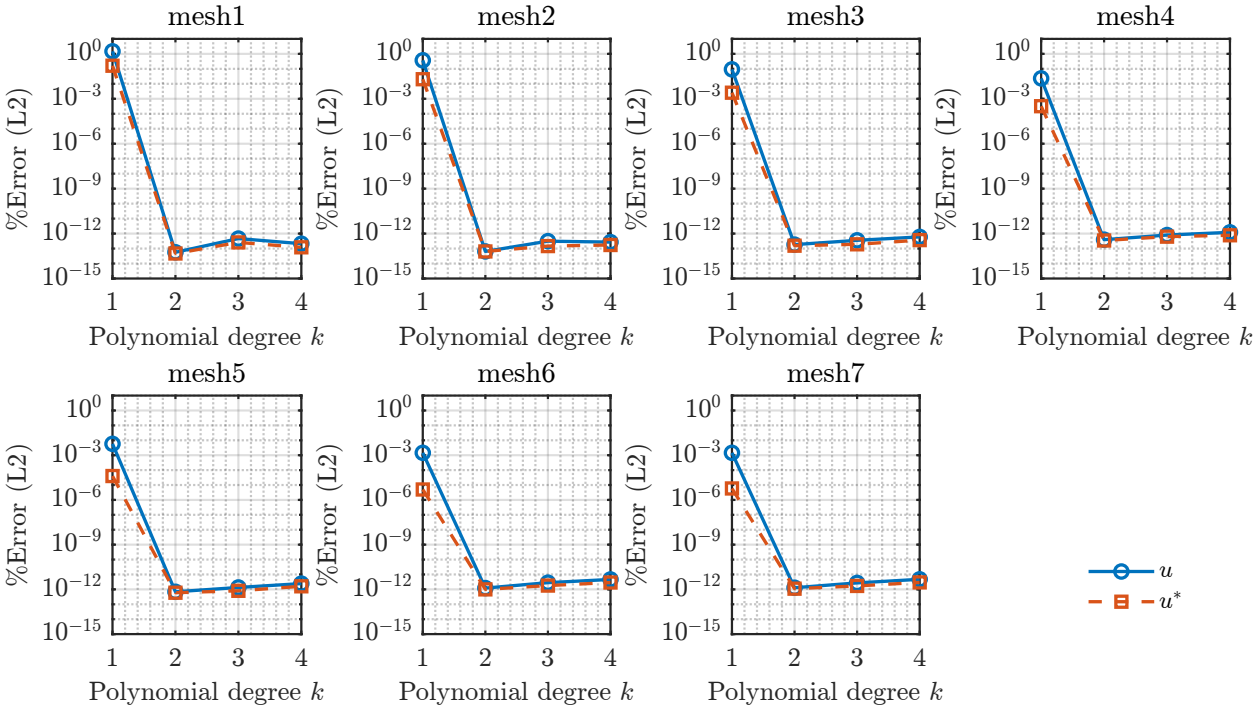


(b) Python CPU

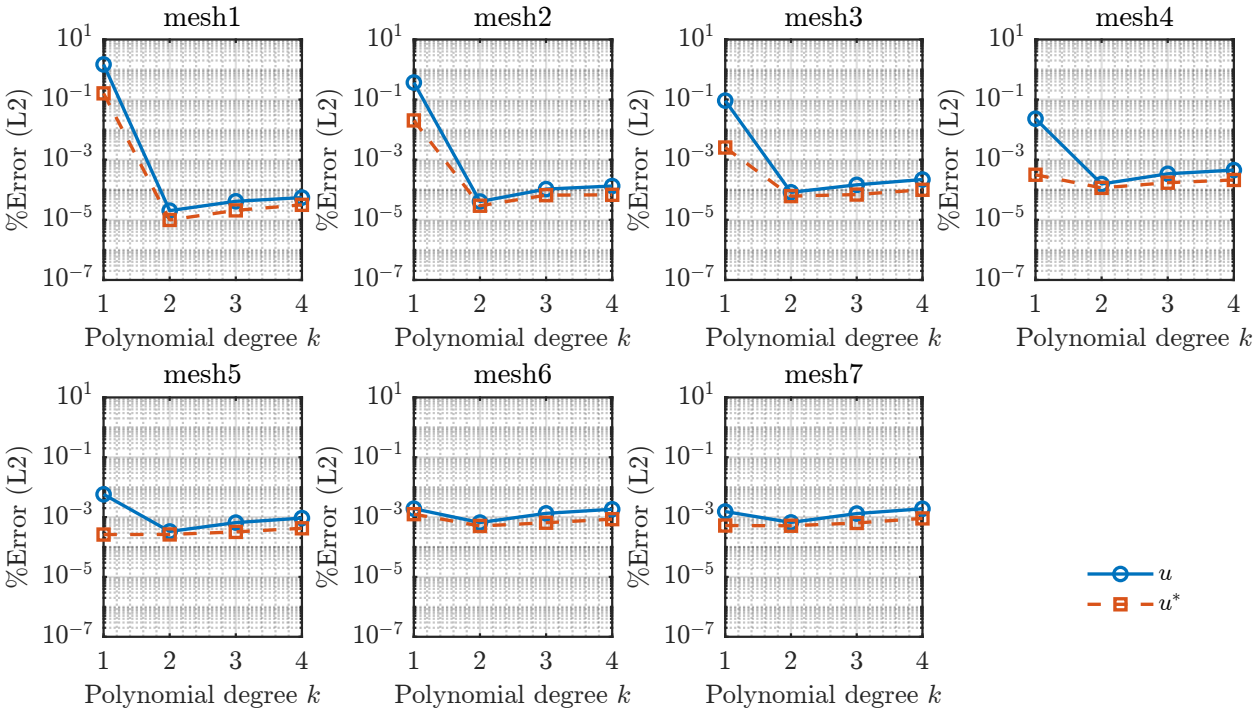


(c) Python GPU

Figure 5.3: *Comparison of HDG solution errors $L_2(u)$ by mesh and polynomial degree (k) for MATLAB CPU, Python CPU and Python GPU implementations.*



(a) MATLAB CPU



(b) Python GPU

Figure 5.4: *Comparison of HDG solution and superconvergent postprocessed solution errors $L_2(u)$ vs $L_2(u^*)$ by mesh and polynomial degree (k) for MATLAB CPU and Python GPU implementations*

5.2.2 Comparisons on the computational cost

Normalized total run time vs polynomial degree

In the present subsection, the **normalized total run time vs polynomial degree** is assessed. For each mesh (*mesh1* to *mesh7*), Figure 5.5 compares MATLAB CPU, Python CPU, and Python GPU *total runtimes* as the polynomial degree k increases to assess the sensitivity to this factor. Notice that each curve is **normalized within its own implementation hardware and mesh**.

All three implementations have progressively longer run times as k is increased in most cases. Python CPU grows steadily with k, meaning the CPU code's total cost is strongly k-driven at any problem's polynomial degree. However, on MATLAB CPU and Python GPU , the normalized curves are comparatively flat, especially on larger meshes, pointing out that in those implementations, the increase in computational cost could be due to mesh size, more than k.

