Automatic Creation of High-Bandwidth Memory Architectures from Domain-Specific Languages: The Case of Computational Fluid Dynamics

STEPHANIE SOLDAVINI, Politecnico di Milano, Italy
KARL F. A. FRIEBEL, Technische Universität Dresden, Germany
MATTIA TIBALDI, Politecnico di Milano, Italy
GERALD HEMPEL, Technische Universität Dresden, Germany
JERONIMO CASTRILLON, Technische Universität Dresden, Germany
CHRISTIAN PILATO, Politecnico di Milano, Italy

Numerical simulations can help solve complex problems. Most of these algorithms are massively parallel and thus good candidates for FPGA acceleration thanks to spatial parallelism. Modern FPGA devices can leverage high-bandwidth memory technologies, but when applications are memory-bound designers must craft advanced communication and memory architectures for efficient data movement and on-chip storage. This development process requires hardware design skills that are uncommon in domain-specific experts. In this paper, we propose an automated tool flow from a domain-specific language (DSL) for tensor expressions to generate massively-parallel accelerators on HBM-equipped FPGAs. Designers can use this flow to integrate and evaluate various compiler or hardware optimizations. We use computational fluid dynamics (CFD) as a paradigmatic example. Our flow starts from the high-level specification of tensor operations and combines an MLIR-based compiler with an in-house hardware generation flow to generate systems with parallel accelerators and a specialized memory architecture that moves data efficiently, aiming at fully exploiting the available CPU-FPGA bandwidth. We simulated applications with millions of elements, achieving up to 103 GFLOPS with one compute unit and custom precision when targeting a Xilinx Alveo U280. Our FPGA implementation is up to 25× more energy efficient than expert-crafted Intel CPU implementations.

1 INTRODUCTION

Numerical simulations are computationally-intensive applications that are used to solve many complex problems in industry [47]. These data-intensive applications are massively parallel since they use a combination of tensor operators to compute smaller and independent contributions that operate on different data to compose the final result [56]. In this context, **high-performance computing (HPC)** is a powerful tool to reduce the costs of testing while giving the possibility of exploring more solutions. HPC solutions can provide high-resolution physics results for many domains, including molecular dynamics [10] or weather simulations [52].

In this context, FPGA devices are increasingly used to achieve energy-efficient high performance by exploiting spatial parallelism with specialized accelerators. They are thus good candidates for

Authors' addresses: Stephanie Soldavini, Politecnico di Milano, Milan, Italy, stephanie.soldavini@polimi.it; Karl F. A. Friebel, Technische Universität Dresden, Dresden, Germany, karl.friebel@tu-dresden.de; Mattia Tibaldi, Politecnico di Milano, Milan, Italy, mattia.tibaldi@polimi.it; Gerald Hempel, Technische Universität Dresden, Dresden, Germany, gerald.hempel@tu-dresden.de; Jeronimo Castrillon, Technische Universität Dresden, Dresden, Germany, jeronimo.castrillon@tu-dresden.de; Christian Pilato, Politecnico di Milano, Milan, Italy, christian.pilato@polimi.it.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Association for Computing Machinery.
1936-7406/2022/9-ART $15.00
https://doi.org/XXXXXXX.XXXXXXX

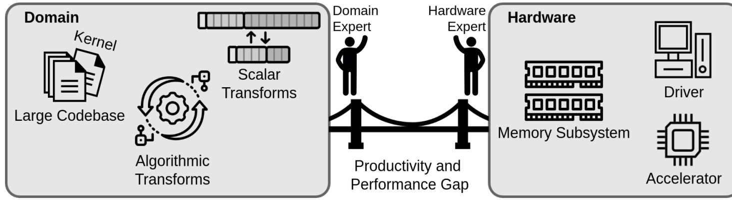


Fig. 1. Our flow can help bridge the gap between domain and FPGA experts by providing a framework to match software and hardware requirements.

accelerating numerical simulations. Embedded FPGA devices can provide parallel architectures that overcome embedded processors, but are limited in terms of resources and memory bandwidth [16]. So, they cannot complete with large HPC data centres. Instead, modern FPGA data center cards offer advanced **high-bandwidth memory (HBM) architectures** with multiple high-speed memory channels that enable efficient and parallel data transfers between the CPU and the reconfigurable logic [63, 26]. HBM is a modern memory technology that can offer a bandwidth of hundreds of Gigabytes per second [25]. For example, both Intel and Xilinx offer HBM solutions in their FPGA devices: Intel Stratix 10 MX FPGAs include HBM2 with 16GB of data that can be accessed up to 409 GB/s, while Xilinx Alveo FPGA cards offer 8GB of HBM2 at 460 GB/s bandwidth. Such platforms enable the acceleration of memory-bound applications. Additionally, designers can use **high-level synthesis (HLS)** tools to raise the abstraction level of hardware design [38, 30]. However, designing efficient architectures for such systems is complex as it requires a concurrent optimization of communication, computation, and storage [40]. These optimizations may be limited by platform constraints, like the physical architecture, which can make the routing stages more difficult, or the number of physical resources, which can limit the number of parallel executions (see Section 2.3 for further details on these kinds of challenges).

Application and hardware developers face orthogonal challenges in fully exploiting HBM architectures [12]. In Figure 1, we highlight some key problems addressed in this work on both sides, which create a gap between the experts in terms of:

- **productivity**: application designers usually have limited knowledge on hardware design and cannot create efficient hardware architectures to fully exploit the available FPGA technology. For this reason, they prefer to use DSL descriptions to abstract the semantics of their operators;
- **performance**: coordinating data transfers and execution requires an intimate knowledge of both the application and the target platform that is uncommon in many developers In this case, fine-tuned hardware descriptions are required to overcome these challenges.

On the architecture side, HBM poses several challenges. They have many parallel memory channels, but the overall number is limited (up to 32 in modern Xilinx Alveo cards). The communication cost between host and device memories is expensive: large CPU-FPGA data transfers can be much longer than the computation time of the kernel offloaded to the FPGA fabric.

In this paper we look into computational kernels that are composed of tensor operations, using CFD as a paradigmatic example. Such tensor expressions are common across domains, including fluid dynamics, quantum chemistry, deep learning, image processing and data analytics [44], and put high pressure on FPGA resources (especially DSP and BRAM). These resources are finite and they can limit the deployment of parallel kernels. The designer must carefully trade-off kernel optimizations to achieve the best target architecture. In addition, the number of possible implementations of tensor expressions (see [6] for a distribution of the speedup for simple loop programs) only exacerbates this problem. Finally, accelerators demand efficient data movement to exchange data with the off-chip memory, but the designer needs to exploit the available bandwidth. Optimizing HLS code

to maximize the performance is thus a complex task that requires the modeling and exploration of different solutions at both software and hardware levels [54].

To address these challenges and support the designer in the optimization process, we propose a **DSL-to-bitstream workflow** composed of: (1) an **MLIR-based DSL compiler** that abstracts low-level specification details (for enhancing designer's productivity by identifying the most suitable HLS-ready code to exploit hardware parallelism) and generates HLS-optimized codes for the kernels that enable us to deploy multiple hardware modules and to transfer the data more efficiently; and (2) an **automated HLS-based flow** that exploits this information to design architectures that can leverage the HBM subsystem for efficient data transfers (for enhancing performance results). **Our workflow allows application designers to describe their applications in a high-level, domain-specific, and platform-agnostic language and use an automated toolchain to create the corresponding system architecture that exploits the intrinsic parallelism and the characteristics of HBM systems.** The designer is able to select and apply various optimizations, enabling a non-FPGA-expert to evaluate several alternatives. This paper extends the work presented in [16] to target FPGA data center cards with HBM architectures. Our novel flow is based on multi-level intermediate representation (MLIR) [53] to progressively lower the specification without losing semantic information. The rest of the flow creates the memory architecture (in C++) around the accelerator kernels, leveraging commercial HLS to generate the hardware description and, in turn, the FPGA bitstreams. Our main contributions are:

- we present a **compiler infrastructure based on MLIR** for a DSL for tensor operations to automatically generate HLS-ready code for the computational kernels;
- we describe an **HBM-oriented hardware generation flow** that interfaces with commercial HLS to generate an optimized system architecture;
- we show how our flow can help an application designer in **optimizing the *Inverse Helmholtz* operator**, a key element of CFD simulations. This complex operator subsumes other widely-used tensor operators, like tensor contraction and tensor-based interpolation.

With our flow, the designer can generate, evaluate, and compare several alternatives, for example, trading off accuracy of the results and performance. Our analysis also provides useful guidelines for designers to understand what to do (or not to do) when implementing similar tensor-based applications on HBM-based systems, especially when aiming at scaling up the computation with multiple computing units that execute in parallel.

2 BACKGROUND

In this section, we present the main concepts at the basis of our work. We first introduce our target application (Section 2.1), highlighting that it is representative of other similar numerical applications. We then describe the characteristics and the challenges of our target platform (Section 2.2 and Section 2.3, respectively). We conclude with an analysis of the related work (Section 2.5).

2.1 CFDlang DSL for Spectral Element Methods

In numerical mathematics, spectral element methods (SEM) are common in solving partial differential equations (PDEs), like the Navier-Stokes equations [35], which cannot be solved analytically. SEM approximates the solution by transforming the unknown physical quantities of the problem into spectral coefficients. To reduce the numerical complexity, the simulated volume is divided into N_{eq} smaller **elements**. To further reduce the error, SEM uses an approximation based on polynomials of a higher degree ($p > 1$). The solution is expressed as a linear system of equations which can be solved locally for each element. An element solution e can be represented in three dimensions as a tensor $v_{ijk,e}$ with $i, j, k \in \{0, \ldots, p\}$. Often, the polynomial degree p is the same

```
1 var input S    : [11 11]
2 var input D    : [11 11 11]
3 var input u    : [11 11 11]
4 var output v   : [11 11 11]
5 var t          : [11 11 11]
6 var r          : [11 11 11]
7 t = S#S#S#u . [[1 6][3 7][5 8]]
8 r = D * t
9 v = S#S#S#r . [[0 6][2 7][4 8]]
```

Fig. 2. DSL code for the Inverse Helmholtz operator ($p = 11$).

for all spatial dimensions. All elements operate on independent tensors and can be elaborated in parallel.

In our example, we focus on solving the Helmholtz equation $\lambda u - \nabla^2 u = f$ for quadrilateral elements. The Inverse Helmholtz operator subsumes simpler operators (e.g., interpolation) that are similarly relevant in CFD simulations [23]. The operator can be formulated as:

$$t_{ijk,e} = \sum_{l=0}^{p} \sum_{m=0}^{p} \sum_{n=0}^{p} S_{li}^{T} \cdot S_{mj}^{T} \cdot S_{nk}^{T} \cdot u_{lmn,e} = (S \otimes S \otimes S \otimes u_e)_{iljmkn}{}^{lmn} \tag{1a}$$

$$r_{ijk,e} = D_{ijk,e} \cdot t_{ijk,e} \tag{1b}$$

$$v_{ijk,e} = \sum_{l=0}^{p} \sum_{m=0}^{p} \sum_{n=0}^{p} S_{li} \cdot S_{mj} \cdot S_{nk} \cdot r_{lmn,e} = (S \otimes S \otimes S \otimes r_e)_{limjnk}{}^{lmn} \tag{1c}$$

The CFDlang DSL for tensor operations [44] enables us to concisely encode these kinds of operators. We have chosen this particular DSL because of its excellent domain capture and limited optimization scope. CFDlang is target-agnostic and its user interface is close to the mathematical problem specification, reducing the programming burden on application developers. Figure 2 shows a CFDlang program implementing the Inverse Helmholtz operator, where lines 7-9 are the direct transcriptions of the expressions (1a), (1b) and (1c). The CFDlang description does not define exact implementation details, allowing the compiler to optimize the operations for a given target. Another implicit part of the given DSL program is that it is assumed to be applied to all the independent elements in an implicit outer "element loop".

From the application developer viewpoint, once the kernel is specified in CFDlang, it can be integrated into larger Fortran or C++ code applications via their respective foreign function interface (FFI) mechanisms where we can embed also the FPGA runtime library calls.

Previous work found that typical software implementations achieve performances between 1 and 16 GFLOPS based on the polynomial degree p, with an average power consumption of at least 100W [44]. GPUs, in turn, do not feature as good a scaling behavior for these kernels [24].

2.2 HBM-based Platform Description

HBM is a novel memory architecture that enables high-performance and adaptability for memory-bound applications [17]. HBM is a **3D-stacked DRAM** that offers high-bandwidth and energy-efficient data movements. The logic die is connected to the HBM die(s) with through-silicon vias (TSVs). The rest of this work focuses on the **Xilinx Alveo U280** data center accelerator card. However, other HBM-based FPGA devices (like the Intel Stratix 10 MX) are conceptually similar [12]. The Alveo U280 is built on the Xilinx 16nm UltraScale+ architecture and offers a rich set of memory solutions, as shown in Figure 3. The Alveo U280 card features the XCU280 FPGA, which combines three super logic regions (SLRs). An SLR is a physical section of the FPGA with a specific amount of resources and connections, as shown in Table 1. SLR0 integrates an HBM controller to interface with the HBM2 subsystem through 32 **pseudo-channels (PCs)** each with

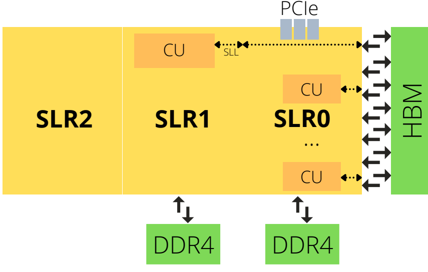


Fig. 3. Architecture of the XCU280 device.

Table 1. Alveo U280 SLR resources.

Resources	SLR0	SLR1	SLR2
HBM	32×256MB	-	-
DDR4	16GB	16GB	-
PLRAM	2×4MB	2×4MB	2×4MB
CLB LUT	369K	333K	367K
CLB Register	746K	675K	729K
Block RAM tile	507	468	512
UltraRAM	320	320	320
DSP	2,733	2,877	2,880

direct access to 256 MB of storage (8 GB in total). Each 256-bit PC operates at 450 MHz, yielding a maximum bandwidth of 14.4 GB/s. The full system can thus achieve a theoretical bandwidth of 460.8 GB/s. SLR0 also connects to the host via 16 lanes of the PCI Express (PCI-e) interface. SLR0 and SLR1 each connect to 16 GB of DDR4 each. Finally, each region has up to 8 MB of PLRAM for fast access to small data sets. In the following, we will use the term **global memory** for the set of memories available on the board. The host must transfer data into the device global memory before they can be accessed by the FPGA logic.

The target system for the Alveo U280 is composed of multiple **compute units (CUs)**. Each CU is a user-defined hardware module that can be attached to any of the PCs through independent AXI interfaces, while the built-in HBM controller and switch have access to all physical channels. The CU can be described in C++ and synthesized with HLS or specified directly in RTL. Multiple CUs allow parallel execution but must be connected to different HBM channels. The **system configuration file** describes the connections between the CU ports and the HBM channels. The required logic is automatically generated during system synthesis.

