Dataflow Optimized Reconfigurable Acceleration for FEM-based CFD Simulations

Anastassis Kapetanakis, Aggelos Ferikoglou, George Anagnostopoulos, and Sotirios Xydis
Microprocessors and Digital Systems Lab, National Technical University of Athens, Greece
{akapetanakis, aferikoglou, geoanagn, sxydis}@microlab.ntua.gr

***Abstract*—Computational Fluid Dynamics (CFD) simulations are essential for analyzing and optimizing fluid flows in a wide range of real-world applications. These simulations involve approximating the solutions of the Navier-Stokes differential equations using numerical methods, which are highly compute- and memory-intensive due to their need for high-precision iterations. In this work, we introduce a high-performance FPGA accelerator specifically designed for numerically solving the Navier-Stokes equations. We focus on the Finite Element Method (FEM) due to its ability to accurately model complex geometries and intricate setups typical of real-world applications. Our accelerator is implemented using High-Level Synthesis (HLS) on an AMD Alveo U200 FPGA, leveraging the reconfigurability of FPGAs to offer a flexible and adaptable solution. The proposed solution achieves 7.9× higher performance than optimized Vitis-HLS implementations and 45% lower latency with 3.64× less power compared to a software implementation on a high-end server CPU. This highlights the potential of our approach to solve Navier-Stokes equations more effectively, paving the way for tackling even more challenging CFD simulations in the future.**

***Index Terms*—Computational Fluid Dynamics (CFD), Simulation, Accelerator, FPGA, High-Level Synthesis (HLS)**

I. Introduction

Computational Fluid Dynamics (CFD) play a vital role in analyzing and optimizing fluid flow in complex systems, enhancing design, performance, and safety across industries such as aerospace [1], automotive [2], and environmental engineering [3]. Constructing physical prototypes for studying these problems is costly and time-consuming, particularly for intricate fluid dynamics scenarios that require elaborate setups and significant resources. CFD simulations offer a cost-effective and efficient alternative by providing detailed insights and flexibility for design optimization without physical models. They facilitate virtual testing across diverse conditions and use cases [4], [5], helping to identify potential issues early, mitigate risks, and improve overall design performance [6].

CFD simulations involve solving the Navier-Stokes Partial Differential Equations (PDEs), which are fundamental in describing fluid flow behavior [7]. Solving these PDEs analytically is challenging, so numerical methods are used to approximate their solutions. The most common methods are the Finite Difference Method (FDM) and the Finite Element Method (FEM). FDM [8] approximates the derivatives in the Navier-Stokes equations using differences between function values at discrete points on a structured grid, facilitating implementation. However, FDM's dependence on structured grids restricts its effectiveness for complex geometries and irregular boundaries, making it challenging to handle intricate boundary conditions and maintain stability. FEM [9] discretizes the domain into small elements and employs interpolation functions to approximate solutions, utilizing unstructured meshes that can adapt to intricate geometries and irregular boundaries. This flexibility makes FEM especially effective for modeling complex real-world applications, such as flows around irregularly shaped aircraft wing [10]. However, it comes with increased implementation complexity and higher computational demands.

Numerically solving the Navier-Stokes equations is time-consuming because of the need for fine grid spacing, numerous time-steps to reach statistical steady-state solutions, and the complexity of the algorithms. To achieve a practical time-to-solution, it is crucial not only to use efficient numerical modeling but also to leverage the advanced capabilities of modern computational architectures. General-purpose processors, such as Central Processing Units (CPUs) and Graphics Processing Units (GPUs), are commonly used for numerically solving PDEs. Current CPU implementations [11] primarily rely on Matrix-Vector multiplication, utilizing matrix processing libraries to enhance speed. However, this approach has two major drawbacks that hinder performance and efficiency: i) it requires substantial buffer memory to store the large, sparse matrices, and ii) CPUs find it challenging to leverage data and computation reuse in these matrices [12]. Recently, GPUs have also been utilized to accelerate CFD simulations [13], [14] due to their ability to manage large-scale data and computations more efficiently than CPUs. Despite these advantages, GPUs continue to struggle with low energy efficiency, even when processing simpler PDEs on smaller grids [15].