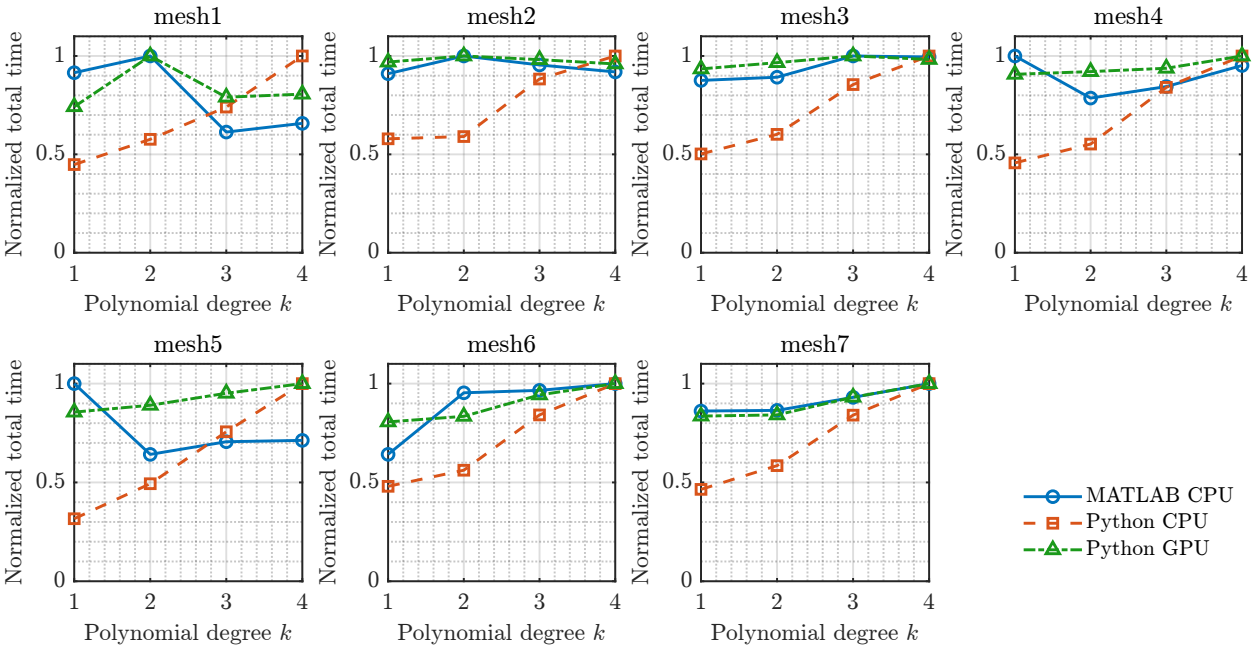


Figure 5.5: Normalized total runtime of the HDG code in the three implementations MATLAB CPU, Python CPU, and Python GPU.

Total run time

Here we compare the **total run time** of the three implementations (*MATLAB CPU*, *Python CPU*, and *Python GPU*) for all tested meshes and polynomial degrees k. Times are reported in milliseconds on a logarithmic scale.

Figure 5.6 plots total time versus mesh id with one curve per degree k in each panel. Two trends stand out. First, both CPU implementations grow steeply with mesh size, as expected from the cost of assembly and the global solve. Notice that the *Python CPU* total time is consistently above *MATLAB CPU*. On the other hand, the *Python GPU* panel is much flatter: for small meshes the GPU is not dramatically faster (kernel launch and data–movement overheads are non–negligible), but as the problem size increases the GPU keeps the growth under control and becomes clearly the

fastest option, which is the main goal of this project.

Moreover, across all three panels, changing k shifts each curve slightly compared to the dominant effect of the mesh size. This is particularly visible on the GPU, where the curves for different k nearly collapse for the smaller meshes and separate only for the largest ones. That behavior is consistent with the local (element) kernels scaling well on the device, while the global cost on the CPU side is more sensitive to the increased trace DoFs that come with higher k, especially seen in Python GPU.

Therefore, we can conclude that from medium to large meshes, the GPU path delivers the best time to solution, *Python CPU* is the slowest, and *MATLAB CPU* sits in between, matching the expectations for a GPU-accelerated HDG pipeline versus CPU-only baselines.

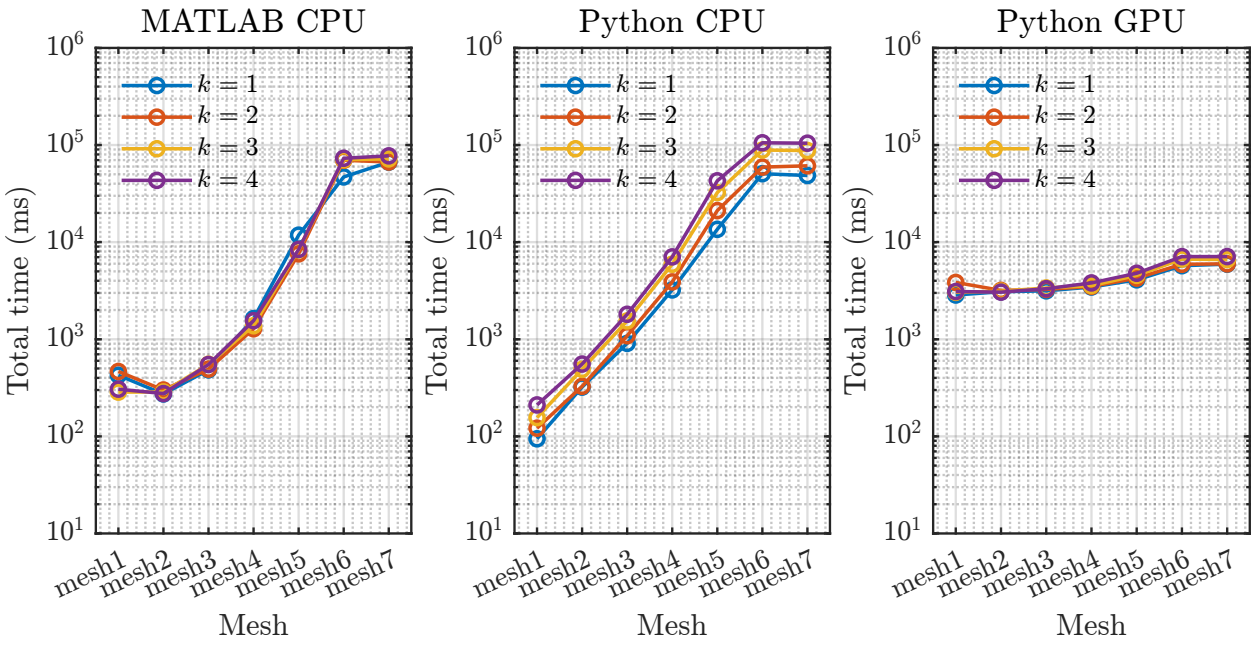


Figure 5.6: *Comparison of total run time by mesh and polynomial degree (k) for MATLAB CPU, Python CPU and Python GPU implementations*

To see the benefits of GPU clearer, Figure 5.7 reports the speed-up CPU/GPU for the total run time of the solver on a fixed mesh, with bars for polynomial degrees $k = 1, \ldots, 4$.