Xilinx offers a **unified software platform**, called Vitis, to develop FPGA applications. Vitis includes a rich set of hardware-accelerated open-source libraries optimized for Xilinx FPGA and the Xilinx Runtime library (XRT) to facilitate communication between the host application (running on the host CPU) and the accelerator deployed on the reconfigurable portion of the card, which is connected via PCI-Express. It also includes user-space libraries and APIs, kernel drivers, and board utilities that can be used to measure performance and monitor power consumption. In this work, we aim at automating the generation of CU descriptions and the associated configuration file directly on top of the existing Xilinx libraries.

2.3 Challenges for Efficient HBM-Optimized Designs

In modern FPGA data center cards like the Xilinx Alveo U280, DDR4 memory is excellent for having access to large data sets with modest latency, but the transfer bandwidth is limited to 36 GB/s and no more than two parallel accesses [22]. Conversely, HBM has slightly higher latency, but it can reach a much higher theoretical bandwidth (460.8 GB/s) when fully using all its 32 PCs [12, 50]. HBM-based architectures are becoming popular for memory-bound applications [26, 7]. They have been used for convolutional neural networks, graph analytics, or weather-prediction [60, 33, 48, 51]. These platforms also feature PLRAM that can be used for small amounts of frequently-accessed data. In the case of the Alveo U280, the designer can configure these connections in the system configuration file. However, several factors can affect the performance of the resulting system.

Challenge 1: CPU-Host Communication Cost. Moving data across the PCI-e interface to the global memory has high cost compared to the data access from the logic regions to the global memory. So, such movements must be optimized to reduce their latency. First, the designers must move enough data to perform significant computation, for example by operating on multiple

elements in sequence before sending back the results. In many cases, the kernel needs to read the same data multiple times. For example, the Inverse Helmholtz operator must read matrix S several times during the two tensor contractions. To avoid multiple global memory accesses to the same data, designers can use internal buffers that gather enough data.

Challenge 2: Read/Write Burst Transactions. Vitis can automatically infer the size of data transfers to create more efficient burst transactions. However, frequently switching between read and write transactions is inefficient due to memory controller timing parameters. So, transactions should be ordered to maximize the data movement in one direction before switching to move data in the other direction. The alternative is to partition reads and writes into separate HBM channels, at the cost of increasing the number of channels required for each CU.

Challenge 3: Full Bandwidth Utilization. The AXI interfaces to the HBM channels can be configured with wider data busses. For example, the designer can use a bus width of 256 bits running at 450 MHz or even 512 bits running at 225 MHz. This allows the CU to exchange more data in each clock cycles, reducing the read/write cost from/to the HBM channels. For example, when configuring the 32 parallel HBM channels with a bus width of 256 bits, the potential bandwidth between the CPU and the FPGA is about 460 GB/s at 450 MHz. However, such data must be processed efficiently and in parallel to avoid performance bottlenecks [26].

Challenge 4: HBM-Data Allocation. To avoid congestion in the switch, it is important to partition the data into the different HBM memory regions such that each CU uses as few channels as possible, and shares these channels with as few other CUs as possible. Otherwise, the designer needs to introduce an efficient crossbar to mitigate congestion [11].

Challenge 5: Synthesis-Related Issues. FPGA devices include hard macros like DSP and BRAM for efficient computation and storage, respectively. However, the physical location of these FPGA resources can affect routing. Also, the XCU280 FPGA, which is the device at the basis of the Alveo U280, is partitioned into three SLRs, but only SLR0 is connected to the HBM channels (cf. Section 2.2). When the CUs cannot fit into a single SLR, they are automatically split over multiple SLRs using special resources called super long line (SLL) routes. These resources allow routing between CUs in any SLR, enabling access to all memory resources, but they introduce performance overhead. So, if a CU requires access to the HBM, it should be allocated in SLR0. Also, hardware modules that access multiple HBM channels should be close together to reduce wirelength.

2.4 MLIR

MLIR [32] is a recent compiler infrastructure that allows defining custom abstractions and high-level transformations. It has received considerable traction as a framework to develop domain-specific compilers for heterogeneous systems, especially for machine learning frameworks. MLIR is in itself not a fixed intermediate representation (IR), but an infrastructure with a central plug-in mechanism to extend the vocabulary of the IR by means of *dialects*. A dialect implements an intermediate abstraction as a set of operations, types, and attributes. Examples of prominent dialects include `linalg` for linear algebra operations, `affine` for nested loop programs, and the `llvm` dialect which enables a seamless transition into the LLVM compiler framework at the lowest level of abstraction. Custom dialects allow compiler designers to develop analyses and transformations at the right level of abstraction. Dialects can be integrated into larger language stacks, fostering reuse of abstractions and code transformations. A typical way of integrating a dialect is by means of *lowering*, where a more abstract dialect is converted into a more concrete one through specific transformations. Arguably the most complete flow built on top of MLIR to date is presented in [58], where authors describe how MLIR allows for a decentralized and composable compiler.

In this paper we leverage MLIR to create a composable and reusable domain-specific compiler. We implement the abstractions for tensors described in Section 2.1 and propose a lowering

pipeline that integrates with existing dialects in MLIR, while exposing profitable transformations for HBM-optimized designs. We also show how hardware-specific abstractions, e.g., for number representations, are integrated into the compilation flow.

2.5 Related Work

As FPGA devices are entering into data centers, HBM architectures are becoming extremely popular to accelerate a variety of data-intensive applications [25]. Benchmarking these architectures defined that HBM provides higher bandwidth but also higher access latency than traditional DDR [60, 22, 36, 34]. Instead, comparing FPGA performance with both CPUs and GPUs in HPC workloads concluded that there is potential in this technology. They also point out that although FPGAs struggle to compete in absolute terms with GPUs, they achieve much greater energy efficiency [7, 39, 28]. [17] and [11] argue that one of the main problems in FPGA is the absence of a comprehensive memory hierarchy, like the one in CPUs. For this reason, they propose two different custom crossbar solutions that efficiently access the HBM and create an additional memory layer using the BRAMs available inside the FPGA. However, these solutions restrict the amount of BRAMs available to implement the user's application and they do not fully exploit the FPGA pipeline parallelism. Other works like [21, 12, 51] define a series of guidelines to achieve high performance. Specifically, the designer must always instantiate the maximum number of computing units to exploit HBM parallelism, use 256-bit packets to maximize the bandwidth of the HBM AXI port, exploit the algorithm parallelism through hardware pipelining, and finally execute the task in a dataflow manner [26]. However, none of these works evaluated these optimizations together, along with the effects when scaling the number of computing units. None of the papers presented so far defines methodologies at the HLS level that can be applied to design to improve its performance, especially for tensor applications. [13] proposes three buffer restructuring approaches, coarse-grained, fine-grained, and hybrid, to adjust the BRAM and LUT utilization as needed. Similar, tensor optimizations were proposed in [54]. [43] studies the effect of HLS pragma directives on the design. In particular, they indicate that loop unrolling and array partitioning directives can cause overhead in off-chip memory performance due to non-burst access. Finally, they provide a code transformation technique to exploit the data width of the memory controller. The Merlin compiler[1] provides an infrastructure for source-to-source transformation to accelerate applications on FPGA that are already described in C++ [14]. Instead, operating at the DSL level allows us to implement high-level transformations to the source code, including exploring different memory layouts.

The core of our DSL abstraction consists in modelling tensor expressions. There are several such DSLs, like TensorComprehensions (TC) [59] and TensorFlow Eager [2], that are capable of representing tensor operations in a more general domain. However, many are based on backing software libraries, such as TensorFlow [1] and Theano [5], which are less flexible in terms of their back-end. TVM [9] is a compiler with a focus on machine learning (ML) that is not as limited in that regard, even providing an FPGA back-end. Its focus on ML makes it hard to apply TVM to other application domains. A similar application-specific flow is provided by ESP4ML [19], which explores a bigger design space of systems-on-chip (SoCs) using HLS but does not provides automatic generation from tensor expressions. More generally applicable, the Halide [42] ecosystem provides extensive support for heterogeneous systems. Spatial [27] provides automatic compilation and system generation from high-level descriptions. The framework offers high-level communication libraries but the application development burden is still on the designer.

Thanks to MLIR [53], as a framework for designing intermediate languages, several projects have recently proposed programming flows for heterogeneous systems. In the MLIR infrastructure,

[1] https://github.com/Xilinx/merlin-compiler

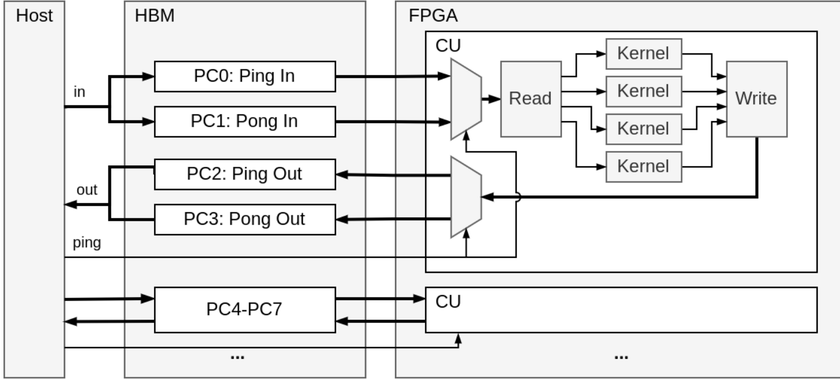


Fig. 4. Target system for the massively-parallel applications, like computational fluid dynamics (CFD).

compiler internals are composed using a plug-in system, which allows connecting a variety of front-, middle- and back-ends, which may be entirely orthogonal. For example, with Teckyl[2], the aforementioned TC has received a front-end for the MLIR compiler infrastructure, enabling a variety of back-end consumers. As an alternative to just lowering to LLVM [31], IREE[3] offers perhaps the first functional end-to-end flow entirely designed within MLIR. Closer to our domain is the recent Open Earth Compiler [20] for efficient GPU code generation for stencil operations. We also rely on the plug-and-compile MLIR philosophy, which will allow others to reuse our abstractions and allows us to profit from existing language stacks and compilation flows.

In the context of MLIR, projects such as Polygeist [37] and Phism [65] use polyhedral modelling to ingest, optimize, and even emit C code for HLS. Our compiler still employs polyhedral analysis and rescheduling as described in [16]. Streaming implementations are a challenge in FPGA implementations, which can tie into the polyhedral model via consecutivity constraints [62].

In conclusion, to address productivity issues in scientific computing, designers need a comprehensive framework that 1) allow them to use high-level tensor expressions to specify the behavior, 2) automate the hardware generation process on top of commercial HLS solutions, and 3) efficiently target modern HBM architectures.

3 AUTOMATIC CFDLANG-TO-BITSTREAM FLOW

This section describes our proposed approach. We first discuss our envisioned target architecture to implement CFD applications on HBM architectures (Section 3.1). We then outline our CFDlang-to-Bitstream flow (Section 3.2), followed by details on the two major components: the CFDlang compiler (Section 3.3 and Section 3.4) and the hardware generation flow (Section 3.5 and Section 3.6).

3.1 Target System Architecture

To simulate the complete volume, the CFD application applies the *Inverse Helmholtz* operator to N_{eq} independent elements. To exploit such intrinsic parallelism and address the challenges discussed in Section 2.3, we aim at building a target system like the one in Figure 4.

From the system-level perspective, the CFD simulation runs on the host CPU, which sends the data to the FPGA HBM via PCIe. Once the data are in the HBM, the compute unit (CU) can fetch them in parallel through AXI channels for several simulation elements. Since each HBM interface is 256 bits wide, the channel can be conceptually divided into multiple lanes based on the data

[2]https://github.com/andidr/teckyl
[3]https://google.github.io/iree/

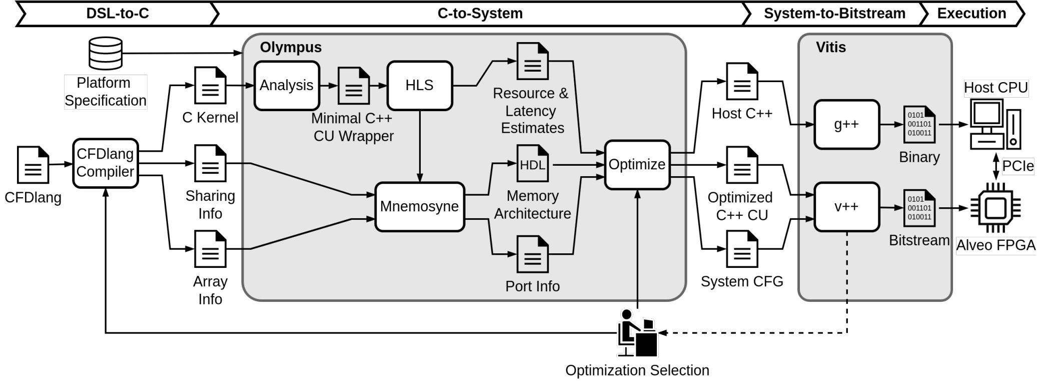


Fig. 5. Tool flow from CFDlang to FPGA bitstream generation.

bitwidth. Each lane can serve an independent accelerator, called *Kernel*, derived from the high-level DSL description, i.e., the Inverse Helmholtz operator of Figure 2. The CU is the fundamental unit that can be directly attached to the HBM channels and can be internally composed of several kernels with a wide range of communication patterns depending on the high-level description of the functionality to be implemented. To manage data exchanges between the HBM interfaces and the parallel kernels, the CU uses two additional hardware modules, i.e., *Read* and *Write* in Figure 4, that execute in parallel to the kernels and communicate with a dataflow model. The *Read* module fetches the input data of one element (matrices S, D, and u) from the HBM memory into internal buffers, while the *Write* module transfers the corresponding results (matrix v) into the HBM memory. To avoid conflicts and reduce the complexity of the AXI interfaces, we assign these modules to independent HBM channels. The data read and write modules within the CU split the 256-bit HBM data into 32-bit or 64-bit data for the computation modules based on the data format.

To hide the communication cost, the host computes the maximum number of sets of input data that can fit in one HBM channel. We thus define a **batch** as the number of elements E that the CU kernels will elaborate before the host retrieves the output. Executing a batch of several elements maximizes the size of the host data transfers, minimizing their associated overhead. In this way, the data exchanges between the CPU and the global memory can be overlapped with the CU execution using a double-buffer method. While the host is exchanging data with the *Ping* (*Pong*) channel, CUs can operate on the *Pong* (*Ping*) channel. Given the CU structure, we can compute the number of batches $N_b = N_{eq}/E$ to be executed based on the total number of elements N_{eq} to be simulated.

To further exploit parallelism, we can then replicate the CU structure multiple times, each of them operating on independent HBM channels. Let N_{cu} be the number of CUs that can be instantiated based on the resource constraints, the host application executes $I = N_b/N_{cu}$ iterations, evenly distributing the number of batches over the available CUs.