Several domain-specific accelerators have been developed to numerically solve PDEs using the FDM method. Chen et al. [15] proposed an FDM accelerator for 2D Laplace and Poisson equations on grids up to 128x128, utilizing processing-in-memory (PIM), though it suffers from limited computing precision. Mu et al. [16] developed an accelerator for the 2D Laplace equation on a 21x21 grid with dynamic computing precision. In their subsequent work [17], they expanded support to 3D Laplace equations on a 16x16x16 grid without external memory accesses, but both designs are limited to specific grid sizes. Li et al. [18] recently introduced FDMAX, an elastic accelerator architecture designed to overcome some of these limitations. FDMAX can handle 2D Laplace, Poisson, Heat, and Wave equations with arbitrary grid sizes using 32-bit floating-point precision, offering notable improvements in performance and energy efficiency. However, existing solutions

This work has been partially funded by the EU Horizon 2022 program under grant agreement No 10109669 REFMAP (https://www.refmap.eu/).

978-3-9826741-0-0/DATE25/© 2025 EDAA

target the FDM method and do not address the Navier-Stokes equations. This implies that adapting these architectures would still encounter major limitations due to the inherent constraints of FDM, particularly in handling complex geometries. Additionally, these accelerators are based on ASIC designs, which are inflexible, costly, and time-consuming to fabricate [19]. Once built, ASICs cannot be modified, making it challenging to adjust key parameters like boundary conditions or grid sizes that are essential for realistic CFD simulations. Recently, Friebel et al. [20] introduced a FEM-based reconfigurable accelerator. However, their emphasis on resource-constrained FPGAs restricts performance as they do not take advantage of the capabilities provided by modern cloud FPGA devices.

In this work, we present the first FEM-based reconfigurable accelerator specifically tailored for solving the Navier-Stokes equations. The proposed accelerator employs FEM for spatial discretization, enabling it to adjust to the intricate geometries and setups typical in practical complex CFD applications. We developed our accelerator architecture using High-Level Synthesis (HLS), a user-friendly approach for programming FPGAs. Rather than opting for an ASIC-based design, we utilize the reconfigurability of FPGAs, enabling flexible and dynamic hardware adaptation. This allows for adjustments to different grid sizes or boundary conditions, for instance. From an architectural standpoint, we introduced tailored source code restructurings that enables HLS to exploit Task Level Parallelism (TLP). To further boost TLP efficiency, memory-aware optimizations are proposed to parallelize off-chip memory transfers to the FPGA's reconfigurable fabric, alongside directive-based HLS micro-architectural optimizations to enhance the accelerator's performance through Initiation Interval minimization. We thoroughly evaluate the performance of our accelerator by deploying it on an AMD Alveo U200 FPGA, showing that the proposed solution achieves 7.9× higher performance with respect to optimized Vitis-HLS implementations. In comparison with its software implementation counterpart mapped on a high-end server CPU, we show that the proposed solution delivers 45% latency gains in end-to-end CFD simulations, while dissipating 3.64× less power.

II. Theoretical Background

A. Navier-Stokes Equations

The Navier-Stokes PDEs describe the evolution of a fluid's velocity field over time, influenced by forces like pressure, viscous stresses, and external factors such as gravity [7]. We focus on the 3D compressible Navier-Stokes equations, as detailed in [14], which are described by the following equations:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}) = 0 \tag{1}$$

$$\frac{\partial (\rho \boldsymbol{u})}{\partial t} + \nabla \cdot (\rho \boldsymbol{u} \otimes \boldsymbol{u}) + \nabla p - \nabla \boldsymbol{\tau} = \boldsymbol{f} \tag{2}$$

$$\frac{\partial E}{\partial t} + \nabla \cdot ((E+p)\boldsymbol{u}) - \nabla \cdot (\boldsymbol{\tau} \cdot \boldsymbol{u}) - \nabla \cdot (\kappa \nabla T) = S \tag{3}$$

Equations 1, 2, and 3 correspond to the conservation of mass, conservation of momentum, and the conservation of energy, respectively. The spatiotemporarily dependent variables to be solved are the *fluid density* (ρ), *velocity* ($\boldsymbol{u}$), and *temperature* (T). The *total energy* (E) and *pressure* (p) are related to these variables through constitutive equations, following the ideal gas law. The *viscous stress tensor* ($\boldsymbol{\tau}$) represents the internal frictional forces within the fluid caused by its viscosity. The terms $\boldsymbol{f}$ and S are the *source terms* of Equations 2, and 3, accounting for external energy inputs/losses within the system. The parameter κ is a constant that represents the *fluid's thermal conductivity*. We solve the equations using the initial and boundary conditions defined by the Taylor-Green Vortex (TGV) problem [21], [14].

B. Numerical Methods

Since the Navier-Stokes equations cannot be solved analytically, numerical methods are used to approximate the solution. Given that ρ, $\boldsymbol{u}$, and T are spatiotemporally dependent, we employ the Finite Element Method (FEM) for spatial discretization and the Runge-Kutta Method (RK) to compute the system's time evolution. In the following paragraphs, we outline the fundamentals of these two methods.