Firstly, for small meshes (mesh1–mesh3), the bars remain well below unity (e.g., 0.03–0.85), showing that for tiny problems the GPU is under-utilized: kernel-launch overheads, synchronizations, and host–device transfers dominate, while the working sets are too small to occupy the device effectively.

A practical reason for the poor GPU performance on the smallest cases is the very low kernel occupancy. During the runs, Numba reports:

" NumbaPerformanceWarning: Grid size 8 will likely result in GPU under-utilization due to low occupancy ".

With only a handful of thread blocks launched (e.g., when the mesh has few elements and k is small), many streaming multiprocessors remain idle and the GPU cannot hide memory latency;

fixed costs (kernel launches and host device transfers) dominate [29]. This explains why the speed-up is $< 1\times$ on meshes 1 - 3 and only approaches or exceeds parity once the grid size increases (more elements and/or higher k), which raises occupancy and amortizes overheads.

Secondly, one could observe a break-even (mesh4). Results are around $1\times$ and begin to exceed it as k increases. At this scale, the arithmetic in assembly, the global CG, and the local elemental kernels start to amortize fixed overheads, bringing CPU and GPU into parity. This will be further analyzed in the next subsection.

Thirdly, for large meshes (mesh5–mesh7), the speed-up rises sharply with both mesh size and degree, reaching 4×–9× on *mesh5* and about 9×–15× on *mesh6–mesh7*. This trend matches expectations: more elements and higher k increase arithmetic intensity and parallel work per kernel, so the GPU sustains higher throughput while CPU time grows steeply.

Regarding the effect of degree k, within each mesh the bars grow monotonically with k, consistent with heavier per-element linear algebra and denser trace systems that favor the GPU's massively parallel execution. The growth begins to level off on the largest meshes (near $15\times$), suggesting we are approaching the memory/bandwidth limits predicted by the roofline model. Note that this will be further studied in the next section.

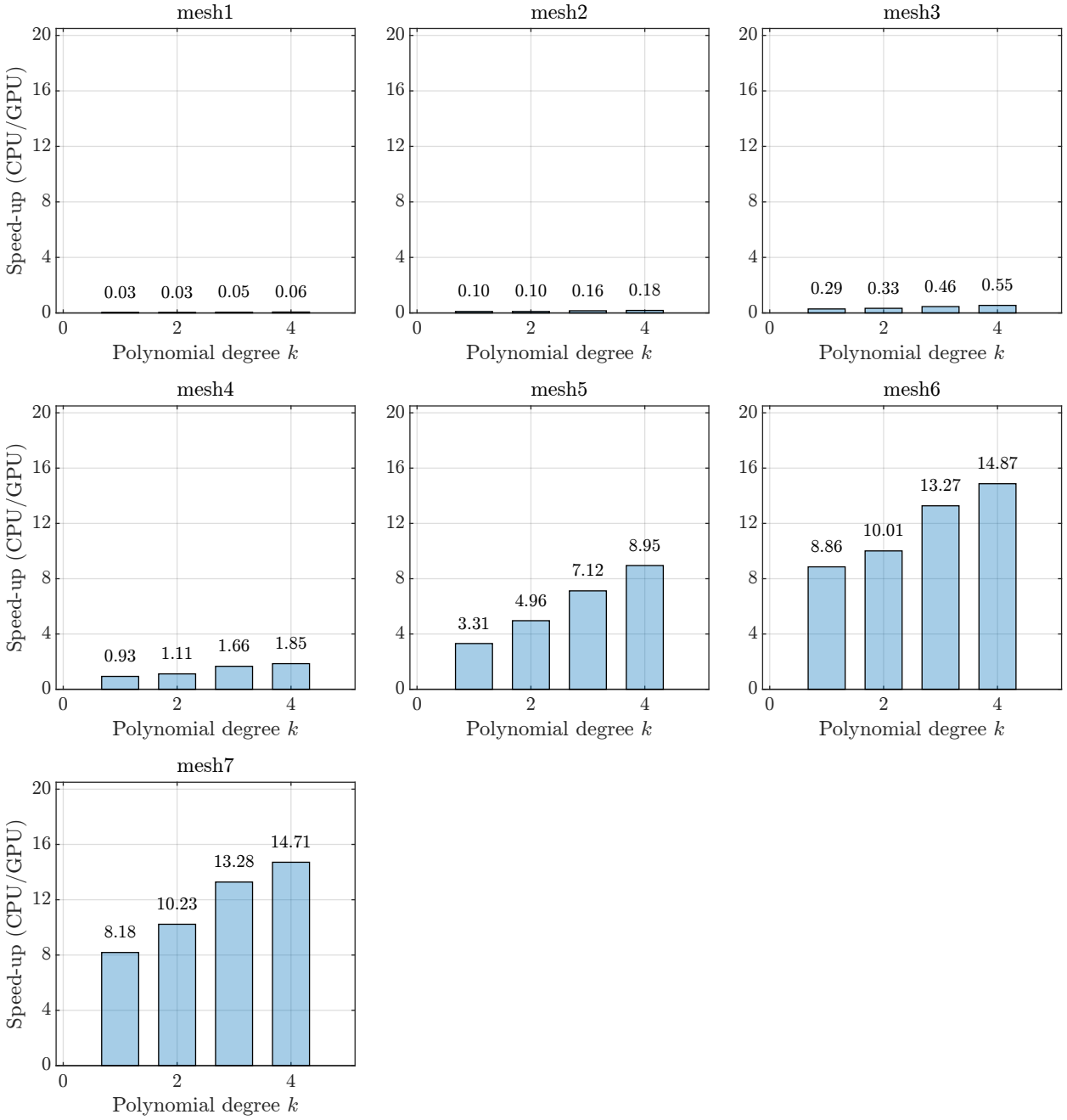


Figure 5.7: *Comparison of the speed-up CPU/GPU for the total run time of the solver by mesh and polynomial degree (k).*

5.3 Python GPU baseline implementation analysis

To understand where time is spent in the GPU implementation we profiled a *baseline* path that uses CuPy/Numba kernels as explained in the previous chapter. The execution is divided into eight phases that are measured consistently across all runs: *preprocess*, *assembly*, *Dirichlet reduce*, *global solve (CG)*, *element (local) solve*, *postprocess*, *errors*, and *transfers*. For each mesh–degree case, we normalize the phase times by the total runtime of that case, being shown in Figure 5.8.

In that figure, for the smallest problems the assembly phase dominates: fixed launch overheads and modest parallelism keep the GPU under-occupied [29], so building the sparse structures takes the

largest share. As the problem grows, the share of *assembly* steadily drops while the *global CG solve* (purple) becomes the bottleneck, ultimately accounting for the majority of runtime on the largest meshes (consistent with a memory-bound SpMV-heavy solver) [2]. The *local element* stage and *postprocess* grow with k but remain secondary contributors. *Errors*, *transfers* and textitpreprocess stay small across all cases, typically a few percent each, confirming that most time is spent on core computation once the problem is large enough.