In the following, we describe our DSL-to-bitstream flow to automatically create this architecture on top of the existing HLS platform (e.g., Vitis for the Alveo U280). We also describe how to automate the integration of the optimizations to increase performance and resource efficiency, and how to modify the host code to match the memory layout required by the hardware modules.

3.2 Overview of the Proposed Flow

We propose a modular tool flow that simplifies the creation of FPGA accelerators for tensor-based applications, like numerical simulations, that are expressed in domain-specific languages. Figure 5 shows an overview of our flow which is composed of two major steps. We start from the DSL

description of the kernel and generate the corresponding optimized and HLS-friendly C code (**DSL-to-C generation**). The compiler performs a series of transformations at different levels of abstraction (cf. Section 3.4), inserts pragmas to guide the HLS flow, and produces meta data for the memory generation. Starting from the C code of the kernel, we create the corresponding parallel system with multiple CUs (cf. Section 3.5), each of them connected to one or more PCs (**C-to-system generation**). Also, each CU can instantiate one or more kernels based on the required amount of FPGA resources (cf. Section 3.6), along with the logic to move data from the global memory to the kernel on-chip buffer (*Read* and *Write* modules). We specify the CU functionality in C++, wrapping the kernel code with specific code and HLS directives to enable the generation of the memory-optimized architecture. To do so, we use a combination of distinct tools: a redesign of the **CFDlang** compiler (cf. Section 3.3), which integrates novel kernel-level transformations, the **Mnemosyne** tool [41], which improves memory sharing inside the single kernel to reduce on-chip memory requirements, and the newly-developed **Olympus** tool, which generates the system architecture (i.e., CU description and configuration file) around the kernels based on the optimization requested by the designer. Olympus also generates the corresponding host software to control the accelerators. This code is composed of multiple steps (data allocation, kernel configuration, data transfers, synchronization primitives). Each step is wrapped into a specific function so that it can be further specialized in the hardware generation flow with specific optimizations. We then use the `v++` tool of the commercial Xilinx Vitis Unified Platform to create the hardware description with HLS and generate the final FPGA bitstream. The Vitis platform also builds and links the host application with the Xilinx Runtime (`g++`). Our flow can easily be extended to target similar HBM-based platforms (like the Intel Stratix 10 MX) with the respective toolchains.

3.3 Compiler Architecture

Compiling for the template architecture in Figure 4 is fundamentally different to compiling for mainstream multicore CPUs. Instead of extending the original CFDlang compiler [44] with new target support, we devise an entirely new implementation based on the MLIR framework. With MLIR, we raise the language abstraction level while remaining target-agnostic in our front-end. At the same time, our middle-end is also mostly agnostic towards the concrete DSL we chose.

This also allows us to profit from the well-engineered and widespread MLIR and LLVM ecosystems, enabling re-use of abstractions and of lowering flows to different architectures. In the following, we describe the new CFDlang compiler infrastructure, focusing on the stack of dialects we created to support CFD simulations (cf. Figure 6). Our infrastructure includes the key additions of only three new dialects: `cfdlang`, i.e., our language front-end, `teil`, i.e., our domain optimization middle-end, and `base2`, i.e., our example for a hardware-specific back-end task. Apart from the dialects `linalg` and `affine` mentioned in Section 2.4, the figure also includes other standard dialects, like `tensor` for cross-domain tensor operations, `scf` to model structured control flow with, e.g., explicit loops, and external tools such as the ISL, which we use to produce the code for the downstream HLS compilation. The transformations performed within these dialects are discussed in Section 3.4.

Since we aim at leveraging a commercial HLS tool that does not currently support MLIR as input, our DSL compiler is an MLIR-based transpiler that emits C99 source code and additional metadata outputs to directly interface with Vitis HLS and Mnemosyne, respectively.

3.3.1 The Front-end. Following our motivating example given by Figure 2, we enter the compiler flow at the highest level of abstraction. Here, the `cfdlang` dialect replaces both the abstract syntax tree (AST) and expression tree representations that were used previously in CFDlang [44]. During translation, the DSL parser directly emits `cfdlang` dialect operations while ingesting the DSL

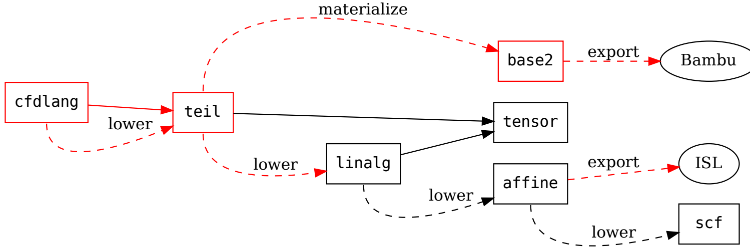


Fig. 6. MLIR dialects and tools (ellipses) in the compiler and their dependencies. Our contributions to the MLIR language stack are marked in red.

```
// t = S#S#S#u . [[1 6][3 7][5 8]]
cfdlang.define @t : [11 11 11] {
    %S = cfdlang.eval @S : [11 11]
    %u = cfdlang.eval @u : [11 11 11]
    %0 = cfdlang.prod %S, %u : [11 11], [11 11 11]
    %1 = cfdlang.prod %S, %0 : [11 11], [11 11 11 11 11]
    %2 = cfdlang.prod %S, %1 : [11 11], [11 11 11 11 11 11 11]
    %3 = cfdlang.cont %2 : [11 11 11 11 11 11 11 11 11] indices [2 7][4 8][6 9]
    cfdlang.yield %3 : [11 11 11]
}
```

(a) Translation of Figure 2 (excerpt)

```
1     %0 = teil.prod %S, %u : tensor<11x11x!teil.num>, tensor<11x11x11x!teil.num>
2     %1 = teil.diag 2 5 %0 : tensor<11x11x11x11x11x!teil.num>
3     %2 = teil.red add 2 %1 : tensor<11x11x11x11x!teil.num>
4     %3 = teil.prod %S, %2 : tensor<11x11x!teil.num>, tensor<11x11x11x!teil.num>
5     %4 = teil.diag 2 5 %3 : tensor<11x11x11x11x11x!teil.num>
6     %5 = teil.red add 2 %4 : tensor<11x11x11x11x!teil.num>
7     %6 = teil.prod %S, %5 : tensor<11x11x!teil.num>, tensor<11x11x11x!teil.num>
8     %7 = teil.diag 2 5 %6 : tensor<11x11x11x11x11x!teil.num>
9     %8 = teil.red add 2 %7 : tensor<11x11x11x11x!teil.num>
```

(b) Optimized `teil` lowering of Figure 7a

Fig. 7. DSL lowering in the compiler.

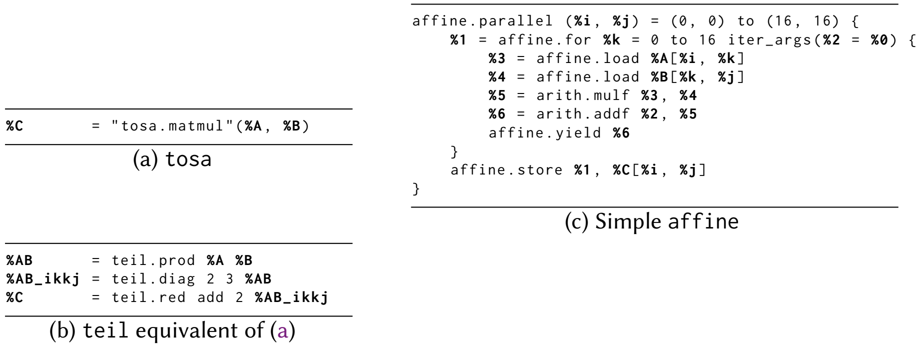
code, resulting in Figure 7a. This dialect mirrors the syntactical elements of the DSL in MLIR. To establish a working front-end, we implement a translation from the DSL to the `cfdlang` dialect, which also works backwards. In addition, we implement a lowering to our DSL agnostic middle-end dialect `teil`. We benefit from the MLIR diagnostic engine, and immediate semantic analyses and verification in the MLIR infrastructure.

As this dialect replaces the AST, it does not perform aggressive canonicalization and instead attempts to preserve the input program as closely as possible, with the exception of type declarations. CFDlang's implicit scalar type greatly simplifies its type system and unclutters this particular MLIR representation. The set of operations is also kept extremely simple, mapping language elements $1:1$ onto operations in the CFDlang dialect, with elements such as `cfdlang.eval` taking the place of identifier expressions and the like. Transformations and optimizations are left to the middle-end.

3.3.2 The Middle-end. The program description obtained in Figure 7a is tied to the CFDlang DSL, which is not desirable for implementing reusable optimization pipelines. We address this issue by introducing another level of abstraction based on the concept of tensors as immutable values, implemented by the `teil` dialect. The exact semantics of this cross-domain tensor abstraction are directly imported from its specification [45]. We first lower onto this dialect as illustrated by Figure 7b, removing DSL-specific elements from the IR, and then continue to transition downwards as outlined in Figure 6. The set of abstractions and transformations starting with and below the `teil` dialect make up our compiler's middle-end.

Unlike MLIR's `linalg` dialect for linear algebra and tensor expressions, `teil` is more restrictive on the operations it models. Similarly, `teil` does not allow for partially defined values or incomplete domains, but also prohibits treating tensors as arrays. `teil` is a true value-based dialect with tensors as first-class citizens without any links to array materialization. It is therefore more related to the `vtensor` concept of torch-mlir[4] and the `linalg_ex` extensions found in IREE[5]. This allows us to perform deductive reasoning on complex tensor operators, assigning them to array buffers later.

`teil` is another dialect bridging the gap between other high-level tensor dialects, such as the related HLO and TOSA dialects, and lower dialects, such as `linalg` and `affine`. While HLO[6], TOSA[7] and TPP [18] are closer to Instruction Set Architectures (ISAs) for tensor accelerators in ML applications, `teil` aspires to be a cross-domain tensor expression dialect. For example, `tosa.matmul` (Figure 8a) encodes the well-known generic matrix-matrix multiplication (GEMM), which is broken down into primitive operations in `teil` as shown in Figure 8b. Trying to map arbitrary `teil` programs onto `tosa` or HLO is generally not possible, however, for reasons of unsupported operators and data types.



```
%C       = "tosa.matmul"(%A, %B)
```

(a) tosa

```
%AB      = teil.prod %A %B
%AB_ikkj = teil.diag 2 3 %AB
%C       = teil.red add 2 %AB_ikkj
```

(b) teil equivalent of (a)

```
affine.parallel (%i, %j) = (0, 0) to (16, 16) {
    %1 = affine.for %k = 0 to 16 iter_args(%2 = %0) {
        %3 = affine.load %A[%i, %k]
        %4 = affine.load %B[%k, %j]
        %5 = arith.mulf %3, %4
        %6 = arith.addf %2, %5
        affine.yield %6
    }
    affine.store %1, %C[%i, %j]
}
```

(c) Simple affine

Fig. 8. GEMM in MLIR (types omitted)

3.3.3 The Back-end. Lowering `teil` to the `affine` dialect (cf. Figure 8c), we integrate with our polyhedral optimizations for CFD applications described in [16]. Starting with `affine`, the MLIR stack also allows us to import optimizations for mainstream CPUs such as in [44]. In the next section, we will show how this maps to the template architecture from Figure 4.

We designed the `base2` dialect to include arbitrary-precision floating-point operations. Both `cfdlang` and `teil` use an abstract scalar type, implemented by `base2`'s parametric types, and operations that model arbitrary-precision data. We can then use the `ieee754` type to encode custom floating-point types that can be consumed by subsequent HLS tools, like Bambu [15].

3.3.4 Limitations. We support programs that can be directly mapped onto the primitives of TeIL [45], which is currently the most abstract middle-end representation of our compiler. On one hand, these primitives are common in many scientific applications similar to CFD simulations. On the other hand, the MLIR ecosystem "at large" is modular and flexible, offering additional dialects that can be potentially lowered towards `affine` and `base2`, extending our flow to more domains.

A downside to using the CFDlang DSL for this demonstration is that it does not allow for dependencies between the implicit elements. Solvers commonly examined on FPGA platforms include

[4] https://github.com/llvm/torch-mlir
[5] https://github.com/google/iree
[6] https://github.com/tensorflow/mlir-hlo
[7] https://developer.mlplatform.org/w/tosa/

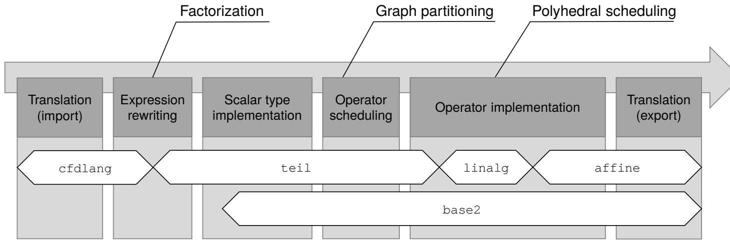


Fig. 9. Steps in the CFDlang compiler, including MLIR dialects and principal transformations.

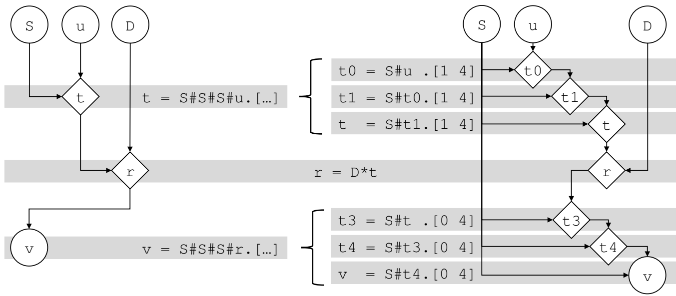


Fig. 10. Expression rewriting example. Circles indicate buffers and diamonds virtual tensors. Arrows are added to indicate dataflow dependences, and nodes ordered according to a naive schedule.

Lattice-Boltzmann methods (LBM) [46], which need to exchange halo regions with neighbouring compute units. However, for single devices, we expect this problem to be negotiated through memory, via overlapped compute and alternating kernels, thus still utilizing most of our flow.

3.4 Compilation Process

In [16] we described analysis and transformations at the kernel-level that allowed for memory allocation and operation scheduling for FPGA offloading. These were based on polyhedral techniques, which still apply to the `affine` intermediaries that our MLIR flow produces. For example, the liveness analysis required for Mnemosyne's sharing optimization follows the same approach described in [16]. To address the HBM challenges (cf. Section 2.3), we introduce new transforms at higher levels, namely operator graph partitioning and pipelining. These increase the achieved memory bandwidth, required to take advantage of the high throughput of the HBM-based target architecture. We elaborate on Section 3.3 by discussing the individual steps as shown in Figure 9.

3.4.1 Expression Rewriting. The `teil` dialect provides a high level view on the tensor expressions, allowing them to be rewritten effectively. During the expression rewriting phase, our compiler uses strictly beneficial mathematical identities to reduce the runtime complexity of the program. Within `teil`, which uses abstract scalars modeling $\mathbb{R}$, these are always semantically preserving.