Finite Elements Method. Consider a convection-diffusion model of the form $A(x) = 0$, where $A(x) = M(x) + C(x) + D(x)$. The operators $M(x) = \frac{\partial x}{\partial t}$, $C(x) = \nabla \cdot \boldsymbol{f}(x)$, and $D(x) = -\nabla \cdot (\lambda \nabla x)$, correspond to the *Mass*, *Convection*, and *Diffusion* terms, respectively[1] For the geometry to be simulated, we assume it is represented by a discretized mesh. The mesh consists of volume elements defined by vertices and edges, allowing for the representation of complex geometries beyond simple cubes. The unknown function x at each node of element e is represented by the vector $\boldsymbol{x}^e = [x_1^e \ \dots \ x_n^e]^T$. To approximate x^e, a linear combination of n shape functions N_i is used, each of which is equal to 1 at its respective node and 0 at all other nodes within the same element. This leads to the trial function $x^e = \sum_i x_i^e \cdot N_i$. The trial function is then substituted into the original differential equation to compute the residual of the differential equation for that element. The goal of FEM is to find the coefficients x_i^e for all elements such that:

$$\sum_e \int_{V_e} N_i \cdot A(x^e) dV = 0 \ \ i = 1, \dots, n \tag{4}$$

Since these integrals are typically not solvable in closed form, the Gauss-Lobatto-Legendre (GLL) numerical integration technique is employed. This reformulates Equation 4 as:

$$\sum_e \sum_g W_g N_i(\boldsymbol{\xi_g}) \cdot A(x^e(\boldsymbol{\xi_g})) = 0 \ \ i = 1, \dots, n \tag{5}$$

where W_g and $\boldsymbol{\xi_g}$ denote the quadrature weights and points. The computation of the *Mass*, *Diffusion*, and *Convection* terms at the quadrature points $\boldsymbol{\xi_g}$ yields a linear system of equations of the form $\boldsymbol{K}\tilde{\boldsymbol{x}} = \boldsymbol{b}$ where $\tilde{\boldsymbol{x}} = [(\boldsymbol{x}^{e_1})^T \dots (\boldsymbol{x}^{e_m})^T]^T$ combines the unknown vectors of all elements. In this linear system, $\boldsymbol{b}$ represents a constant term, while $\boldsymbol{K}$ is a diagonal matrix that encapsulates the information from Equation 5.

Runge-Kutta Method. Consider an initial value problem for an Ordinary Differential Equation (ODE) of the form

[1]Equations 1, 2, and 3 can be mathematically expressed as Convection-Diffusion models [22].

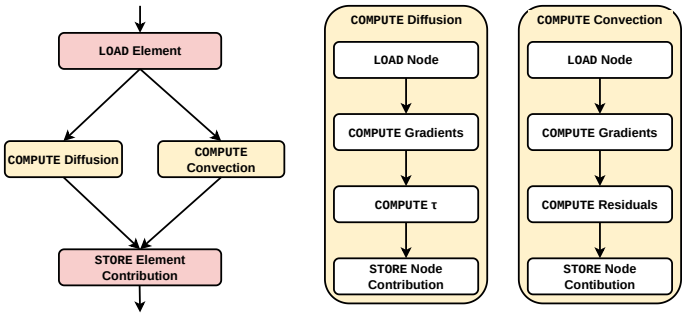

Fig. 1: Dataflow Graph of Core Computation

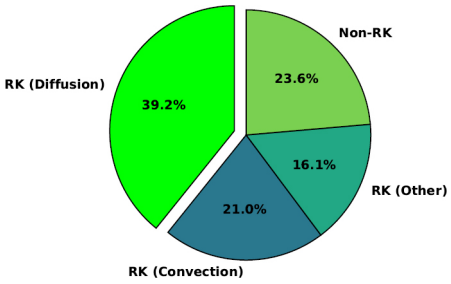

Fig. 2: Breakdown of Average Execution Time

$\frac{dy}{dt} = f(t, y), \quad y(t_0) = y_0$. The Runge-Kutta method [23] is a numerical approach used to solve ODEs when analytical solutions are difficult or impossible to obtain. It enhances simpler methods by offering more accurate approximations at each time step, typically by evaluating the slope at multiple points within the interval. Following the approach outlined in [14], we employed the fourth-order Runge-Kutta method (RK4), known for its effective balance between accuracy and computational efficiency.