Regarding the implication this previoulsy commented has, the profile suggests a clear optimization order: (i) improve the CG path (use vendor SpMV, better preconditioning, merge reductions, and ensure high occupancy), (ii) reduce assembly cost (shared-memory staging, coalesced writes, and kernel fusion), and (iii) only then perfectize local/postprocess kernels [1], [2]. Because percentages are normalized per case, absolute runtimes still grow with mesh size; what changes is which phase limits throughput.

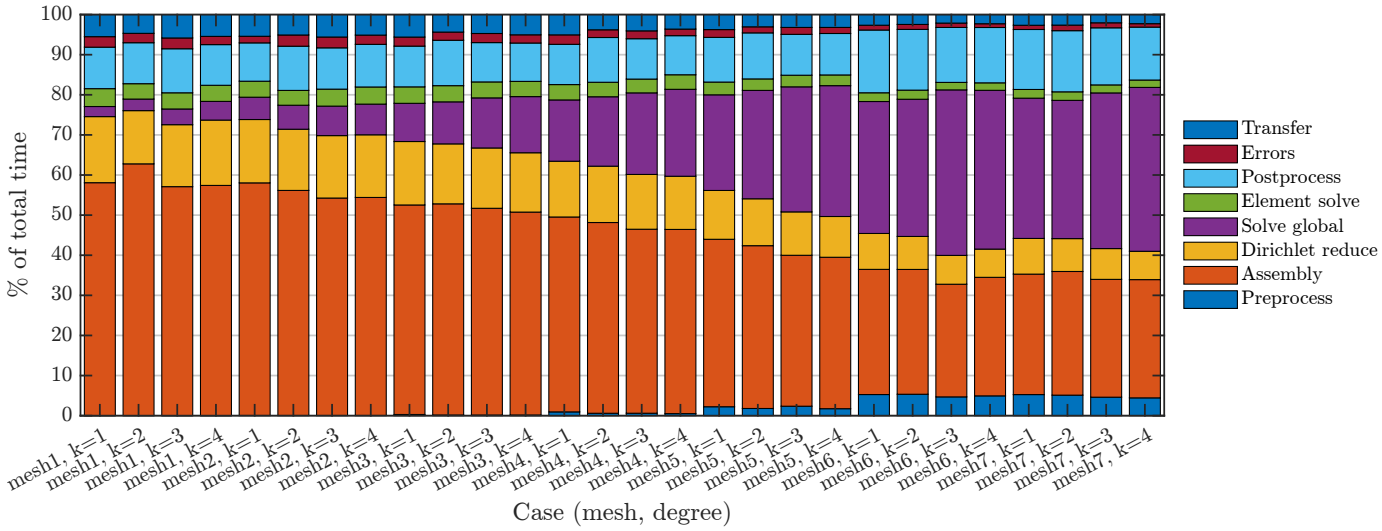


Figure 5.8: *Comparison of the breakdown time among the different solver stages for the different stages of the solver for mesh 7, $k = 4$. Python GPU*

Figure 5.9 reports the comparison of the speed-up CPU/GPU for the different stages of the solver for mesh 7, $k = 4$. Three phases clearly benefit from the GPU are *Errors*, *Postprocess*, and *Assembly* stages Both stages expose massive fine–grain parallelism for any element or quadrature degree and their arithmetic can be batched efficiently on the device.

By contrast, *Local element* and *Global Solve* sit close to unity ($\approx 1.1\times$ and $\approx 1.3\times$), indicating modest acceleration. These kernels are largely memory–bound: the combination of low arithmetic intensity and small effective working sets limits the attainable throughput (see Figure 5.12), so the GPU only marginally outpaces the CPU. Finally, the remaining stage, *Dirichlet*, shows slowdowns. This step is too small to benefit from offloading, so overheads dominate and the CPU wins.

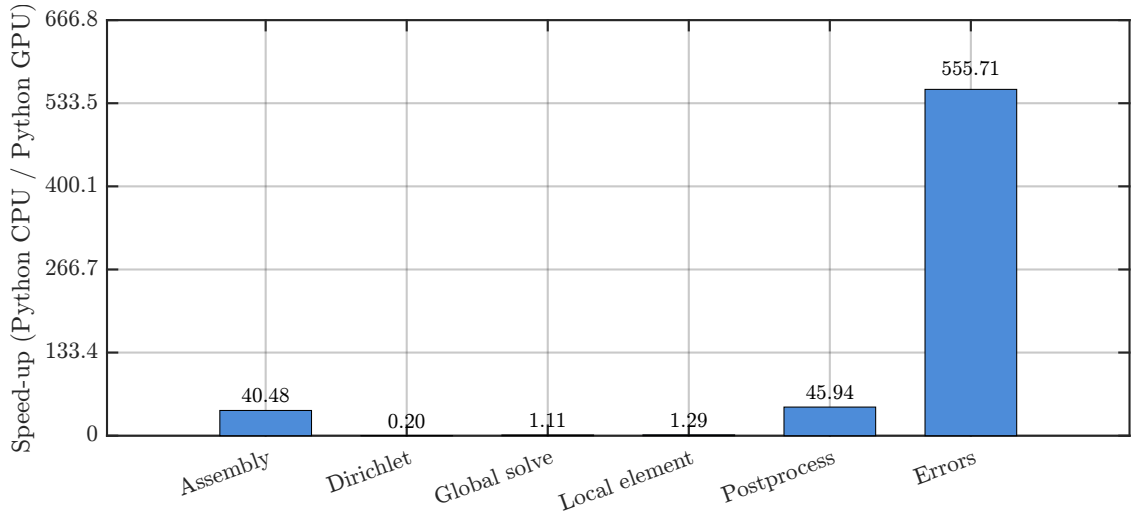


Figure 5.9: *Comparison of the speed-up CPU/GPU for the different stages of the solver for mesh 7, $k = 4$*

Then, focusing on the problematic phase, the global solve (CG), we analyze the Sparse Matrix–Vector Multiply (SPMV) performance, in particular, the achieved memory bandwidth (GB/s) discussed in the previous chapter. This is represented in Figure 5.10.

One could see that for small meshes (mesh1 - mesh3), all curves sit at very low GB/s: launch overheads and low occupancy dominate. From mesh4 - mesh5 onward the GPU fills and bandwidth scales with problem size:

- $k = 3$ scales best, reaching $\approx 280-300$ GB/s on mesh6, i.e. about $40-45\%$ of peak for this device.
- $k = 1$ and $k = 2$ reach $\approx 110-200$ GB/s on the largest meshes ($\sim 17-30\%$ of peak).
- $k = 4$ grows but underperforms $k = 3$: longer and more irregular rows reduce cache locality and coalescing, and increase register pressure. Notice that for even polynomial degrees $k = 2$ and $k = 4$ the curve changes of tendency in the same way at medium-high meshes.