One such rewrite, introduced to CFDlang in [44], uses associativity and distributivity to factorize contractions, as shown in Figure 10. This exact rewrite can also be observed in Figure 7b, where the input expression is given by Figure 7a (with a `teil` lowering beforehand). In this transformation, the computational complexity of a contraction on an outer product is reduced by pulling it (partially) down to the factors. The design of `teil` makes these scenarios easy to recognize, and fully automates this and other kinds of rewriting, such as aggressively transforming towards GEMM patterns.

3.4.2 Scalar Type Implementation. Digital signal processors (DSPs), which are used to implement IEEE-754 floating-point operations, are critical resources in FPGAs. In addition to being relatively scarce and routing intensive, when compared to lookup tables (LUTs) and flip-flops (FFs), floating-point operations require multiple clock cycles. Such long pipelines are especially large design hazards once the design frequency drops, as the adverse effects are multiplicative.

To reduce the impact of floating-point operations, our compiler infrastructure supports custom number representations. While the existing CPU compilation flow uses `double` as default scalar type, the FPGA flow allows arbitrary-precision types. The exploration of this design space, however, is not automated by this work, and left up to the user. We intend on coupling the compiler with exploration frameworks such as [49, 8] in the future. Implementations make use of the fact that `teil` can work with multiple equivalence classes of its own abstract scalar type. This allows deferring the choice of a concrete type while still retaining the ability to reason about precision boundaries.

As a starting point, we developed the `base2` dialect that provides a minimal interface to encode this information, which must be processed by a back-end consumer. To test our implementations on the CPU using software-emulated arbitrary-precision floating-point arithmetic, we emit calls to libsoftfloat[8]. Additionally, calls to this library can be conveniently synthesized to hardware arbitrary-precision floating-point logic by HLS tools that support this feature, like Bambu [15].

3.4.3 Operator Scheduling. To saturate the bandwidth of the HBM memory interface, the compiler must generate an implementation that maximizes the throughput from and into the PCs. This is mostly done by trading resources for frequency and cycle count using chained pipelines.

Compare Figure 10 for the level of abstraction that this transformation is preformed on, which is the tensor value graph. Following all rewrites to the whole input program from Figure 2, this operator scheduling arrives at the graph shown in Figure 11. The additions here are the operator groups, represented by grey boxes, and streams, represented by arrows. Within each box, standard FPGA-directed pipeline scheduling will be implemented.

To achieve maximum throughput, a grouping must be found that allows streaming the inputs and outputs at saturating rates. Unfortunately, although these constraints can be propagated from the PCs, rates are not reliable measurements for this process. The actual throughput of the hardware design depends on the achieved design frequency, which remains a variable until the end, leaving timings uncertain. As a result, our current implementation relies either on user input, or assumes that no variables outside of its control will lower the design frequency.

Our current strategy first aggressively partitions the graph into the smallest possible operators (i.e. one per tensor value), and then collapses them. Assuming less stages are better since they use fewer resources, operators can be merged automatically under a given private local memory (PLM) and DSP budget. This heuristic prefers collapsing chains, thus reducing the first in - first out (FIFO) queues required to implement the top-level dataflow. For our running example, this strategy yields the three top-level groups shown and named in Figure 11.

Aside from the groups shown here, the template architecture fixes two groups implicitly, namely the read from and write to HBM (cf. Figure 4). We found that our designs synthesized in a way where the group cycle intervals can be reasonably estimated by the sum of trip counts of their child loops. When collapsing, the group with the longest interval determines the lower bound on the achievable latency, and thus our heuristic uses that interval as a budget to collapse towards.

Starting from Figure 11, we can obtain an implicit top-level pipeline over the groups. Our simple implementation selects one topological ordering, as-late-as-possible (ALAP) [4], which is later used for the system generation (cf. Figure 5). Respecting the concurrent nature of hardware implementations, future extensions may consider merging stages that can execute in parallel.

[8]https://github.com/ucb-bar/berkeley-softfloat-3

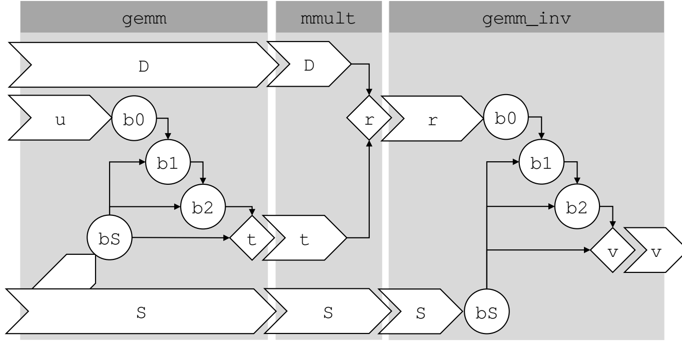


Fig. 11. Operator scheduling example. The program from Figure 10 is partitioned into 3 groups (gemm, mmult, and gemm_inv). Interfaces between stages are turned into streams, as indicated by the broad arrows.

3.4.4 Operator Implementation. To ensure the group pipeline scheduled in the previous step runs as designed, the inner nodes of each group must be implemented as a regular pipelined loop nest.

In [16], we extended the CFDlang compiler with polyhedral analysis and code generation capabilities. This framework allowed us to implement transformations to render the code amenable to futher processing with HLS tools. In particular, we incorporated changes to the memory layout and loop structure, alleviating resource burdens and improving pipelining. Here, the polyhedral model was an especially good fit for liveness analysis, which allowed us to defer an implementation problem, the memory sharing, to the C-to-system generation part.

We lower from `teil` to `affine` and plumb directly into this previous code generator from [16] (cf. Figure 12). This also allows us to reuse code generation via the integer set library (ISL) [61], circumventing the need for a new HLS back-end dialect. The streaming property of tensors between groups can be trivially upheld using polyhedral scheduling, by constraining the order of the writes. More than one output stream per operator requires additional effort that was not needed for this work but has been examined previously [62]. A stopgap solution lies in buffering the reads in the groups (as shown in Figure 11), which may sometimes even be a necessary resource trade-off if consecutivity cannot be established. In [16] we described memory sharing optimizations that reduce this pressure on the FPGA's BRAM again. For the kernel evaluated in this paper, memory sharing turned out to be not as relevant, especially when increasing the number of groups. In fact, a downside of the stage level sharing is that it may limit the ability to pipeline the internals of the stage, ultimately sacrificing throughput and increasing the cycle interval.

3.4.5 Automation. There are two sides to automation of these transforms in the MLIR flow, because of the separation of policy and mechanism. The `teil` dialect allows us to fully automate tensor expression rewriting, within the confines of mathematical reasoning, including decision making. The `base2` dialect in conjunction with `teil` provides a mechanism for arbitrary-precision scalars, but offers no support for exploring that design space. In `teil`, we can encode partitions of operator pipelines with ease, but as we also see in vendor toolchains, simple heuristics can not replace proper design space exploration based on performance estimation in the general case. The ISL is a powerful tool that can make and enforce decisions (e.g. schedule loops based on streaming constraints), but has an inherently biased and non-deterministic scheduler.

For custom data types, we currently rely on command-line parameters to guide the grouping in the presence of design frequency hazards and apply array layout and partitioning maps [16]. While we aim at keeping these hints out of the DSL, they can be represented using meta-operations in the IR. An example of this is HeteroCL [29], and this has already been proposed for TeIL [55].

```
affine.for %arg5 = 0 to 11 {
    affine.for %arg6 = 0 to 11 {
        affine.for %arg7 = 0 to 11 {
            affine.for %arg8 = 0 to 11 {
                %6 = affine.load %bS[%arg5, %arg8] : memref<11x11xf64>
                %7 = affine.load %b0[%arg6, %arg7, %arg8] : memref<11x11x11xf64>
                %8 = arith.mulf %6, %7 : f64
                %9 = affine.load %b1[%arg5, %arg6, %arg7] : memref<11x11x11xf64>
                %10 = arith.addf %8, %9 : f64
                affine.store %10, %b1[%arg5, %arg6, %arg7] : memref<11x11x11xf64>
            }
        }
    }
}
```

(a) in affine

```
for (int c1 = 0; c1 <= 10; c1 += 1)
    for (int c2 = 0; c2 <= 10; c2 += 1)
      for (int c3 = 0; c3 <= 10; c3 += 1) {
        // stmt0
        b1[121 * c1 + 11 * c2 + c3] = 0;
        for (int c4 = 0; c4 <= 10; c4 += 1) {
            #pragma HLS pipeline
            // stmt0
            b1[121 * c1 + 11 * c2 + c3] = b1[121 * c1 + 11 * c2 + c3] + bS[11 * c1 + c4] *
                 ↪  b0[121 * c2 + 11 * c3 + c4];
        }
      }
```

(b) in C99

Fig. 12. Figure 7b lines 1-3 during operator implementation

3.5 Hardware Generation Architecture

Our hardware generation flow aims at optimizing the data transfers around the kernel implementation produced by CFDlang. It receives as input the description of the kernel and the compatibility graph of the internal buffers (from CFDlang), along with information about board resources (from the user). It produces an optimized CU description (in C++) and the platform configuration file based on the number of CUs that can be instantiated. The configuration file specifies also the proper connections to the HBM channels. To implement our hardware generation flow, we developed Olympus, which creates an optimized system-level architecture (in synthesizable C++) for our accelerators. During its execution, it interfaces with Mnemosyne [41] to optimize the on-chip memory associated with each kernel and limit the number of local memory resources. Mnemosyne uses the buffer compatibility graph generated by the CFDlang compiler to determine when the physical on-chip memory banks can be reused without performance overhead [41]. After this, Olympus reads the kernel interface and determines how to connect the input/output ports to the rest of the system to efficiently exchange data with the HBM channels. Data ports are connected to HBM channels via AXI Master interfaces, while configuration ports are connected to the host via AXI-Lite, memory-mapped interfaces. Data exchanged with HBM channels (i.e., input and output matrices) are buffered on-chip to allow fast, fixed-latency access during kernel execution.

The generation of the system architecture is guided by the designer, as shown in Figure 5. Starting from the C kernel produced by the CFDlang compiler, Olympus generates a minimal wrapper to run HLS and obtain an estimate of the resources needed for the target FPGA device. Then, we apply optimizations to both the on-chip local memory of each kernel (with Mnemosyne) and the system-level memory architecture to exchange data with the HBM (with Olympus).

At the kernel level, we use Mnemosyne to generate the RTL of the on-chip memory architecture associated with each kernel. Mnemosyne is a tool that exploits sharing compatibilities, i.e., when distinct internal buffers have no overlapping lifetime, to assign them to the same physical banks based on a given cost metric. In these cases, the tool generates custom logic to manage the accesses

```
void kernel(double S[121], double D[1331], double u[1331], double v[1331], double t0[1331],
            double t1[1331], double t[1331], double r[1331], double v0[1331], double v1[1331]) {
    //...
}
```

(a) After CFDlang

```
void top_kernel(double S[121], double D[1331], double u[1331], double v[1331]) {
    double t0[1331], t1[1331], t[1331], r[1331], v0[1331], v1[1331];
    kernel(S, D, u, v, t0, t1, t, r, v0, v1);
}
```

(b) After Mnemosyne

```
void top_kernel(double S[121], double D[1331], double u[1331], double v[1331]) {
    double t0[1331], t1[1331];
    kernel(S, D, u, v, t0, t1, t0, t1, t0, t1);
}
```

(c) After Mnemosyne with array sharing

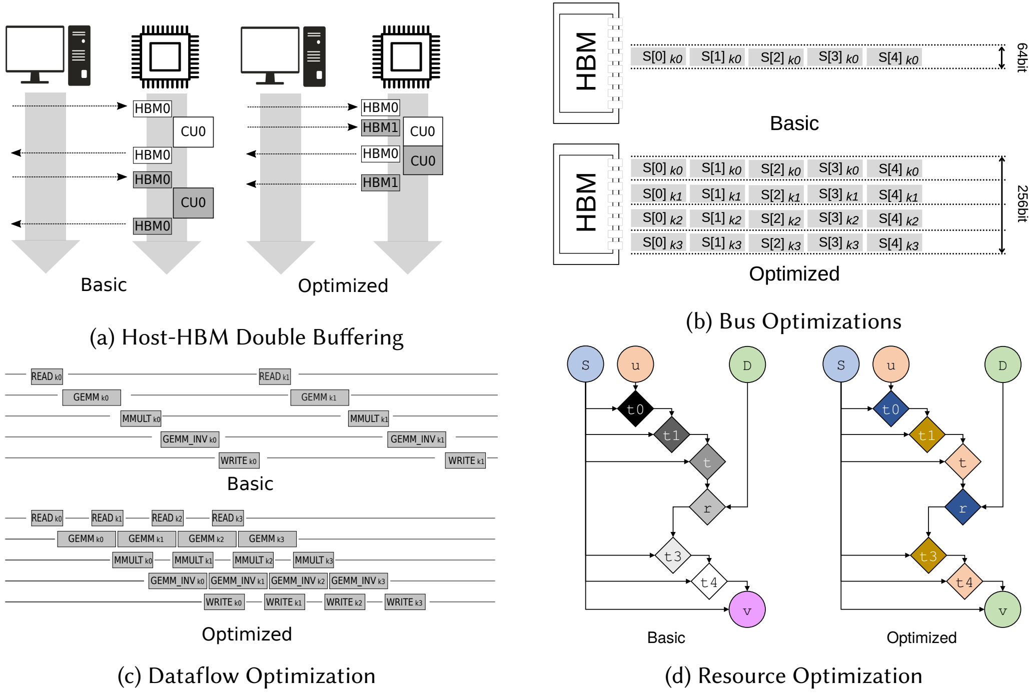
Fig. 13. Kernel interface before and after Mnemosyne. Internal buffers are exposed by CFDlang. Mnemosyne reproduces a kernel description that has only "real" input and output buffers for interfacing with the HBM channels. Other temporary on-chip buffers are internally optimized and possibly shared.

to the same memory banks from different kernel interfaces [41, 16]. Figure 13 shows the conceptual interface, described in C, of a kernel before and after the execution of Mnemosyne. Mnemosyne requires the specification of the compatibility graph of the local arrays and, in our flow, these metadata are computed and produced by CFDlang during liveness analysis [16]. Mnemosyne wraps the RTL kernel description (produced by HLS) with the resulting RTL description of the kernel memory architecture to expose only input and output ports to the CU. This conceptual interface is then used for integration of the kernel into the CU.

At the system level, Olympus generates the C++ description of the memory architecture around the kernel description, which is integrated as custom RTL to use the results produced by Mnemosyne. Since the hardware cost of the kernel may limit the number of parallel CUs, the designer can use Olympus to understand which optimizations can be applied given the FPGA available resources. Indeed, we characterize each optimization with an estimation of the extra resources. With this information, the designer can select the most suitable optimizations and Olympus generates the corresponding CU description around the CFDlang-generated code of the kernel and the system configuration file. In the future, this process can be further automated by combining it with state-of-the-art design space exploration frameworks. Each CU can feature multiple kernels, each of them connected to a lane to fully utilize the AXI bandwidth (cf. Section 3.1). The CU wrapper implements data-movement optimizations and is designed accordingly with changes to the host application and the configuration file. For example, the kernel may benefit from a change in the way data is written to and read from global memory and therefore the host application must be updated accordingly. The configuration file, instead, defines how each CU interfaces with the HBM. By modifying this file, Olympus defines how each CU is connected to the individual channels.