C. Source Code Description & Characterization

In this section, we provide a brief overview of the source code developed using the numerical methods discussed in II-B to solve the problem described in II-A. The process begins with loading and preprocessing the discretized mesh.The main computation takes place within a time-stepping loop, where the algorithm proceeds through four stages of the Runge-Kutta (RK) method at each time step. During each stage, the algorithm computes the *Diffusion* and *Convection* terms based on FEM. As illustrated in Figure 1, the algorithm calculates the contribution of each grid element e to the diffusion and convection terms by loading the element's data, independently computing both terms, and storing the results for the next iteration. This process requires computations across the nodes of each element. First, the node data are retrieved, followed by the computation of the gradient, $\boldsymbol{\tau}$, and residuals. Finally, the node's contribution is stored. After each RK step, the algorithm updates the values of $\rho, \boldsymbol{u}, T, E$, and p. This procedure continues until the solution is computed for all time steps.

To pinpoint the most time-consuming parts, we performed a detailed profiling analysis across different input sizes, .i.e. number of mesh nodes, ranging from 1M to 4M. Figure 2 shows the average breakdown of execution time. As shown, the RK method was the most time-intensive, accounting for an average of 76.5% of the total execution time. Within the RK method, the diffusion and convection functions emerged as the primary hotspots, consuming 39.2% and 21.04% of the total execution time, respectively. Therefore, the entire RK method is amenable for acceleration, with particular emphasis on optimizing convection and diffusion. Similar profiling data have been also recently reported [4] targeting FEM-based multi-GPU CFD, further strengthening the validity of our results.

III. FEM Reconfigurable Accelerator Design

The following sections offer a detailed overview of the proposed accelerator architecture and the inter-task, memory and intra-task micro-architectural optimizations introduced for the RK method, i.e. the most computationally demanding component Section II-C. The remaining computations are handled by the host CPU. We utilize HLS for introducing the proposed accelerator architecture and optimizing its FPGA mapping.

A. Accelerator Architecture

An overview of the proposed accelerator is illustrated in Figure 3. The CPU is responsible for handling the initialization and overseeing the iteration process across the time steps. It is also responsible for transferring the necessary data via PCIe to the off-chip memory of the target FPGA. On the other hand, the accelerator is divided into two separate kernels, with each assigned to a different Super Logic Region (SLR). The *Runge-Kutta Loop* (RKL) kernel performs the core computations, while the *Runge-Kutta Update* (RKU) kernel evaluates ρ, $\boldsymbol{u}$, T, E, and p at every time step. RKL is assigned to the SLR with direct access to off-chip memory, while RKU connects to the same memory through the *Super Long Lines* (SLL) interconnect. Although SLL connections have higher latency compared to intra-SLR fabric connections [24], RKU is less time-consuming, and its data access patterns are far more regular than those in RKL. This RKL-RKU partitioning enables more effective utilization of FPGA's resources, by de-stressing the Place&Route phase from trading performance optimality for high resource utilization and routing congestion.

RKL is optimized to effectively carry out the core computations of the FEM-based CFD simulation. Specifically, for the computation of the *Diffusion* and *Convection* terms, the original source code was reorganized into a *Load-Compute-Store* form to exploit **Task Level Pipelining** (TLP). Initially, the data required for each element is transferred in batches from off-chip memory to the BRAMs and URAMs within the Programmable Logic (PL) ❶. The next step involves executing the computations for the *Diffusion* and *Convection* terms ❷. Since these computations share significant functionality, as shown in Figure 1, and no data dependencies are present, we combined these operations into a single module to improve hardware reuse during each element's computation. This stage uses the node data already stored in the PL ❷a, calculates the gradients of both terms, as well as the $\boldsymbol{\tau}$ and residuals ❷b, and finally stores the node's contribution to the total diffusion and