In the runtime breakdown of the baseline GPU implementation (Figure 5.8), the global solve (CG) dominates for large meshes. Figure 5.10 explains why: each CG iteration is built around one or more SpMVs; when SpMV only reaches $20-45\%$ of the memory-bandwidth ceiling, the method becomes memory-bound and the CG share grows with size. Furthermore, one could think that in order to optimize the code, one should push the roofline closer to the memory ceiling.
Figure 5.11 reports the attained throughput (actual amount of data a system successfully processes) for each major phase of the GPU implementation as the mesh is refined and for different polynomial degrees $k \in \{1, 2, 3, 4\}$. In all figures, the vertical axis shows achieved performance in GFLOP/s.

Across phases, performance increases with both mesh size and polynomial order. On very small meshes the curves remain close to zero because the kernels are latency– and launch–dominated. Once the problem supplies enough parallel work, throughput increases, delivering larger GFLOP/s as it raises arithmetic intensity and improves GPU occupancy [37].

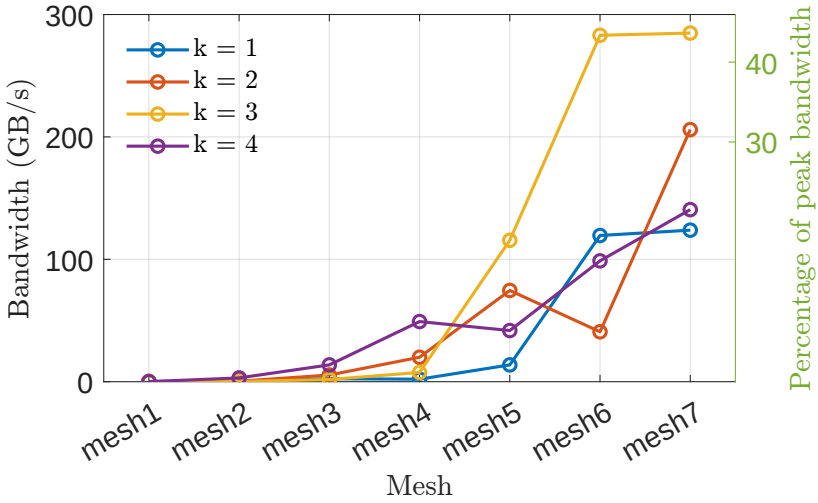


Figure 5.10: *Comparison of the Sparse Matrix–Vector Multiply (SPMV) memory bandwidth (GB/s) by mesh and polynomial degree (k). Python GPU implementation.*

The conjugate–gradient (global solve) phase exhibits the best scaling in Figure 5.11. For the largest meshes the $k = 4$ curve approaches $\sim$1 GFLOP/s, with $k = 3$ somewhat lower and $k = 1{-}2$ below both. This is consistent with a memory–bound, SPMV–centric loop: enlarging the system (by refining the mesh or increasing k) raises the operational intensity and amortizes fixed costs, allowing the GPU to sustain a higher rate. In the GFLOP/s panels the CG phase reaches the highest throughput among phases, yet it stays well below the compute roof and is largely memory-bound (SPMV dominated), as seen in Figure 5.10. At the same time, its arithmetic work scales with system size. The combination *"large work + only moderate GFLOP/s"* explains why, in Figure 5.8, CG becomes the dominant slice for the larger meshes and higher k.

Beyond the global solve, all phases exhibit the same qualitative behavior: their sustained throughput (GFLOP/s) increases with mesh size and polynomial order k, but remains far from the compute roof because kernels operate on many small dense blocks and are largely memory–bound. The local element reconstruction and the quadrature–based postprocess/error evaluations only reach meaningful rates on the largest meshes. Furthermore, their absolute performance stays at $\mathcal{O}(10^{-1})$ GFLOP/s for k=4, making them secondary contributors to wall time. Assembly grows with problem size but delivers very low GFLOP/s due to irregular memory traffic. Finally, the Dirichlet projection is negligible in both work and time.

The stacked "% of total time" bars follow directly from phase throughput: for phase i,

$$t_i = \frac{\text{FLOPs}_i}{\text{GFLOP/s}_i}, \qquad \text{share}_i = \frac{t_i}{\sum_j t_j}.$$

When the work grows with mesh/k but GFLOP/s remains modest, t_i increases and the bar widens. This is precisely why the CG phase. dominated by SPMV and therefore bandwidth–limited, becomes the largest slice at fine meshes/high k, whereas assembly, Dirichlet, local, postprocess and errors stay comparatively small.

If one wants to measurably reduce time-to-solution, one should prioritize the global CG solve, increasing SPMV bandwidth, and strengthening the preconditioner. Other phases can benefit from batching and data reuse, but their low arithmetic intensity and small effective block sizes cap

achievable GFLOP/s; thus, micro-optimizations there will have far less impact on overall runtime than improvements along the CG path.