The resulting components are then passed to Vitis, i.e., the HLS platform that we use, to automatically generate the bitstream required for board configuration. When timing is not met, Vitis automatically downscales the execution frequency. While this is useful to enable proper functionality, the designer has little control over this process. Conversely, Olympus introduces some optimizations for the synthesis process (cf. Section 3.6.4).

3.5.1 Limitations. Our hardware generation flow does not currently include a step for including platform-specific optimizations, like the mapping of array streams to specific memory resources.



(a) Host-HBM Double Buffering (b) Bus Optimizations (c) Dataflow Optimization (d) Resource Optimization

Fig. 14. Optimizations targeting the memory and communication challenges of the HBM-based systems.

This is the step that also needs customization when bringing the flow to a new HLS tool or platform as it requires insertion of the proper directives.

Also, our flow is required to emit C99 source code to interface with the given HLS tool. A direct interface would be beneficial to avoid losing semantic details that are contained in the MLIR dialects.

3.6 Hardware Generation Process

Let p be the polynomial degree for the simulation, each Inverse Helmholtz kernel needs $p \times p$ values for matrix S and $p \times p \times p$ values for matrices D and u. The kernel then produces $p \times p \times p$ values for the output matrix v. These values are useful to estimate the number of resources for input, temporary, and output buffers. The size of the total amount of data used to compute one element can be used to determine how many elements worth of data can fit into an HBM channel (max size is 256MB). The number of elements is the **batch size**, i.e., the number of executions the CU can perform without interruption before needing the host to send more data. Data transfers are required between consecutive batches.

In the following, we describe the optimizations that we apply to our CFD application by means of the Olympus options (step *Optimize* in Figure 5). Such optimizations (and the Olympus hardware generation process) can be easily adapted to similar tensor-based applications.

3.6.1 Host-HBM Double Buffering. In a naive implementation, the host code transfers the input data required to execute one batch of kernel elements into HBM. The host then invokes the CUs to execute on each of these elements and generate the corresponding output results. The host transfers these outputs back from HBM to its main memory. Each CU interfaces with one PC and we can instantiate up to 32 CUs (each with one kernel) to operate in parallel. However, all communication

and execution for a single CUs is serialized. Since the host-HBM communication is as expensive as the computational part, this significantly affects the overall performance.

To overlap the host-HBM data transfers with the CU execution, we use double buffering as shown in Figure 14a. Each CU interfaces with two PCs, namely "even" and "odd". The host reads the output from the last iteration and writes new input into the "even" channels while the PCs operate on the data in the "odd" channels, and vice versa. When the total host transfer time for input and output of one batch is less than the total CU execution time for the same batch, the host transfer time is entirely hidden and the CUs are actively executing at all times.

Olympus Implementation: Double buffering requires changes in the wrapper to determine which PC the CU should operate on for the current batch, in the configuration file to attach more channels to the same CU, and in the host code to manage the data transfers to separate HBM channels. Additionally, since we use two PCs to implement double buffering, we limit the maximum number of parallel CUs to 16. However, when the total number of CUs that can be instantiated is less than 8, we also separate input and output channels to simplify the control logic and improve logic connectivity of the FPGA resources.

3.6.2 HBM-FPGA Bandwidth Optimization. The data elements of the Inverse Helmholtz operator are 2- or 3-D tensors ($p \times p$ or $p \times p \times p$, respectively), where each element is a 64-bit floating-point number (*double*). The HBM interfaces have 256 bits, so transferring one double at a time uses only 25% of the bandwidth. Packing four doubles (or more depending on the custom data type) onto the bus allows us to significantly reduce the number of clock cycles for data transfers. However, we need to efficiently manage the multiple parallel data to avoid serialization when writing them into the buffers. To fully exploit the parallelism, we conceptually divide the 256-bit bus into four 64-bit "lanes" and replicate the kernel four times within a CU. Each of these kernels is directly connected to one of the "lanes", i.e., to 64 bits of the bus, so that all kernels can operate in parallel. In this way, *Read*/*Write* modules still require the same amount of cycles (cf. Figure 14b) but we can now start the computation of four elements in parallel.

Olympus Implementation: To obtain the layout shown in Figure 14b, Olympus modifies the host code to interleave the input for the multiple elements before sending it to HBM and de-interleave the output after receiving the results. The optimization only needs information on the bus bitwidth (e.g., 256 bits) and the data type bitwidth (i.e., 32 or 64 bits). Both parameters are available from the user-supplied board specification and the compiler-supplied array information, respectively. From this, Olympus generates the CU *Read* and *Write* functions to split and aggregate the data into the appropriate number of lanes. The overall CU structure is then created by composing the *Read*/*Write* functions with multiple instances of the kernels. Similarly, the data reorganization portion of the host code can be generated with the same information by specializing the allocation functions of the host application.

3.6.3 Dataflow Optimization. Each single execution of the CU in the batch must read data from the HBM, execute the operator on them, and write back the result into the HBM. The Inverse Helmholtz operator is implemented in DSL as three operations: a tensor contraction, a Hadamard product, and another tensor contraction. We can further decompose the single operations into elementary blocks, as shown in Figure 11. The fundamental blocks can be implemented as subfunctions in the kernel that communicate via AXI Stream in a dataflow model. These hardware modules will thus execute in a pipelined manner, significantly improving the throughput. The number of elementary blocks in each subfunction is a tradeoff between latency and resource requirements. Indeed, having more blocks in the same subfunction increase resource sharing opportunities but also increases the latency of the pipeline stages, reducing the throughput. This optimization improves the performance but also increases the resource usage, potentially limiting the number of CUs that can be instantiated.

Olympus Implementation: The dataflow optimization is enabled by the compiler generating a kernel composed of subfunctions using streams, instead of one flat kernel function. The exact scheduling of the stages may not be as straightforward as the three conceptual operators, as the compiler has freedom to optimize the grouping for the best performance. Olympus then creates data streams among the subkernels for data communication. In order to stream data between the subkernels, data must be buffered when the subkernel does not operate on it in the same order that it is streamed or when the same values are reused multiple times inside the same subfunction. In most cases, this means that data streamed in gets stored in an internal buffer, then the data can be operated on using random access, and as each result is computed, it is streamed out. Data structures that are reused across multiple blocks (like matrix S) must be streamed through these blocks and buffered inside them to keep a consistent structure and avoid multiple hardware modules accessing the same data concurrently. This optimization does not require any changes in the host code and the size of the streams can be configured by the designer when selecting the optimization. When no information is provided, the tool assumes to use the full size of the array as the size of the corresponding stream.

3.6.4 Resource Optimizations and Multiple Compute Units. The Inverse Helmholtz operator is composed of seven loops that are executed in sequence. Each loop produces a matrix. Intermediate matrices are used by the next loops. Each of these matrices requires on-chip resources (generally BRAMs) to store the values. The number of available BRAMs can limit the number of FPGA kernels. However, once the matrix is not used anymore, the corresponding BRAM resources can be used by the same kernel to store new data.

Using the liveness information generated by the CFDlang compiler, we can reduce the number of on-chip resources required by each kernel. Arrays with disjoint lifetimes can use the same physical memory buffer as shown in Figure 14d. Reducing the kernel's BRAM requirements can increase the total number of kernels that we can instantiate. However, sharing opportunities can operate only inside each subkernel. So, the effects of this optimization may be limited. It is worth noting that we currently optimize the use of memory resources only by exploiting sharing opportunities while platform-specific optimizations (like the implementation of arrays with specific memory resources) can be integrated as an additional step.

Also, given the physical nature of the input data, we can adapt the measurement scale so that the values are always in a range between -1 and 1. This observation allows us to change the way we interpret the data, passing from a floating-point representation to a fixed point one. In particular, we use a 64-bit representation where 24 bits are assigned to the integer part (including the sign) and 40 to the decimal part. Specifically, we used the `ap_fixed` library to specify these formats so that they can be automatically synthesized by Vitis HLS. So, these optimizations are compatible with any HLS tool (like Bambu [15]) that can synthesize these formats. This step brings with it considerable advantages. Fixed-point operations require simpler hardware than floating-point operations. Tensor operators make heavy use of multipliers. So, fixed-point operations allow designers to obtain faster hardware that uses fewer resources and consumes less energy.

In CFD, another method to decrease resource utilization is to reduce the degree p. This will be at some cost to convergence of the overall simulation, but this choice is up to the designer who provides the required value of p for the simulation in the input DSL. Once this value is decided, our flow will be able to automatically instantiate more parallel compute units due to the smaller data size and reduced number of the operations/loop iterations.

Olympus Implementation: For buffer sharing, as discussed above, we use Mnemosyne to automatically generate optimized PLM units that can share physical banks in a way that is completely transparent to the kernel execution [41, 16]. Olympus only combines the resulting kernel description

with the rest of the system in a transparent way. Data representation is left as a design choice for the application developer, as the tolerable error depends heavily on the application. Using this choice as input, the data type can be automatically changed in the implementation to be able to observe the effects on area, power, and performance. Fixed-point implementations only require a redefinition of the data types before HLS using the given arbitrary-precision libraries. We decided to implement the conversion from/to double in the host code to save hardware resources. However, this requires an adaptation of the data allocation functions, which receive the input values in double but need to write fixed-point values in the FPGA buffers, and the functions to retrieve the results that must implement the opposite conversion. Finally, the polynomial degree p is an intrinsic parameter of the input DSL code (cf. Figure 2). We can re-run the complete flow to generate an accelerator with a different polynomial degree to enable the proper compiler and hardware generation optimizations.

4 EVALUATION

In this section, we use our DSL-to-bitstream flow to evaluate and compare several implementations of the CFD application. To do so, we present our experimental setup (Section 4.1), we analyze the impact of each optimization for the Inverse Helmholtz kernel (Section 4.2), and we compare our final results with software implementations (Section 4.3).

4.1 Experimental Setup

To evaluate our DSL-to-bitstream flow, we extended the flow presented in [16] as described in Figure 5: CFDlang is implemented on top of the MLIR infrastructure, Mnemosyne [41] is an open-source tool[9], and Olympus is a new in-house prototype. Olympus is built in Python on top of the Pyverilog library [57] for hardware generation (i.e., the generation of the kernel wrappers around Mnemosyne artifacts) and the Pycparser library[10] for code generation. With our novel flow, we targeted a Xilinx Alveo U280 card on an AMD EPYC 7282 [3] server running Centos 7. We used Xilinx Vitis 2021.1 [64] for synthesis and bitstream creation. Unless otherwise specified, we target a synthesis frequency of 450 MHz for both the platform and the CU description. For each implementation, we evaluate performance and energy efficiency by using the GFLOPS and GFLOPS/W metrics, respectively. The GFLOPS metric is obtained by dividing the total number of floating-point operations executed by the application by the total execution time, while the GFLOPS/W metric is obtained by dividing the GFLOPS metric by the average power consumption of the system. To get accurate information about FPGA power consumption, we profile the power consumption during the system execution with Xilinx XRT and we use the average value.

4.2 Impact of Hardware Optimizations

To evaluate the cumulative benefits introduced by each hardware optimization, we progressively apply them to the *Inverse Helmholtz* operator used as case study and so extensively discussed in this paper. We evaluated the implementations for two polynomial degrees: $p = 7$ and $p = 11$. In particular, given the polynomial degree p, we assume that each contraction is composed of three loop nests that execute two floating-point operations (one addition and one multiplication) for $p \times p \times p \times p$ times each. Similarly, the Hadamard product requires $p \times p \times p$ multiplications. So, the entire Inverse Helmholtz operator the following number of floating-point operations:

$$N_{op}^{el} = 2 \cdot [2 \cdot (p \cdot p \cdot p \cdot p) + 2 \cdot (p \cdot p \cdot p \cdot p) + 2 \cdot (p \cdot p \cdot p \cdot p)] + (p \cdot p \cdot p) = (12 \cdot p + 1) \cdot (p \cdot p \cdot p) \quad (2)$$

[9] https://github.com/chrpilat/mnemosyne
[10] https://github.com/eliben/pycparser

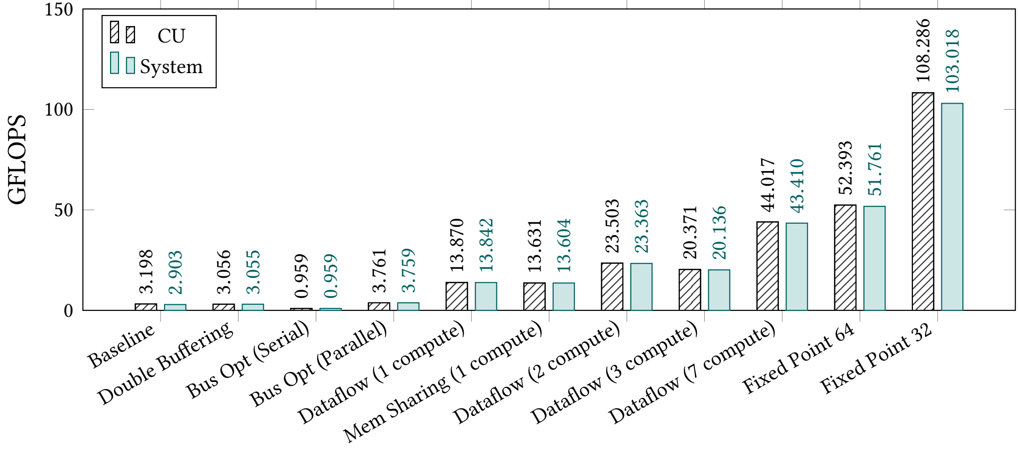


Fig. 15. Performance of each optimization implemented with 1 CU and $p = 11$.

So, a single element is required to execute N_{op}^{el}=177,023 floating-point operations when $p = 11$ and N_{op}^{el}=29,155 floating-point operations when $p = 7$. The total number of floating-point operations for a CFD simulation can be obtained as:

$$N_{op} = N_{eq} \times N_{op}^{el} \tag{3}$$

We executed all experiments with $N_{eq} = 2,000,000$, i.e., we simulated 2,000,000 elements.

We executed our CFDlang on the DSL description in Figure 2 to generate the C kernel for hardware optimization. We first performed experiments to evaluate the effects of optimizations with $p = 11$. In particular, we progressively added the following optimizations:

- Baseline: No optimizations are used. The code serially executes kernels and data transfers, while each CU contains only one kernel and is connected to the HBM with 64-bit AXI channels.
- Host-HBM Double Buffering: We introduce this well-known optimization to hide CPU-FPGA communication latency.
- HBM-FPGA Bus Optimization: We evaluate the effect of widening the bus to 256 bits, with only one kernel unit (and serializing the data) and with multiple lanes feeding parallel kernel units.
- Dataflow optimization: We create several variants of the compute functions with one, two, three, and seven subkernels. We evaluate the performance vs. resources trade-off.
- Resource Optimization: We apply on-chip memory sharing (only in the case of dataflow implementations with one block inside the compute part) and fixed-point optimizations (with 64- and 32-bit implementations).