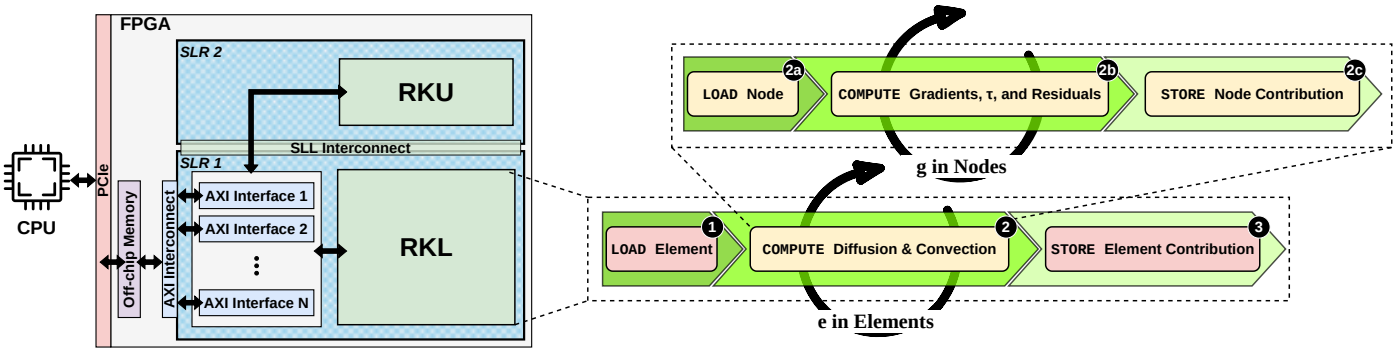

Fig. 3: Overview of the Proposed Accelerator's Architecture

convection calculations ❷c. Once the diffusion and convection computations for the current element are completed, the data is written back to off-chip memory ❸. Since the *Load*, *Compute*, and *Store* tasks are sequential, where each task produces data that the next one consumes, they can be pipelined across iterations to enhance performance. After completing the RKL computations for the current time step, the RKU evaluates ρ, $\boldsymbol{u}$, T, E, and p. Once completed, the iteration concludes, and the CPU begins the next one.

B. Task Level Pipelining

Utilizing *instruction-level optimizations*, such as loop pipelining and unrolling, is a standard approach when accelerating applications on FPGAs with High-Level Synthesis. However, relying solely on these optimizations often results in limited overall performance, resource over-utilization, and memory bandwidth bottlenecks, as they focus primarily on fine-grained loop-level improvements. For example, applying loop pipelining to the outer loop of a nested structure, which is common in most scientific computations, often requires fully unrolling the inner loops for the pipeline to function effectively, leading to kernels that may exceed the available resources of the FPGA. Moreover, these approaches do not exploit coarse-grained parallelism, which limits scalability and leaves substantial performance potential unutilized.

Task Level Pipelining (TLP) forms a key factor for effective *Dataflow Optimization*, and it is an effective approach for addressing these limitations, providing enhanced performance and improved resource efficiency. TLP involves partitioning the core computation into N sequential tasks, $Task_1, Task_2, \ldots, Task_N$, where each task passes data to the next through inter-task buffers, which can be either First-In-First-Out (FIFO) or Ping-Pong (PIPO) buffers. The N tasks form the TLP stages, with the most time-consuming task determining the Initiation Interval (II), which represents the number of clock cycles needed before the next iteration of the entire pipeline can begin. In a specific time step of the pipeline, the inter-task buffers temporarily store the data produced by $Task_k$. The data will then be processed by $Task_{k+1}$, while $Task_k$ runs concurrently the next iteration.

As illustrated in Figure 3, we identified two areas where TLP can be applied: (a) the element-wise computations (i.e., tasks ❶, ❷, and ❸), and (b) the node-wise computations within each element (i.e., tasks ❷a, ❷b, and ❷c). Noting that the *Diffusion* and *Convection* terms share considerable functionality and perform nearly identical computations, and with no data dependencies detected, we code-merged these similar operations into a single function/module to enhance hardware reuse. To apply TLP optimization effectively and prevent deadlocks, two key conditions were considered [25]. First, the *Single-Producer-Single-Consumer* rule was established to ensure that each task has a single producer providing data and a single consumer receiving it, facilitating smooth data flow and preventing conflicts. Second, it was ensured that inter-task buffers do not bypass any tasks and transfer data sequentially, thereby maintaining the integrity of the pipeline process. Meeting these conditions can be particularly challenging for complex applications like our CFD simulation and often requires extensive manual rewriting. Sections III-C and III-D provide a detailed examination of how we optimized off-chip memory reads within our tasks, followed by the fine-grained HLS directive optimizations for higher Instruction-Level-Parallelism (ILP).

C. Off-chip Memory Transfer Parallelization

This section discusses the two primary optimizations applied to the tasks in our pipeline that interact with off-chip memory. These optimizations aim to enhance memory throughput and prevent contention, which can hinder the performance of TLP.