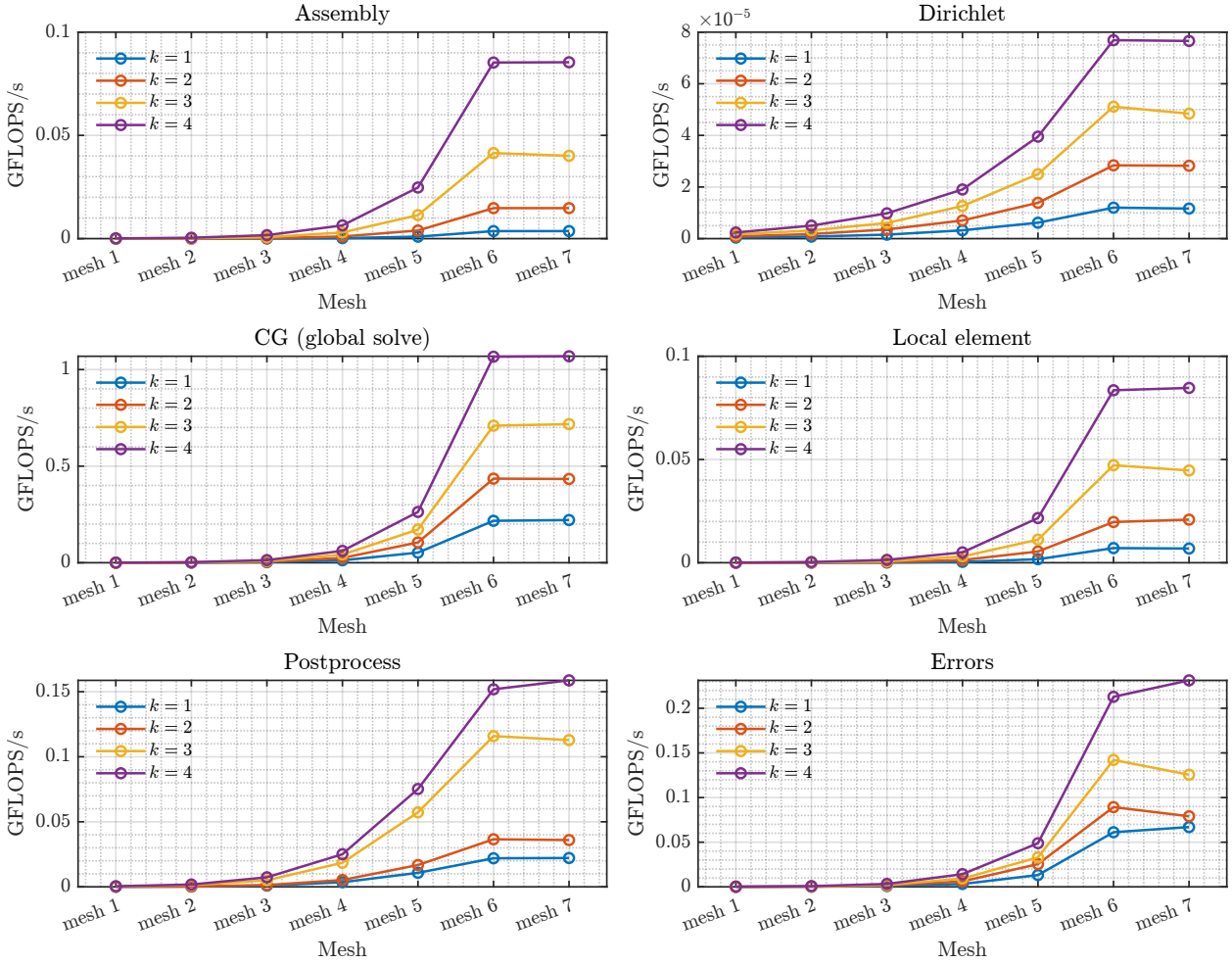


Figure 5.11: *Comparison of the attained throughput or data processed for each major phase of the GPU by mesh and polynomial degree (k). Python GPU implementation.*

Finally, Figure 5.12 shows TUCAN GPU roofline: the blue line is the machine roof (memory roof with slope = peak bandwidth up to the vertical magenta line at the crossover operational intensity $OI_{\times}$ = Peak GF/s/BW, then a flat compute roof at the GPU peak). Each colored marker is one *(mesh, k)* run of a given phase, located at its measured operational intensity (FLOP/Byte) and attained performance (GFLOP/s).

All measured points lie well to the left of $OI_{\times}$ and far below the flat roof, i.e. they are firmly in the memory-bound regime. Increasing mesh size generally moves points upward (more parallel work amortizes launch and latency costs) but does not change their OI, which is algorithmic and phase-specific. None of the phases approaches the limit; the gap to the memory roof reflects non-ideal access patterns and occupancy limits from small kernels.

Let's do a final interpretation per phase:

- **CG (global solve).** This cluster attains the highest GFLOP/s among phases, yet remains far below the roof. Performance is set by memory traffic and by the irregular row-wise gathers

(low OI) [1]. The upward trend with mesh/k is consistent with better GPU utilization for larger systems.

- **Postprocess & Errors.** Both are quadrature-driven kernels with $OI \approx 0.8$–1.5. They gain throughput on larger meshes, but still sit well under the roof because they do operations with modest arithmetic per byte.
- **Local element.** The reconstruction applies many tiny dense blocks. Its OI is moderate ($\sim$ 0.4), but block sizes are small, so latency and limited occupancy bound performance; throughput improves with mesh/k but remains below the bandwidth ceiling.
- **Assembly.** Points around $OI \sim 2$–8 but with very low GFLOP/s. Element-wise formation plus global scatters produce irregular memory access and synchronization, so compute throughput stays modest despite the complex work.
- **Dirichlet projection.** Extremely small kernels ($OI \sim 0.5$) and negligible GFLOP/s; this phase is essentially free at the application scale and never limits time-to-solution.

Since all phases sit in the memory-bound region and well below the memory roof, improving data movement dominates returns. For CG this means raising effective SPMV bandwidth. Regarding local/postprocess/error kernels, batching and reuse of element data help, but small effective block sizes cap attainable GFLOP/s. Then, assembly and Dirichlet remain minor contributors. Finally, the clustering seen here is consistent with the time-breakdown chart Figure 5.8 and the *CG* (SPMV) Figure 5.10: *with large work and only moderate bandwidth, becomes the dominant share of total runtime as the problem grows.*

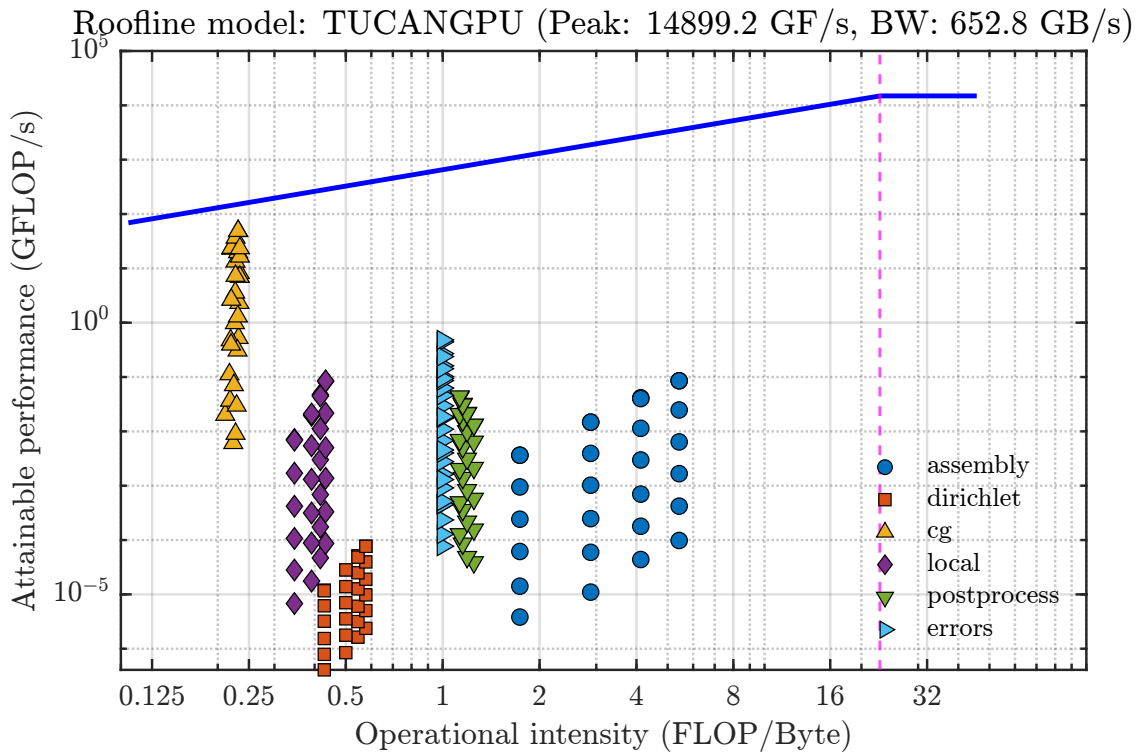


Figure 5.12: *Roofline model of TUCAN GPU. Each colored marker is one (mesh, k) run of a given phase, located at its measured operational intensity (FLOP/Byte) and attained performance (GFLOP/s).*

CHAPTER 6

Conclusions

Contents

6.1 Present work conclusions

Throughout this project we have presented the HDG formulation and shown that, when paired with GPU computing, it can solve CFD problems with high accuracy at a lower computational cost.