For each of these implementations, we measured total and kernel execution times, maximum and average power consumption, and cost in terms of hardware resources. Figure 15 shows the performance (in terms of GFLOPS) achieved in each experiment when adding the specific optimization on top of the previous ones. In each experiment, the left black & white bar (CU) shows the GFLOPS of the CUs on their own, without considering host-FPGA data transfers, while the right azure bar (System) includes the entire application. Comparing the two bars allows us to evaluate the peak performance of the kernels and the effects of data transfers.

The Baseline case achieves only 2.9 GFLOPS, and the difference between the CU performance and the overall system performance is 9.2%. This is due to the serial nature of the implementation where data is transferred from the host to the HBM, then processed by the CU and sent back to the host before starting a new batch. If more data needs to be transferred, this discrepancy between

CU performance and overall system performance will grow larger, as the CU needs to wait for all of the data to be sent before beginning execution.

After the Double Buffering optimization, the CU performance remains similar, with a small degradation due to overhead, while the system performance is now exactly the same as the CU performance. This is an improvement over the Baseline implementation, because now the host to HBM data transfers are happening in parallel to and are entirely hidden behind the CU execution.

We then executed two experiments for evaluating Bus Optimization. In the *Serial* version, we attempt to utilize the full bandwidth of the 256-bit bus by packing four doubles. The CU reads them in parallel but then it serializes them when it needs to access its own local buffers. While this optimization is supposed to speed up data reads from the HBM, its implementation in the CU leads to a performance *degradation* of about 3×. This is mostly due to the complexity of aligning the data ($p \times p$ and $p \times p \times p$) to multiples of fours inside the bus. To combat this, but still use the full bus bandwidth, this optimization was replaced with the *Parallel* implementation where four kernels are instantiated in the CU and the data for each "lane" is stored in separate buffers, one for each kernel. This led to a 3.92× speedup over the *Serial* implementation, as the four lanes are now all read in parallel. There is only a 1.23× speedup over the Double Buffering implementation, where a 4× speedup would be expected. This is because in both Bus Opt implementations, the HLS tool only instantiated two double-precision multipliers (rather than 11 in all other implementations). So when the innermost loops are unrolled, a resource limitation violation increases the initiation interval (II) to 4, effectively counteracting the expected 4× speedup. We use the *Parallel* architecture in the following experiments.

Next, we tested various forms of the Dataflow Optimization. Each implementation of this optimization separated the kernels into read, compute, and write modules and streams were used to pass data between them, allowing a pipelined structure. When using one compute subkernel (*Dataflow (1 Compute)*) test, the speedup was 3.68×. In these cases, the HLS tool instantiated the full 11 multipliers, removing the II violation. This decision of the HLS tool along with the overlapping execution of the read, compute, and write modules allows us to effectively achieve a 4× speed-up, i.e., to full exploit the four parallel lanes. Since the compute module was dominating the execution time, it was further split into 2, 3, and 7 modules. The Inverse Helmholtz operator comprises seven loop nests, each implementing the operations in the seven grey rows on the right side of Figure 10. To split into 2 modules, the kernel is divided into a first module with the first three loop nests with `S` and `u` as input and `t` as output, and a second module with the last four loop nests with `S`, `D`, and `t` as input and `v` as the final output. The split was made as such so that the first module does not need `D` as an input and the second module does not need `u` as an input. To split into 3 modules, we use the division shown on the left side of Figure 10 and in Figure 11. This is the most natural division as it matches the initial DSL representation and the first three loop nests implement the `gemm` operator, the fourth loop nest implements the `mmult` operator, and the last three loop nests implement the `gemm_inv` operator. Another benefit to this division is that the `mmult` loop nest consumes and produces data in the same order it is sent via the streams, meaning that no extra buffering is needed for this module and each data element can be immediately processed as it is received leading to a minimal latency. To split into 7 modules, each loop nest is a separate module.

All three of these tests gained speedup over the *1-Compute* version by breaking the total execution time of a single module down further. The *2-Compute* version is 1.7× faster than the *1-Compute* version. The discrepancy between this result and the ideal speedup of 2× is due to the extra data buffering that must be done in each module in order to allow random access. In the *1-Compute* case, the input arrays are each buffered one time, which adds an overall latency equal to the total input data size. In the *2-Compute* case, the `S` array is needed by both modules and must be buffered twice. Additionally, the output of the first module, `t`, is used as input to the second module and must

also be buffered. This extra buffering is overlapped while the two modules execute in a pipeline fashion, but it means that the latencies of each module are not exactly half of the total latency of one unified module. However, *3-Compute* modules was slower than *2-Compute* modules. The overall execution time is determined by the module with the longest latency, as it is the limiting factor in the overall latency of the system. Because the loop nest implementing the `gemm` operator has a minimal latency, moving it to a separate module does not significantly reduce the latency of the largest module. In fact, in each case, the module with the longest latency was the same, but the extra modules and control routing caused the tools to frequency scale the *3-Compute* case to execute at 266 MHz whereas the *2-Compute* case executed at 292 MHz. When this is taken into account, the performance of both tests is approximately the same. The *7-Compute* test, however, performed the best because each of the compute modules were much smaller than the previous tests. In this case, the latencies of these modules were now slightly shorter than the latency of the read module, meaning that this is the limit of the performance increase by dividing the compute portion. The *7-Compute* test gained a total speedup of 4.03× over the Bus Opt *Parallel* implementation.

To evaluate the efficiency of the allocated resources, we computed the “ideal” GFLOPS value for each of the double-precision floating-point implementations. We identified the total number of double-precision adders and multipliers instantiated in the CU (# Ops) by analyzing the synthesis reports generated by Vitis HLS. The ideal GFLOPS is computed by multiplying this value by the frequency of the CU and represents the performance if all operators were always in use concurrently. Table 2 compares these values with the measured GFLOPS of each implementation. In the last column, an “efficiency” is calculated as a ratio between the ideal and achieved GFLOPS.

Table 2. Efficiency of floating-point operators

	# Ops	f (MHz)	Ideal GFLOPS (# Ops $\times f$)	Achieved GFLOPS	Efficiency
Baseline	22	274.6	6.041	2.903	0.481
Double Buffering	22	259.8	5.716	3.055	0.535
Bus Opt (Serial)	4	286.5	1.146	0.959	0.837
Bus Opt (Parallel)	16	296.6	4.746	3.759	0.792
Dataflow (1 compute)	88	286.2	25.186	13.842	0.550
Dataflow (2 compute)	176	291.9	51.374	23.363	0.455
Dataflow (3 compute)	180	266.3	47.934	20.136	0.420
Dataflow (7 compute)	532	199.5	106.134	43.410	0.409

This “efficiency” reflects the behavior of the allocation and scheduling of the HLS tool more than the efficiency of our system design surrounding each HLS kernel. However, we can still gain some insight into our design in the cases where the HLS decisions were the same. For instance, in the *Baseline* and *Double Buffering* cases, the same kernels are used and therefore they each have the same # Ops. The efficiency increases because less time is “wasted” waiting on data transfers from the host. Both *Bus Opt* implementations reduce the # Ops because the HLS tool used a different local memory type with less read ports, restricting the unrolling and therefore only used two adders and two multipliers for each kernel. The efficiency values of these implementations are also much higher because these are the only cases where the multipliers themselves are pipelined. The ideal GFLOPS metric expects each operator to produce a result every clock cycle, so in all other cases where the operators are not pipelined, there are several cycles of latency for each operation reducing the efficiency. Between each of the *Dataflow* implementations, the efficiency drops slightly as the computation is split into more modules because it is impossible to split the computation into equal latency modules. The module of longest latency may be computing at all times, but the shorter length modules must stall.

Table 3. Resource utilization for each optimization implemented with 1 CU and $p = 11$. Highlighted in red is any value over 25% utilization, indicating possible issues when instantiating more than one CU.

	f_{max} (MHz)	LUT		FF		BRAM		URAM		DSP	
Baseline	274.6	141137	(10.8%)	214402	(8.2%)	244	(12.1%)	57	(5.9%)	150	(1.7%)
Double Buffering	259.8	148873	(11.4%)	228561	(8.8%)	246	(12.2%)	57	(5.9%)	150	(1.7%)
Bus Opt (Serial)	286.5	146088	(11.2%)	225542	(8.7%)	268	(13.3%)	3	(0.3%)	55	(0.6%)
Bus Opt (Parallel)	296.6	182632	(14.0%)	295340	(11.3%)	330	(16.4%)	12	(1.3%)	192	(2.1%)
Dataflow (1 compute)	286.2	215199	(16.5%)	335009	(12.8%)	330	(16.4%)	240	(25.0%)	592	(6.6%)
Dataflow (2 compute)	291.9	291964	(22.4%)	446258	(17.1%)	330	(16.4%)	240	(25.0%)	1068	(11.8%)
Dataflow (3 compute)	266.3	293757	(22.5%)	448385	(17.2%)	298	(14.8%)	164	(17.1%)	1096	(12.1%)
Dataflow (7 compute)	199.5	473743	(36.4%)	735030	(28.2%)	330	(16.4%)	252	(26.3%)	3016	(33.4%)
Mem Sharing (1 compute)	282.4	229115	(17.6%)	336133	(12.9%)	282	(14.0%)	124	(12.9%)	592	(6.6%)
Fixed Point 64	233.8	254242	(19.5%)	342390	(13.1%)	330	(16.4%)	252	(26.3%)	4368	(48.4%)
Fixed Point 32	244.5	231062	(17.7%)	346507	(13.3%)	1338	(66.4%)	0	(0.0%)	2294	(25.4%)

The efficiency values for all implementations (except *Bus Opt*) are all near 0.5 because each multiply-accumulate is implemented as eleven parallel multipliers and eleven sequential adders. Even though the additions are sequential, the tool still allocated eleven of them. Because the *Bus Opt* implementations are restricted to two adders, their efficiencies are higher.

At this point, we want to start replicating the CUs using the remaining area available in the FPGA fabric to maximize parallelism. For this reason, we need to evaluate the hardware cost of each implementation. The numbers of LUT, FF, BRAM, URAM, and DSP used by each case for $p = 11$ are shown in Table 3. In general, each test from Baseline to Dataflow (7 Compute) showed an increase in resource utilization. Any utilization value over 25% is shown in red. These are the resources most likely to cause placement and routing issues when instantiating multiple CUs. We tested a few methods to reduce resource utilization to be able to increase the number of instantiated CUs.

The *Mem Sharing* optimization is applied to the Dataflow *1-Compute* implementation where several arrays are used in the compute module (cf. Figure 14d). Mnemosyne generated an architecture to internally share arrays based on their liveness intervals. This decreased the BRAM utilization by 14.5% and the URAM utilization by 48.3% while the LUT and FF utilization only increased minimally and the DSP utilization remained the same. Also, the execution time was only slightly reduced (a slowdown of 0.98×). Conversely, this optimization cannot be applied to the Dataflow *2-Compute*, *3-Compute*, and *7-Compute* implementations because, in these cases, each compute module only uses arrays that cannot be shared, as they are always in use during the module execution. This optimization is indeed beneficial when on-chip memory inside the CU is the limiting factor, and when replicating the CUs brings more improvements than dataflow execution.

Another method to reduce resources is to change the numerical representation. All of the previous tests used the floating-point format with double precision. In general, fixed-point representations utilize fewer resources than floating-point ones. We tested 64- and 32-bit fixed-point representations by modifying the Dataflow *7-Compute* implementation. The 64-bit implementation uses 24 bits for the integer portion and 40 bits for the fractional portion. The 32-bit implementation uses 8 bits for the integer portion and 24 bits for the fractional portion. These values are provided by the user after an analysis of the algorithm. Because the 32-bit data is half the size, we instantiate 8 kernels per CU and divide the 256-bit bus into 8 lanes. In the *Fixed Point 64* test, the LUT utilization reduced by 46.3%, the FF utilization reduced by 53.4%, the RAM utilization remained the same, and the DSP utilization increased 44.8%. In the *Fixed Point 32* test, with respect to the *Fixed Point 64* test, the LUT and FF utilization remained roughly the same. The DSP utilization was nearly halved. The BRAM increased by about four times while the URAM decreased to zero. This is because the data representation is half as long, so the overall size of the data structures are half as big. The arrays

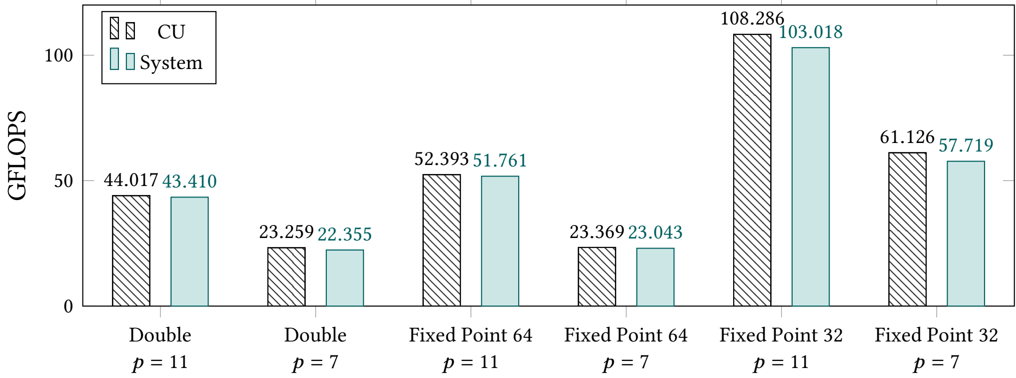


Fig. 16. Performance of each data representation implemented with 1 CU and $p = 11$ or $p = 7$

Table 4. Resource utilization for each data representation implemented with 1 CU and $p = 11$ or $p = 7$. Shown in red is any value over 25% utilization, indicating possible issues when instantiating multiple CUs.

	p	f_{max} (MHz)	LUT		FF		BRAM		URAM		DSP	
Double	11	199.5	473743	(36.4%)	735030	(28.2%)	330	(16.4%)	252	(26.3%)	3016	(33.4%)
Double	7	225.9	328267	(25.2%)	527809	(20.2%)	438	(21.7%)	0	(0.0%)	1888	(20.9%)
Fixed Point 64	11	233.8	254242	(19.5%)	342390	(13.1%)	330	(16.4%)	252	(26.3%)	4368	(48.4%)
Fixed Point 64	7	201.4	191348	(14.7%)	299992	(11.5%)	438	(21.7%)	0	(0.0%)	2760	(30.6%)
Fixed Point 32	11	244.5	231062	(17.7%)	346507	(13.3%)	1338	(66.4%)	0	(0.0%)	2294	(25.4%)
Fixed Point 32	7	297.0	177280	(13.6%)	306386	(11.8%)	438	(21.7%)	0	(0.0%)	1382	(15.3%)

representing the tensors are no longer big enough for the synthesis tool to decide it is efficient to use URAM to store them. When taking into account the size of the physical memories, the total memory space is approximately halved. The performance of the *Fixed Point 64* test had a slight speedup of 1.19× due to the simplification of the logic allowing the frequency to be higher. The Dataflow *7-Compute* test with double format was scaled to 199 MHz while the *Fixed Point 64* test was scaled to 234 MHz. The performance of the *Fixed Point 32* test had a speedup over the double format of 2.37× and it reaches up to 103 GFLOPS. This represents a speed up of more than 35× over the Baseline version. The *Fixed Point 64* test exhibited a mean square error of 9.39×10^{-22} while the *Fixed Point 32* test had a mean square error of 3.58×10^{-12}. It is up to the application designer to determine what an acceptable error is and decide on an appropriate number format, and our flow can help facilitate a design space exploration of these parameters.