Arrays to Memory Channel Assignment. The tasks depicted in Figure 3 contain loops that access off-chip memory through one or more AXI interfaces. These interfaces are connected to an AXI-Interconnect, which in turn connects to the off-chip memory. Each memory access, typically corresponding to an array element, must be explicitly mapped to a specific AXI interface. To minimize iteration latency in these loops, we schedule memory accesses concurrently by assigning them to separate AXI interfaces, as depicted in Figure 4. Figure 4 shows a code snippet of *Load-Element* task ❶, demonstrating how the respective AXI interface directives are applied to assign these memory accesses to different AXI interfaces. This approach eliminates interface contention, which would otherwise force the memory accesses to occur sequentially. However, at certain stages of the algorithm, the arrays to be transferred exceed the available AXI interfaces, making it impossible to assign each array individually. To overcome this limitation, we implement interface reuse for arrays accessed by different tasks during

```
...
void readDDR() {
  #pragma HLS INTERFACE MODE=m_axi BUNDLE=gmem_1 PORT=rho
  #pragma HLS INTERFACE MODE=m_axi BUNDLE=gmem_2 PORT=Tem
  #pragma HLS INTERFACE MODE=m_axi BUNDLE=gmem_3 PORT=mu_fluid
  #pragma HLS INTERFACE MODE=m_axi BUNDLE=gmem_4 PORT=E
  for (uint32_t inode=0; inode < NNODE; inode++) {
    rho_elem_type rho_temp = rho[inode];
    Tem_elem_type Tem_temp = Tem[inode];
    mu_fluid_elem_type mu_fluid_temp = mu_fluid[inode];
    E_elem_type E_temp = E[inode];
  }
}
...
```

Fig. 4: Individual AXI Interface Assignment Optimization

successive steps of the algorithm, such as the `LOAD-Element` and `STORE-Element-Contribution` tasks. Since these loops are not executed in parallel, this method ensures that arrays sharing the same interface do not compete for memory bandwidth, thereby optimizing data transfer efficiency.

Decoupling Memory Load and Store Interfaces: In the RK method, we often encounter loops iterating over arrays stored in off-chip memory, executing operations like $x[i] \leftarrow f(x[i], y[i])$, where f is the function applied to arrays x and y. These arrays retrieve data from off-chip memory through an AXI interface. Consequently, the same AXI interface is responsible for reading the values of x and writing back the updated results. This inter-iteration dependency hinders loop pipelining, ultimately slowing down the overall execution. To enable pipelined updates, we introduce an additional interface dedicated to x, where one interface handles reading and the other manages writing. This approach resolves the inter-iteration dependency, allowing for pipelined memory updates.

D. HLS Directives for TLP Initiation Interval Optimization

As outlined in Section III-B, the Initiation Interval (II) of TLP is determined by the most time-consuming task. In this section, we focus on reducing the TLP's II, and thus consequently, the overall execution time of the simulation. Since our goal is to reduce the TLP II, we focus on optimizing the task with the highest latency in an iterative manner, i.e. the HLS optimization directives are applied each time to the task exposing the highest latency criticality. Optimizing all available tasks could result in resource violations due to the limited capacity of the FPGA. Moreover, focusing on low-latency tasks would offer minimal performance improvements.

We prioritize tasks ❷a, ❷b, and ❷c for optimization since they are the most latency-critical. Furthermore, these tasks gain from processing data stored directly in the PL, where small matrices are housed in the 32KB BRAMs and larger matrices that surpass BRAM capacity are stored in the 288KB URAMs. The data is fetched through task ❶. More in detail, we primarily concentrate on optimizing intra-task micro-architecture, by applying three specific HLS directives: a) loop unrolling, b) loop pipelining, and c) array partitioning. After identifying in each step the most latency critical task, we examined HLS directive placement in an intra-task manner. Specifically, for each critical task identified in the current step, 1. we extract the for-loops with a high trip count and multiple operations in the loop body, 2. we examine potential inter-iteration dependencies and 3. we apply the loop pipelining directive. For these large loops, we did not perform unrolling, as this would duplicate the loop body by the factor used, resulting in high resource utilization. For the for-loops with small trip counts, we completely unrolled them based on the factors allowed by our available resources. To enable the parallel data accesses required by our directives, we also apply array partitioning with the appropriate factors. This procedure is repeated until no further optimization could be achieved, either due to unresolved dependencies or resource over-utilization, which would result in lower clock frequencies.

IV. Experimental Evaluation

We implement and evaluate the proposed accelerator architecture regarding performance, resources utilization, and energy efficiency. To convert the developed C++ simulation source and its optimizations into HDL, we utilized *Xilinx Vitis HLS 2021.1* and the *Xilinx Vitis Unified Software Platform 2021.1*.We chose the AMD Alveo U200 as the target FPGA. The Alveo U200 card includes 3 Super Logic Regions (SLRs) and 4 DDR memories, each with a capacity of 16GB. Communication with the host is enabled via PCIe and the Xilinx Runtime (XRT).