First, we verified that HDG benefits from higher polynomial orders and larger meshes, yielding fewer global DoFs than a standard DG discretization and, typically, a sparser global matrix.

We then compared three realizations (4-core *MATLAB CPU*, 4-core *Python CPU*, and *Python GPU*) to assess accuracy and to identify the regimes (mesh size, degree). The three implementations produced the same numerical solution trend across meshes and polynomial degrees. Minimum errors on the GPU (10^{-6}) did not reach the very lowest CPU values (10^{-16}) because single precision (*float32*) was used in the main GPU path to exploit device throughput. Even so, the postprocessed solution reduced the error for all implementations; in the GPU case, we computed this final step in double precision (*float64*) to expose differences.

Regarding run times, all implementations showed that the dominant growth in cost is driven more by mesh size than by the polynomial degree. For the meshes tested, the GPU is not advantageous on very small cases, breaks even around *mesh4*, and provides substantial gains, up to $\sim 15\times$, for large problems, particularly at higher orders.

Regarding the Python GPU baseline implementation analysis, several interesting conclusions are reached. For small problems, the GPU is under-occupied and the *assembly* stage dominates. As the mesh and the polynomial degree grow, the *global solve (CG)* becomes the bottleneck and accounts for the majority of the runtime. This shift matches the behavior expected for memory-bound sparse kernels and is consistent across all test cases.

The acceleration given by the different steps of the code is assessed. Stages with abundant, embarrassingly parallel work, *errors*, *postprocess*, and *assembly*, exhibit large speed-ups versus the CPU because their loops can be batched effectively. In contrast, *local element* and *Dirichlet* either see modest gains or even slow down (respectively) due to very small working sets and launch/latency overheads.

The SPMV bandwidth attained by CG reaches about 17–45% of the device peak, keeping CG firmly in the memory-bound regime. The phase thus combines *large work* with *only moderate throughput*, which explains its growing share in the time breakdown. Furthermore, in the roof model, all measured phase points lie well to the left of the crossover intensity and far below the compute roof, confirming that performance is limited by data movement rather than by raw FLOP capability.

In some stages (Global solve or cg, errors, or postprocess, larger meshes move points upward (better utilization) but do not change their operational intensity. However, for the rest of them and especially for the assembly stage, as the polynomial degree and mesh refinement are increased, the points lie more on the top and right, meaning a better employment of TUCANGPU properties.

Finally, the baseline model developed in this thesis evidences that performance is governed by memory bandwidth: once the problem is large enough, the CG/SPMV path dictates runtime, while the remaining phases either become efficient or remain too small to matter. This diagnosis sets a clear direction for optimization in the next chapters.

6.2 Proposal for future research

HDG Poisson problem

If one wished to shorten time-to-solution, effort should be directed first to the CG path: increase effective SPMV bandwidth (e.g., vendor CSR kernels, format/ordering tuned for coalescing), reduce reductions and synchronizations, and strengthen the preconditioner. Furthermore, it is highly encouraged to explore *matrix-free operator* application for the global trace operator. Notice that optimizing the remaining phases offers limited global impact because their arithmetic intensity and problem granularity cap attainable GFLOP/s.

The baseline uses a single GPU and a Poisson/HDG workload. Future work should evaluate stronger GPU-friendly preconditioners, experiment with mixed precision for CG, explore alternative sparse formats, and consider the overlap of computation and data movement.

From Poisson to realistic CFD cases

Stokes and incompressible Navier–Stokes. The GPU HDG implementation developed in this work for the Poisson equation, once optimized, could be easily extended to the Stokes system and then to unsteady incompressible Navier-Stokes. In these problems, Picard/Newton linearizations could be used, maintaining static condensation so that only trace unknowns would be global. Moreover, one could integrate time integration too.

Canonical validation problems. One could get the advantage of this GPU algorithm to validate accuracy and cost in problems like: manufactured solutions for Stokes or Incompressible Navier-Stokes on curved meshes; lid-driven cavity and Poiseuille channel for steady flows; *2D circular cylinder* at low Reynolds (laminar shedding) to check lift/drag and Strouhal number. Each case should include grid and polynomial-degree studies and direct comparison against reference data.

Discretization and meshing. One could generalize to curved high-order elements (isoparametric mapping), and enable local p-adaptivity depending on the areas that require more accuracy (i.e., raise degree near vortices/walls). Also, one could add robust wall boundary conditions (no-slip, wall models for WMLES).

Bibliography

[1] Richard Barrett, Michael W. Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. Templates for the solution of linear systems: Building blocks for iterative methods. *SIAM (Society for Industrial and Applied Mathematics)*, 1994. URL: https://doi.org/10.1137/1.9781611971538.

[2] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. *Proceedings of SC '09 (ACM/IEEE Conference on High Performance Computing, Networking, Storage and Analysis)*, 2009. URL: https://doi.org/10.1145/1654059.1654078.

[3] Jesse Chan, Zheng Wang, Axel Modave, Jean-François Remacle, and T. Warburton. Gpu-accelerated discontinuous galerkin methods on hybrid meshes. *Journal of Computational Physics*, 2016. URL: https://doi.org/10.1016/j.jcp.2016.04.003.

[4] Bernardo Cockburn, Jayadeep Gopalakrishnan, and Raytcho Lazarov. Unified hybridization of discontinuous galerkin, mixed, and continuous galerkin methods for second order elliptic problems. *SIAM Journal on Numerical Analysis*, 2009. URL: https://doi.org/10.1137/070706616.

[5] Bernardo Cockburn, Jayadeep Gopalakrishnan, Ngoc C. Nguyen, Jaime Peraire, and Francisco-Javier Sayas. Analysis of hdg methods for stokes flow. *Mathematics of Computation*, 2011. URL: https://doi.org/10.1090/S0025-5718-2010-02410-X.

[6] CuPy. Cupy library overview. 2020. URL: https://docs.cupy.dev/en/stable/overview.html.

[7] Miguel Encinar. Hpc in heterogeneous architectures. *UC3M*. URL: https://torroja.dmt.upm.es/~miguel/reveal.js/.

[8] C. Fletcher. Computational techniques for fluid dynamics 1. *Fundamental and General Techniques, Springer, Berlin*, 1998. URL: https://link.springer.com/book/10.1007/978-3-642-58229-5.

[9] Khronos OpenCL Working Group. The opencl specification, version 1.0 (revision 43). *Khronos Group*, 2009. URL: https://old.hotchips.org/wp-content/uploads/hc_archives/hc21/1_sun/HC21.23.2.OpenCLTutorial-Epub/HC21.23.295.opencl-1.0.43.pdf.

[10] Montjoie User guide. Triangular finite elements spaces. URL: https://www.math.u-bordeaux.fr/~durufle/montjoie/triangle.php.

[11] Tomasz Kozlowski Guojun Hu. Assessment of continuous and discrete adjoint method for sensitivity analysis in two-phase flow simulations. *Elsevier*, 2018. URL: https://arxiv.org/abs/1805.08083.