Another method to reduce resource utilization for this kernel is to vary the input parameter p. We tested the Dataflow *7-Compute* implementation using 64-bit double, 64-bit fixed point, and 32-bit fixed point with $p = 7$ and $p = 11$. The results are summarized in Figure 16 and Table 4.

Compared to their $p = 11$ counterparts, the $p = 7$ implementations performed slightly slower. This is because the actual hardware implementation does not scale exactly the same way as the conceptual floating point operations per kernel (used to compute the GFLOPS). However, the resource reduction between $p = 11$ and $p = 7$ is enough to allow for more replication of the CUs. For instance, the *Fixed Point 32* implementation uses 66.4% of the available BRAM for $p = 11$ while it only uses 21.7% for $p = 7$, allowing 4× replication.

In order to further facilitate instantiating multiple CUs, we reduced the stream FIFOs from a naive full size to small enough to save space and still prevent deadlock. This led to a small performance reduction, due to stalls, but significantly reduced the total number of BRAMs. Also, because the

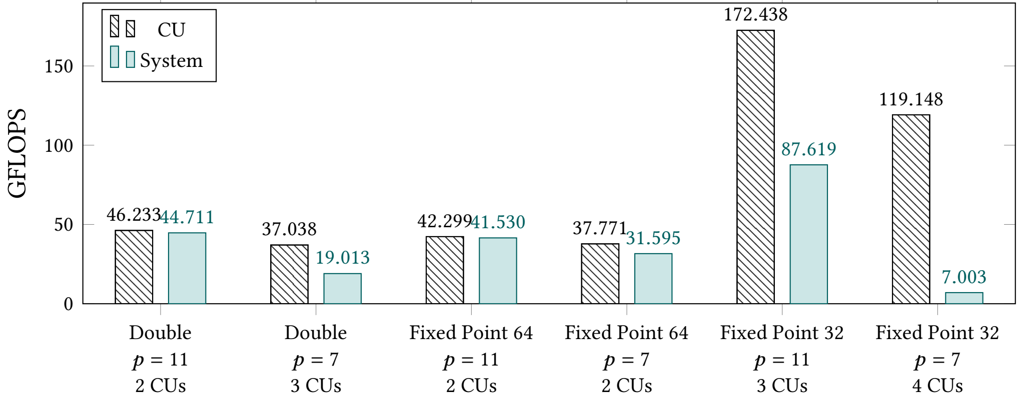


Fig. 17. Performance of each data representation implemented with multiple CUs and $p = 11$ or $p = 7$

Table 5. Resource utilization for each data representation implemented with multiple CUs and $p = 11$ or $p = 7$. Shown in red is any value over 50% utilization to show which resources are the limiting factor.

	p	CUs	f_{max} (MHz)	LUT		FF		BRAM		URAM		DSP	
Double	11	2	146.0	760903	(58.4%)	795663	(30.5%)	442	(21.9%)	456	(47.5%)	6020	(66.7%)
Double	7	3	179.2	777208	(59.7%)	955705	(36.7%)	878	(43.6%)	0	(0.0%)	5651	(62.6%)
Fixed Point 64	11	2	132.3	755752	(58.0%)	485525	(18.6%)	442	(21.9%)	440	(45.8%)	7316	(81.1%)
Fixed Point 64	7	2	168.2	268285	(20.6%)	368056	(14.1%)	658	(32.6%)	0	(0.0%)	5508	(61.0%)
Fixed Point 32	11	3	194.0	479387	(36.8%)	571733	(21.9%)	1274	(63.2%)	960	(100.0%)	6868	(76.1%)
Fixed Point 32	7	4	178.3	404747	(31.1%)	498999	(19.1%)	1100	(54.6%)	0	(0%)	5508	(61.0%)

DSP utilization was particularly high in some cases, we used pragmas to guide the HLS tool on how to use LUTs instead of DSPs to implement fixed-point multipliers. We used this pragma in one of the seven compute modules to shift some of the resource load off of DSPs and onto LUTs.

We were able to instantiate two parallel CUs for the cases of *Double* with $p = 11$, *Fixed Point 64* with $p = 11$, and *Fixed Point 64* with $p = 7$, three CUs for the cases of *Double* with $p = 7$ and *Fixed Point 32* with $p = 11$, and four CUs for the case of *Fixed Point 32* with $p = 7$. The performance results for these implementations are shown in Figure 17 and the area results are shown in Table 5. All of these implementations were built targeting 225 MHz, as most of their 1 CU counterparts could not even achieve this.

In most cases, replicating the CUs actually led to slowdown. This is because the extra logic and routing caused the maximum frequency to be reduced thereby slowing down everything in the system. However, most cases did show speedup in terms of the CU execution time. In particular, the *Fixed Point 32* implementations were able to achieve up to 172 GFLOPS for the kernel but around 87 GFLOPS for the system. This huge discrepancy is due to the fact that even though several CUs are now executing in parallel, all of the data must still be sent from the host to the HBM in series. The host data transfers are now the dominating factor by far, so it is not recommended to replicate CUs until the host data transfer time can be reduced. Otherwise, the overall system will have a slowdown from the extra logic.

From the resource utilization results it can be seen that both 64-bit data types are constrained by resources used for computation, namely LUTs and DSPs. The 32-bit fixed point implementation is also somewhat constrained by DSPs. In any case, this application is composed of almost entirely floating or fixed point multiplications, and performance-optimized designs will quickly use most

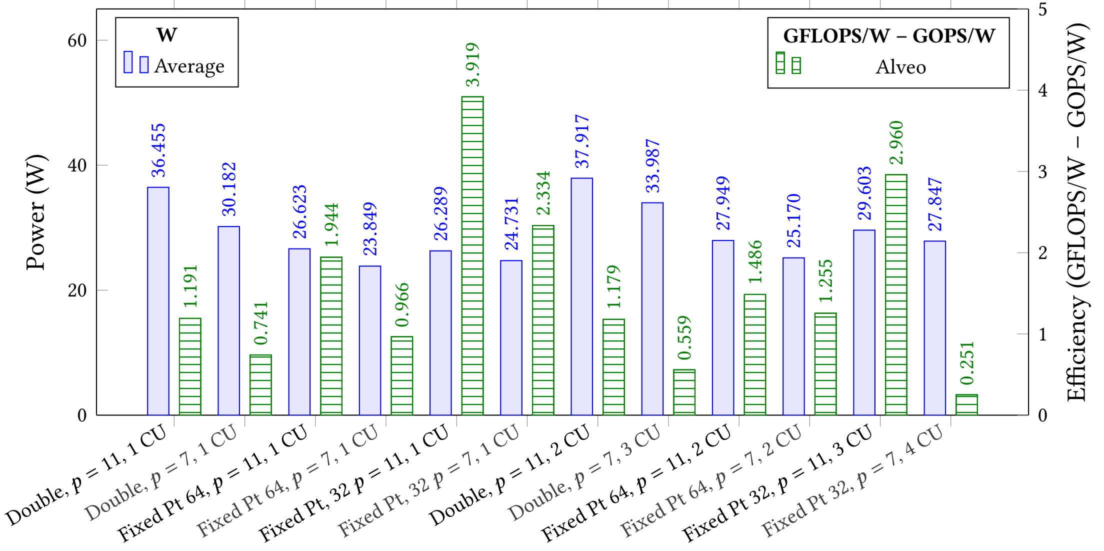


Fig. 18. Power usage of the Dataflow (7 Compute) optimization with each datatype, $p = 11$ or $p = 7$, and 1-CU or multiple-CU.

of the available DSPs. The 32-bit cases are also constrained by the on chip memories, the 3 CU implementation of *Fixed Point 32* with $p = 11$ even uses 100% of the URAM, but both *Fixed Point 32* implementations were able to be replicated more than their *Fixed Point 64* counterparts, due to the data width reduction. The $p = 7$ tests were also, in general, able to be replicated more than their $p = 11$ counterparts due to the effect p has on the amount of computations and the array sizes. *Fixed Point 64* was only able to be replicated twice in both cases of p, as the reduction of DSPs between $p = 11$ and $p = 7$ was not enough to allow for a third CU.

Figure 18 shows the power consumption of the different implementations and a comparison of the energy efficiency (GFLOPS/W for floating-point operations and GOPS/W for fixed-point operations). The bars report the average power consumption measured with the XRT infrastructure. We also include the results of the multiple-CU implementations, to show the effects of replication on both power consumption (W bars) and energy efficiency (GFLOPS/W and GOPS/W bars).

As expected, the fixed-point implementations are more efficient than the floating-point ones. Also, reducing the bitwidth from 64 to 32 bits allows us to achieve the maximum efficiency. This is because these implementations are much faster and use less hardware resources. The $p = 7$ implementations have lower average power consumption than their $p = 11$ counterparts, due to their smaller resource utilization. However, in most cases the efficiency of the $p = 7$ cases is lower due to their longer overall execution time. The multiple-CU implementations are generally less efficient than their single-CU counterparts, both because of the increased work occurring in parallel, yielding a higher average power, and because of longer execution times from frequency scaling.

4.3 Comparison with Software Implementations

This sections presents the results for two additional kernels. The first kernel performs an interpolation, which maps from $\mathbf{u} \in \mathbb{R}^{N \times N \times N}$ to $\mathbf{u'} \in \mathbb{R}^{M \times M \times M}$ via an isotropic operator $\mathbf{A} \in \mathbb{R}^{M \times N}$. We implemented the *Interpolation* kernel with $M = N = 11$. The second kernel computes ∇u, the gradient of u in all 3 dimensions. We implemented the *Gradient* kernel with dimensions $8 \times 7 \times 6$. In

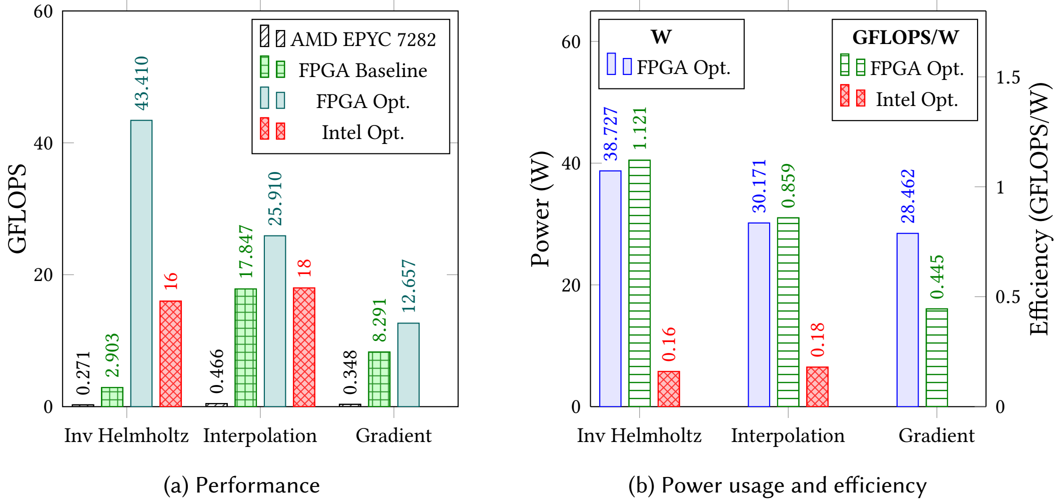


Fig. 19. Performance and power results of various kernels. For the Inverse Helmholtz and Interpolation kernels, we also include the performance and estimated efficiency of the corresponding highly-optimized Intel implementations [44]. All experiments are executed using double-precision floating point.

all cases, we compute the total number of floating-point operations needed for simulating 2,000,000 elements and we use these numbers to compute the GFLOPS and GFLOPS/W metrics.

Figure 19a shows the performance results for the *Inverse Helmholtz*, *Interpolation*, and *Gradient* kernels on the AMD EPYC 7282 and the FPGA. These results include the AMD execution (black bars), baseline (no optimizations) FPGA implementation (green bars) and the fully optimized FPGA implementation (azure bars). The fully optimized kernels all use double-precision floating point data and implement the Double Buffering, Bus Opt (Parallel), and Dataflow (where each loop nest is a subkernel module) optimizations. The baseline implementations achieved 10.7× - 38.3× speedup over their software execution on the AMD EPYC 7282. The optimized FPGA implementations, however, were able to achieve 36.4× - 160.2× speedup over the AMD execution.

To compare our results with state-of-the-art software implementations, Figure 19a also includes the performance of highly-optimized Intel implementations for the *Inverse Helmholtz* and *Interpolation* kernels [44] (red bars). These implementations are generated by the original CFDlang compiler using the process described in [44], which was found to outperform expert-crafted manually optimized kernels. The executables are compiled with the Intel Compiler and uses the Math Kernel Library 2017.2.174 and were profiled on a 24-core Intel Xeon E5-2680 v3 CPU (Haswell), running at 2.50 GHz. The FPGA-optimized *Inverse Helmholtz* and *Interpolation* kernels achieved 2.7× and 1.4× speedup over the optimized Intel execution, respectively.

The average power and efficiency for the optimized kernels are shown in Figure 19b. The estimated efficiencies (GFLOPS/W) of the Intel executions for the *Inverse Helmholtz* and *Interpolation* kernels are also shown. These estimations are calculated using the GFLOPS results of the kernel and a conservative estimate of the average power (100 W), assuming the CPU would be operating under a lower load than the thermal design power (120 W). The *Interpolation* and *Inverse Helmholtz* kernels are 4.8× and 7.0× more efficient than the Intel CPU execution, respectively. Recalling the results from Figure 18, the most power-efficient implementation of the *Inverse Helmholtz* (32-bit fixed-point with p = 11 and 1 CU) is 24.5× more efficient than the Intel execution.

5 CONCLUDING REMARKS

Numerical simulations are compute-intensive HPC applications used to solve complex problems in many scientific fields. Such applications benefit greatly from parallelization. In this context, HBM FPGA devices are increasingly used to achieve high performance with high energy efficiency. However, designing HBM architectures for such systems is complex and requires specific skills. Our analysis of such architectures reveals that the high cost of communication between CPU and FPGA memories and the limited amount of resources to implement parallel kernels are the major issues. To address these challenges, we redesigned a DSL compiler in MLIR to automatically generate HLS-ready code, along with an HLS-based flow to automate the generation of optimized system architectures that implements several memory-related optimizations. Our MLIR framework offers much quicker turnaround times in implementing the language. The differences in flexibility between our custom IR and the new dialects were negligible, while diagnostics, stability, and composition are greatly improved. We have created an opportunity to apply our strategies, even if partially, to other MLIR-based flows to achieve a more direct comparison with our results in future.