A. Comparison with Vitis-HLS Optimized Design

As an initial step, we compare the proposed accelerator with the Vitis-HLS optimized design. Recent Vitis-HLS release applies the following HLS directive as general optimization strategy: i) automatic loop pipelining using the flag c*onfig_compile -pipeline_loops*, ii) unrolling of small tripcount loops through *config_unroll -tripcount_threshold*, and iii) complete partitioning of small arrays using *config_array_partition -complete_threshold*. To assess the effect of input data size, we measure the total execution time of the computationally intensive components of our application, specifically RKU and RKL, for varying numbers of mesh nodes. As illustrated in Figure 5, increasing the number of nodes in the examined mesh results in longer execution times for the RK method in both baselines. Specifically, increasing the number of nodes from 1.4M to 4.2M results in a $3.4\times$ increase in execution time for both the proposed design and the Vitis-optimized version. The proposed approach consistently surpasses the Vitis optimization across all tested node counts, achieving an average improvement of $7.9\times$. The lower performance can be partially attributed to the Vitis-optimized kernel being restricted to a 100 MHz clock frequency, whereas the proposed design operates at 150 MHz. This limitation of the Vitis-HLS optimized design arises from both the *RKL* and *RKU* modules being mapped onto the same SLR, which caused significant routing congestion and restricted the maximum clock speed.

Regarding resource utilization, as shown in Table I, our optimized design leads to $1.5\times$ higher FF% and LUT%, a $1.9\times$ higher BRAM% and DSP%, and a $16.8\times$ higher URAM%, compared to Vitis-HLS optimized design. Although the increase in URAM usage is significant, i.e. Vitis-HLS treats URAM as scarce resource, the utilization of other resources shows no more than a two-fold increase compared to Vitis-HLS optimizations. This indicates that we achieve substantial performance gains with only minimal increases in resource utilization.

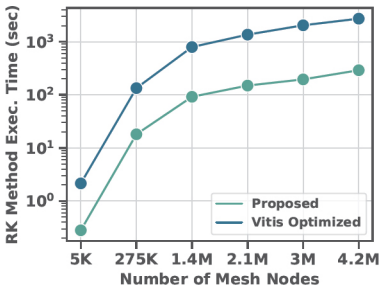

Fig. 5: Execution Time for Different Numbers of Mesh Nodes

Design	FF%	LUT%	BRAM%	URAM%	DSP%
Vitis Opt.@100MHz	17.19	27.68	22.96	0.73	9.17
Proposed@150MHz	25.29	41.15	43.98	11.77	18.23

TABLE I: Post P&R Resource Utilization Percentages

B. Comparison with Server CPU

We compared the proposed FPGA accelerated solution with its software implementation counterpart, i.e. the exact same C++ implementation running in single-threaded mode on a high-performance Linux server, specifically equipped with an Intel Xeon Silver 4210 CPU @ 2.20GHz with 32K L1D/I, 1M L2 and 14M L3 cache. This choice offers a more balanced comparison than alternatives often cited in the literature, such as the ARM Cortex-A53 [20], due to the Xeon's superior processing capabilities. For this comparison, we used a 4.2M node mesh, which closely represents a real-world scenario. Our design achieved a 45% reduction in execution time, demonstrating the effectiveness of the proposed solution. Another key metric showcasing the potential of our implementation is power consumption. The CPU implementation consumed an average of 120.42W across all test cases with varying mesh sizes. In contrast, the FPGA averaged 32.4W for the core application, with an additional 30.7W for peripherals and 1.7W for the rest of the system, resulting in an average power consumption that is 3.64× lower than the CPU. These initial results highlight the potential of reconfigurable accelerators for efficiently accelerating FEM-based simulations, a field that remains unexplored.

V. Conclusion

In this work, we presented a high-performance FPGA accelerator tailored for numerically solving the Navier-Stokes equations, with a focus on FEM due to its capability to accurately represent complex geometries and intricate real-world scenarios. The proposed accelerator, implemented with HLS on AMD Alveo U200 FPGA, delivers 7.9× better performance than the Vitis-HLS optimized version, while compared with its software implementation counterpart running on a high-end server it reduces latency by 45% while consuming 3.64× less power. This underscores the potential of our approach for solving the Navier-Stokes equations more efficiently, paving the way for addressing even more challenging CFD simulations.

References

[1] P. R. Spalart and V. Venkatakrishnan, "On the role and challenges of CFD in the aerospace industry," *The Aeronautical Journal*, vol. 120, no. 1223, pp. 209–232, 2016.

[2] T. Kobayashi and K. Kitoh, "A review of CFD methods and their application to automobile aerodynamics," 1992.

[3] C. F. Janßen, D. Mierke, M. Überrück, S. Gralher, and T. Rung, "Validation of the GPU-accelerated CFD solver ELBE for free surface flow problems in civil and environmental engineering," *Computation*, vol. 3, no. 3, pp. 354–385, 2015.