[12] John Hennessy and David Patterson. Computer architecture - a quantitative approach. 2007. URL: https://www.researchgate.net/publication/200039347_Computer_Architecture_-_A_Quantitative_Approach.

[13] Intel Corporation. oneapi math kernel library (onemkl) developer reference. *Intel Developer Reference*, 2023. URL: https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2023-2/overview.html.

[14] Michael Garland Kevin Skadron John Nickolls, Ian Buck. Scalable parallel programming with cuda. *ACM Queue*, 2008. URL: https://queue.acm.org/detail.cfm?id=1365500.

[15] Robert M. Kirby, Spencer J. Sherwin, and Bernardo Cockburn. To cg or to hdg: A comparative study. *Journal of Scientific Computing*, 2012. URL: https://link.springer.com/article/10.1007/s10915-011-9501-7.

[16] Andreas Klöckner, Tim Warburton, Jeffrey Bridge, and Jan S. Hesthaven. Nodal discontinuous galerkin methods on graphics processors. *Journal of Computational Physics*, 2009. URL: https://doi.org/10.1016/j.jcp.2009.06.041.

[17] Martin Kronbichler and Wolfgang A. Wall. A performance comparison of continuous and discontinuous galerkin methods with fast multigrid solvers. *SIAM Journal on Scientific Computing*, 2018. URL: https://doi.org/10.1137/16M110455X.

[18] Gloria Ortega López. *High Performance Computing for Solving Large Sparse Systems. Optical Diffraction Tomography as a Case of Study.* Ph.d. thesis, University of Almería, Department of Informatics, Almería, Spain, May 2014. URL: https://editorial.ual.es/libro/high-performance-computing-for-solving-large-sparse-systems-optical-diffraction-tomography-144511/.

[19] Yu Lv and John Ekaterinaris. Recent progress on high-order discontinuous schemes for simulations of multiphase and multicomponent flows. *Progress in Aerospace Sciences*, 2023. doi:10.1016/j.paerosci.2023.100929.

[20] Paul E. Plassmann Mark Jones. A parallel graph coloring heuristic. *SIAM Journal on Scientific and Statistical Computing*, 1992. URL: https://doi.org/10.1137/0914041.

[21] Manish Parashar Hari Sudan Mingliang Wang, Hector Klie. A parallel graph coloring heuristic. *Computational Science - ICCS 2009, 9th International Conference, Baton Rouge, LA, USA, May 25-27, 2009, Proceedings, Part I*, 2009. URL: https://link.springer.com/chapter/10.1007/978-3-642-01970-8_87.

[22] Michael Garland Nathan Bell. Implementing sparse matrix-vector multiplication on throughput-oriented processors. *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009, November 14-20, 2009, Portland, Oregon, USA*, 2009. URL: https://dl.acm.org/doi/10.1145/1654059.1654078.

[23] Ngoc C. Nguyen, Jaime Peraire, and Bernardo Cockburn. A hybridizable discontinuous galerkin method for stokes flow. *Computer Methods in Applied Mechanics and Engineering*, 2010. URL: https://doi.org/10.1016/j.cma.2009.12.037.

[24] Ngoc C. Nguyen, Jaime Peraire, and Bernardo Cockburn. A hybridizable discontinuous galerkin method for the incompressible navier–stokes equations. *AIAA Aerospace Sciences Meeting (Paper 2010-362)*, 2010. URL: https://arc.aiaa.org/doi/10.2514/6.2010-362.

[25] Ngoc C. Nguyen, Jaime Peraire, and Bernardo Cockburn. Hybridizable discontinuous galerkin methods. *Lecture Notes in Computational Science and Engineering*, 2011. URL: https://link.springer.com/chapter/10.1007/978-3-642-15337-2_4.

[26] Ngoc C. Nguyen, Jaime Peraire, and Bernardo Cockburn. An implicit high-order hybridizable discontinuous galerkin method for the incompressible navier–stokes equations. *Journal of Computational Physics*, 2011. URL: https://doi.org/10.1016/j.jcp.2010.10.032.

[27] NVIDIA Corporation. Cuda c++ best practices guide. 2020. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide.

[28] NVIDIA Corporation. Cuda refresher: The cuda programming model. 2020. URL: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model.

[29] NVIDIA Corporation. Cuda c best practices guide (occupancy and kernel launch configuration). *NVIDIA Developer Documentation*, 2024. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/.

[30] Norwegian Centre of Excellence. Finite element basis functions. URL: https://hplgit.github.io/num-methods-for-PDEs/doc/pub/approx/sphinx/._main_approx004.html.

[31] Michael Fleck Patrick Zimbrod and Johannes Schilp. An application-driven method for assembling numerical schemes for the solution of complex multiphysics problems. *Appl. Syst.Innov*, 2024. URL: https://www.mdpi.com/2571-5577/7/3/35.

[32] Michael Fleck Patrick Zimbrod and Johannes Schilp. An application driven method for assembling numerical schemes for the solution of complex multiphysics problems. *Appl. Syst. Innov. 2024*, 2024. doi:10.3390/asi1010000.

[33] Jaime Peraire, Ngoc C. Nguyen, and Bernardo Cockburn. A hybridizable discontinuous galerkin method for the compressible euler and navier–stokes equations. *AIAA Aerospace Sciences Meeting (Paper 2010-363)*, 2010. URL: https://arc.aiaa.org/doi/10.2514/6.2010-363.

[34] Xevi Roca, Ngoc C. Nguyen, and Jaime Peraire. Gpu-accelerated sparse matrix–vector product for a hybridizable discontinuous galerkin method. *AIAA Aerospace Sciences Meeting (Paper 2011-687)*, 2011. URL: https://www.mit.edu/~cuongng/project/sc3/sc3.pdf.

[35] Yonghao Zhang Lei Wu Wei Su, Peng Wang. A high-order hybridizable discontinuous galerkin method with fast convergence to steady-state solutions of the gas kinetic equation. *Journal of Computational Physics*, 2019. URL: https://doi.org/10.1016/j.jcp.2018.08.050.

[36] Weka. Hpc gpus explained (how they work together). *s*, 2022. URL: https://www.weka.io/learn/hpc/gpu-and-hpc-explained/?utm_source.

[37] Samuel Williams, Andrew Waterman, and David A. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 2009. URL: https://doi.org/10.1145/1498765.1498785.

[38] Nicholas Wilt. The cuda handbook: A comprehensive guide to gpu programming. *Pearson*, 2013. URL: https://ptgmedia.pearsoncmg.com/images/9780321809469/samplepages/0321809467.pdf.

[39] F. D. Witherden and A. Jamesony. Future directions of computational fluid dynamics. *23rd AIAA Computational Fluid Dynamics Conference*, 2017. doi:10.2514/6.2017-3791.

[40] Sergey Yakovlev, David Moxey, Robert M. Kirby, and Spencer J. Sherwin. To cg or to hdg: A comparative study in 3d. *Journal of Scientific Computing*, 2016. URL: https://link.springer.com/article/10.1007/s10915-015-0076-6.