Our results show that the data format can have a significant impact on performance; a smaller data format simplifies the logic and allows the circuit to have a shorter overall latency and a to operate at a higher frequency. The polynomial degree, an application-specific parameter, can also have an impact on the performance for similar reasons. These parameters also reduce the total FPGA resources needed to perform the computations, allowing for the possibility of instantiating multiple compute units in the FPGA fabric. However, replication does not equate to increased performance unless the host data transfers can be significantly reduced. If the host data transfers are bounding the application, the design can be optimized for power efficiency by only instantiating one compute unit. However, if the host were interfaced with multiple FPGAs and were able to send data in parallel to all of them, replicating the compute units on to separate FPGAs would achieve increased performance. Overall, we were able to achieve up to 103 GFLOPS–more than 6× faster than highly-optimized Intel implementations–with an energy efficiency of about 4 GFLOPS/W–almost 24× more efficient than highly-optimized Intel implementations.

ACKNOWLEDGMENTS

This project is partially funded by the EU Horizon 2020 Programme under grant agreement No 957269 (EVEREST).

REFERENCES

[1] M. Abadi et al. *TensorFlow: A system for large-scale machine learning*. 2016. arXiv: 1605.08695 [cs.DC].

[2] A. Agrawal et al. "TensorFlow Eager: A multi-stage, Python-embedded DSL for machine learning". In: *Proceedings of Machine Learning and Systems* 1 (2019), pp. 178–189. DOI: 10.48550/ARXIV.1903.01855.

[3] *2nd gen AMD EPYC™ 7282*. AMD. 2021. URL: https://www.amd.com/en/products/cpu/amd-epyc-7282.

[4] Z. Baruch. "Scheduling algorithms for high-level synthesis". In: *ACAM Scientific Journal* 5.1-2 (1996), pp. 48–57.

[5] J. Bergstra et al. "Theano: a CPU and GPU Math Expression Compiler". In: *Python in Science Conference (SciPy)*. 2010, pp. 3–10.

[6] A. Brauckmann et al. "PolyGym: Polyhedral Optimizations as an Environment for Reinforcement Learning". In: *Proceedings of the 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Sept. 2021, pp. 17–29. DOI: 10.1109/PACT52795.2021.00009.

[7] E. Calore et al. "Performance assessment of FPGAs as HPC accelerators using the FPGA Empirical Roofline". In: *International Conference on Field-Programmable Logic and Applications (FPL)*. 2021. DOI: 10.1109/FPL53798.2021.00022.

[8] Y. Chatelain et al. "Automatic exploration of reduced floating-point representations in iterative methods". In: *European conference on parallel processing*. Springer. 2019, pp. 481–494.

[9] T. Chen et al. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning". In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USA: USENIX Association, 2018.

[10] M. Chiu et al. "Molecular Dynamics Simulations on High-Performance Reconfigurable Computing Systems". In: *ACM Transactions on Reconfigurable Technology Systems* 3.4 (Nov. 2010). DOI: 10.1145/1862648.1862653.

[11] Y.-k. Choi et al. "HBM Connect: High-Performance HLS Interconnect for FPGA HBM". In: *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. 2021, pp. 116–126. DOI: 10.1145/3431920.3439301.

[12] Y.-k. Choi et al. *When HLS Meets FPGA HBM: Benchmarking and Bandwidth Optimization*. 2020. arXiv: 2010.06075 [cs.AR].

[13] J. Cong et al. "Bandwidth Optimization Through On-Chip Memory Restructuring for HLS". In: *ACM/IEEE Design Automation Conference (DAC)*. 2017. DOI: 10.1145/3061639.3062208.

[14] J. Cong et al. "Source-to-Source Optimization for HLS." In: *FPGAs for Software Programmers*. Ed. by D. Koch et al. Springer, 2016, pp. 137–163.

[15] F. Ferrandi et al. "Invited: Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications". In: *ACM/IEEE Design Automation Conference (DAC)*. 2021. DOI: 10.1109/DAC18074.2021.9586110.

[16] K. F. A. Friebel et al. "From Domain-Specific Languages to Memory-Optimized Accelerators for Fluid Dynamics". In: *IEEE International Conference on Cluster Computing (CLUSTER)*. 2021. DOI: 10.1109/Cluster48925.2021.00112.

[17] N. Fujita et al. "HBM2 Memory System for HPC Applications on an FPGA". In: *IEEE International Conference on Cluster Computing (CLUSTER)*. 2021, pp. 783–786. DOI: 10.1109/Cluster48925.2021.00116.

[18] E. Georganas et al. *Tensor Processing Primitives: A Programming Abstraction for Efficiency and Portability in Deep Learning Workloads*. 2021. arXiv: 2104.05755.

[19] D. Giri et al. "ESP4ML: platform-based design of systems-on-chip for embedded machine learning". In: *IEEE/EDAA Design, Automation & Test in Europe Conference (DATE)*. 2020, pp. 1–6. DOI: 10.23919/DATE48585.2020.9116317.

[20] T. Gysi et al. "Domain-Specific Multi-Level IR Rewriting for GPU: The Open Earth Compiler for GPU-Accelerated Climate Simulation". In: *ACM Trans. Archit. Code Optim.* 18.4 (Sept. 2021). DOI: 10.1145/3469030.

[21] P. Holzinger et al. "Fast HBM Access with FPGAs: Analysis, Architectures, and Applications". In: *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2021. DOI: 10.1109/IPDPSW52791.2021.00030.

[22] H. Huang et al. "Shuhai: A Tool for Benchmarking HighBandwidth Memory on FPGAs". In: *IEEE Transactions on Computers* (2021), pp. 1–1. DOI: 10.1109/TC.2021.3075765.

[23] I. Huismann et al. "Factorizing the factorization – a spectral-element solver for elliptic equations with linear operation count". In: *Journal of Computational Physics* 346 (2017), pp. 437–448. DOI: http://dx.doi.org/10.1016/j.jcp.2017.06.012.

[24] I. Huismann et al. "Load balancing for cpu-gpu coupling in computational fluid dynamics". In: *International Conference on Parallel Processing and Applied Mathematics*. 2017, pp. 337–347.

[25] H. Jun et al. "HBM (High Bandwidth Memory) DRAM Technology and Architecture". In: *IEEE International Memory Workshop (IMW)*. 2017, pp. 1–4. DOI: 10.1109/IMW.2017.7939084.

[26] K. Kara et al. "High Bandwidth Memory on FPGAs: A Data Analytics Perspective". In: *International Conference on Field-Programmable Logic and Applications (FPL)*. 2020, pp. 1–8. DOI: 10.1109/FPL50879.2020.00013.

[27] D. Koeplinger et al. "Spatial: A Language and Compiler for Application Accelerators". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2018, pp. 296–311. DOI: 10.1145/3192366.3192379.

[28] R. Kuramochi et al. "An FPGA-Based Low-Latency Accelerator for Randomly Wired Neural Networks". In: *International Conference on Field-Programmable Logic and Applications (FPL)*. 2020. DOI: 10.1109/FPL50879.2020.00056.

[29] Y.-H. Lai et al. "HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing". In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '19. Seaside, CA, USA: Association for Computing Machinery, 2019, pp. 242–251. DOI: 10.1145/3289602.3293910.

[30] Y.-H. Lai et al. "Programming and Synthesis for Software-Defined FPGA Acceleration: Status and Future Prospects". In: *ACM Trans. Reconfigurable Technol. Syst.* 14.4 (Sept. 2021). DOI: 10.1145/3469660.

[31] C. Lattner et al. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *IEEE International Symposium on Code Generation and Optimization (CGO)*. 2004. DOI: 10.1109/CGO.2004.1281665.

[32] C. Lattner et al. "MLIR: A compiler infrastructure for the end of Moore's law". In: *arXiv preprint arXiv:2002.11054* (2020).

[33] C. Liu et al. "ScalaBFS: A Scalable BFS Accelerator on FPGA-HBM Platform". In: *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. 2021, p. 147. DOI: 10.1145/3431920.3439463.

[34] A. Lu et al. "CHIP-KNN: A Configurable and High-Performance K-Nearest Neighbors Accelerator on Cloud FPGAs". In: *International Conference on Field-Programmable Technology (ICFPT)*. 2020. DOI: 10.1109/ICFPT51103.2020.00027.

[35] Y. Maday et al. "Spectral element methods for the incompressible Navier-Stokes equations". In: *IN: State-of-the-art surveys on computational mechanics (A90-47176 21-64). New York* (1989), pp. 71–143.

[36] M. Meyer et al. "Evaluating FPGA Accelerator Performance with a Parameterized OpenCL Adaptation of Selected Benchmarks of the HPCChallenge Benchmark Suite". In: *IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. 2020, pp. 10–18. DOI: 10.1109/H2RC51942.2020.00007.

[37] W. S. Moses et al. *Polygeist: Affine C in MLIR.* 2021. URL: https://acohen.gitlabpages.inria.fr/impact/impact2021/papers/IMPACT_2021_paper_1.pdf.
[38] R. Nane et al. "A Survey and Evaluation of FPGA High-Level Synthesis Tools". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (2016), pp. 1591–1604. DOI: 10.1109/TCAD.2015.2513673.
[39] T. Nguyen et al. "The Performance and Energy Efficiency Potential of FPGAs in Scientific Computing". In: *IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS).* 2020, pp. 8–19. DOI: 10.1109/PMBS51919.2020.00007.
[40] C. Pilato et al. "EVEREST: A design environment for extreme-scale big data analytics on heterogeneous platforms". In: *IEEE/EDAA Design, Automation & Test in Europe Conference (DATE).* 2021. DOI: 10.23919/DATE51398.2021.9473940.
[41] C. Pilato et al. "System-Level Optimization of Accelerator Local Memory for Heterogeneous Systems-on-Chip". In: *IEEE Transactions on CAD of Integrated Circuits and Systems* 36.3 (2017). DOI: 10.1109/TCAD.2016.2611506.
[42] J. Ragan-Kelley et al. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* 2013, pp. 519–530. DOI: 10.1145/2491956.2462176.
[43] A. D. Rajagopala et al. "Impact of Off-Chip Memories on HLS-Generated Circuits". In: *International Workshop on FPGAs for Software Programmers (FSP).* 2019, pp. 1–10.
[44] N. A. Rink et al. "CFDlang: High-level code generation for high-order methods in fluid dynamics". In: *Proc. of RWDSL.* 2018, pp. 1–10. DOI: 10.1145/3183895.3183900.
[45] N. A. Rink et al. "TeIL: a type-safe imperative Tensor Intermediate Language". In: *ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY).* June 2019, pp. 57–68. DOI: 10.1145/3315454.3329959.
[46] K. Sano et al. "FPGA-based Streaming Computation for Lattice Boltzmann Method". In: *2007 International Conference on Field-Programmable Technology.* 2007, pp. 233–236. DOI: 10.1109/FPT.2007.4439254.
[47] P. Schlatter et al. "Large-scale Simulations of Turbulence: HPC and Numerical Experiments". In: *IEEE International Conference on eScience (e-Science).* 2011, pp. 319–324. DOI: 10.1109/eScience.2011.51.
[48] F. Sgherzi et al. "Solving Large Top-K Graph Eigenproblems with a Memory and Compute-optimized FPGA Design". In: *IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM).* 2021, pp. 78–87. DOI: 10.1109/FCCM51124.2021.00017.
[49] C. Shi et al. "An automated floating-point to fixed-point conversion methodology". In: *2003 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003. Proceedings. (ICASSP '03).* Vol. 2. 2003, pp. II–529. DOI: 10.1109/ICASSP.2003.1202420.
[50] G. Singh et al. "FPGA-Based Near-Memory Acceleration of Modern Data-Intensive Applications". In: *IEEE Micro* 41.4 (2021), pp. 39–48. DOI: 10.1109/MM.2021.3088396.
[51] G. Singh et al. "NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling". In: *International Conference on Field-Programmable Logic and Applications (FPL).* 2020. DOI: 10.1109/FPL50879.2020.00014.
[52] V. Šipková et al. "Manufacturing of weather forecasting simulations on high performance infrastructures". In: *IEEE International Conference on e-Science (e-Science).* 2016, pp. 432–439. DOI: 10.1109/eScience.2016.7870932.
[53] M. Siracusa et al. "A CAD-based methodology to optimize HLS code via the Roofline model". In: *ACM/IEEE International Conference On Computer Aided Design (ICCAD).* 2020, pp. 1–9.
[54] M. Siracusa et al. "Tensor optimization for high-level synthesis design flows". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.11 (2020), pp. 4217–4228. DOI: 10.1109/TCAD.2020.3012318.
[55] A. Susungi et al. "Meta-programming for cross-domain tensor optimizations". en. In: *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences.* Boston MA USA: ACM, Nov. 2018, pp. 79–92. DOI: 10.1145/3278122.3278131.
[56] K. Swirydowicz et al. *Acceleration of tensor-product operations for high-order finite element methods.* 2017. arXiv: 1711.00903 [cs.MS].
[57] S. Takamaeda-Yamazaki. "Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL". In: *Applied Reconfigurable Computing.* Ed. by K. Sano et al. Cham: Springer International Publishing, 2015, pp. 451–460.
[58] N. Vasilache et al. "Composable and Modular Code Generation in MLIR: A Structured and Retargetable Approach to Tensor Compiler Construction". In: *arXiv preprint arXiv:2202.03293* (2022).
[59] N. Vasilache et al. "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions". In: *arXiv preprint arXiv:1802.04730* (2018).
[60] S. K. Venkataramanaiah et al. "FPGA-Based Low-Batch Training Accelerator for Modern CNNs Featuring High Bandwidth Memory". In: *ACM/IEEE International Conference on Computer-Aided Design (ICCAD).* 2020, pp. 1–8. DOI: 10.1145/3400302.3415643.
[61] S. Verdoolaege. "isl: An Integer Set Library for the Polyhedral Model". In: *Mathematical Software – ICMS 2010.* Ed. by K. Fukuda et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 299–302.

[62] S. Verdoolaege et al. "Consecutivity in the isl Polyhedral Scheduler". In: (2017). Publisher: 10330. DOI: 10.13140/RG.2.2.15009.10082.

[63] J. Weerasinghe et al. "Enabling FPGAs in Hyperscale Data Centers". In: *IEEE International Conference on Ubiquitous Intelligence and Computing and IEEE International Conference on Autonomic and Trusted Computing and IEEE International Conference on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*. 2015, pp. 1078–1086. DOI: 10.1109/UIC-ATC-ScalCom-CBDCom-IoP.2015.199.

[64] *Vitis 2021.1 Acceleration Environment*. Xilinx. 2021. URL: https://www.xilinx.com/support/documentation-navigation/design-hubs/2021-1/dh0088-vitis-acceleration.html.

[65] R. Zhao et al. "Phism: Polyhedral High-Level Synthesis in MLIR". In: *arXiv:2103.15103 [cs]* (Mar. 2021). arXiv: 2103.15103. DOI: 10.48550/ARXIV.2103.15103.