[4] K. Koliogeorgi, G. Anagnostopoulos, G. Zampino, M. Sanchis, R. Vinuesa, and S. Xydis, "Auto-tuning Multi-GPU High-Fidelity Numerical Simulations for Urban Air Mobility," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, IEEE, 2024.

[5] R. K. Raman, Y. Dewang, and J. Raghuwanshi, "A review on applications of computational fluid dynamics," *International Journal of LNCT*, vol. 2, no. 6, pp. 137–143, 2018.

[6] D. E. Keyes, L. C. McInnes, C. Woodward, W. Gropp, E. Myra, M. Pernice, J. Bell, J. Brown, A. Clo, J. Connors, *et al.*, "Multiphysics simulations: Challenges and opportunities," *The International Journal of High Performance Computing Applications*, vol. 27, no. 1, pp. 4–83, 2013.

[7] T.-P. Tsai, *Lectures on Navier-Stokes equations*, vol. 192. American Mathematical Soc., 2018.

[8] T. Liszka and J. Orkisz, "The finite difference method at arbitrary irregular grids and its application in applied mechanics," *Computers & Structures*, vol. 11, no. 1-2, pp. 83–95, 1980.

[9] C. A. Felippa, "Introduction to finite element methods," *University of Colorado*, vol. 885, 2004.

[10] J. Vos, A. Rizzi, D. Darracq, and E. Hirschel, "Navier–Stokes solvers in European aircraft design," *Progress in Aerospace Sciences*, vol. 38, no. 8, pp. 601–697, 2002.

[11] P. Fischer, J. Lottes, and H. Tufo, "Nek5000," tech. rep., Argonne National Laboratory (ANL), Argonne, IL (United States), 2007.

[12] B. Asgari, R. Hadidi, T. Krishna, H. Kim, and S. Yalamanchili, "Alrescha: A lightweight reconfigurable sparse-computation accelerator," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 249–260, IEEE, 2020.

[13] Q. Zhu, "FreeStencil: A Fine-Grained Solver Compiler with Graph and Kernel Optimizations on Structured Meshes for Modern GPUs," in *Proceedings of the 53rd International Conference on Parallel Processing*, pp. 1022–1031, 2024.

[14] L. Gasparino, F. Spiga, O. Lehmkuhl, "SOD2D: A GPU-enabled Spectral Finite Elements Method for compressible scale-resolving simulations,"

[15] T. Chen, J. Botimer, T. Chou, and Z. Zhang, "A 1.87-mm 2 56.9-GOPS accelerator for solving partial differential equations," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 6, pp. 1709–1718, 2020.

[16] J. Mu and B. Kim, "29.2 a 21× 21 dynamic-precision bit-serial computing graph accelerator for solving partial differential equations using finite difference method," in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64, pp. 406–408, IEEE, 2021.

[17] J. Mu, C. Yu, T. T.-H. Kim, and B. Kim, "A scalable bit-serial computing hardware accelerator for solving 2D/3D partial differential equations using finite difference method," in *ESSCIRC 2022-IEEE 48th European Solid State Circuits Conference (ESSCIRC)*, pp. 353–356, IEEE, 2022.

[18] J. Li, Y. Zhang, H. Zheng, and K. Wang, "FDMAX: An elastic accelerator architecture for solving partial differential equations," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pp. 1–12, 2023.

[19] B. Zahiri, "Structured ASICs: opportunities and challenges," in *Proceedings 21st International Conference on Computer Design*, pp. 404–409, IEEE, 2003.

[20] F. A. K. Friebel, S. Soldavini, G. Hempel, C. Pilato, and J. Castrillon, "From domain-specific languages to memory-optimized accelerators for fluid dynamics," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, (Los Alamitos, CA, USA), pp. 759–766, IEEE Computer Society, sep 2021.

[21] J. DeBonis, "Solutions of the Taylor-Green vortex problem using high-resolution explicit finite difference methods," in *51st AIAA aerospace sciences meeting including the new horizons forum and aerospace exposition*, p. 382, 2013.

[22] O.C. Zienkiewicz, CBE, FRS, R.L. Taylor, J.Z. Zhu, *The Finite Element Method: Its Basis and Fundamentals*.

[23] J. H. Cartwright and O. Piro, "The dynamics of Runge–Kutta methods," *International Journal of Bifurcation and Chaos*, vol. 2, no. 03, pp. 427–449, 1992.

[24] Z. Di, R. Tao, J. Mai, L. Chen, and Y. Lin, "LEAPS: Topological-Layout-Adaptable Multi-Die FPGA Placement for Super Long Line Minimization," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2023.

[25] Xilinx, Inc, *Vitis High-Level Synthesis User Guide (UG1399)*, 2021.1 ed.