From Domain-Specific Languages to Memory-Optimized Accelerators for Fluid Dynamics

Karl F. A. Friebel[1], Stephanie Soldavini[2], Gerald Hempel[1], Christian Pilato[2], Jeronimo Castrillon[1]
[1]Compiler Construction Chair, cfaed, Technische Universität Dresden, Dresden, Germany
{karl.friebel,gerald.hempel,jeronimo.castrillon}@tu-dresden.de
[2]Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milano, Italy
{stephanie.soldavini,christian.pilato}@polimi.it

***Abstract*—Many applications are increasingly requiring numerical simulations for solving complex problems. Most of these numerical algorithms are massively parallel and often implemented on parallel high-performance computers. However, classic CPU-based platforms suffers due to the demand of higher resolutions and the exponential growth of data. FPGAs offer a powerful and flexible alternative that can host accelerators to complement such platforms. Developing such application-specific accelerators is still challenging because it is hard to provide efficient code for hardware synthesis. In this paper, we study the challenges for porting a numerical simulation kernels onto FPGA. We propose an automated tool flow from a domain-specific language (DSL) to generate accelerators for computational fluid dynamics on FPGA. Our DSL-based flow simplifies the exploration of parameters and constraints such as on-chip memory usage. We also propose a decoupled optimization of memory and logic resources, which allows us to better use the limited FPGA resources. In our preliminary evaluation, this enabled doubling the amount of parallel kernels, increasing the accelerator speedup versus ARM execution from 7 to 12 times.**

***Index Terms*—FPGA, DSL, HLS, CFD**

I. Introduction

In the last years, data-intensive applications have permeated many computing areas due to the surge of deep learning and the ever-increasing demand for resolution in physics simulations (e.g., molecular dynamics, weather simulations). At the same time, the diminishing returns of technology scaling has led to vast system heterogeneity, with GPUs and tensor accelerators [5, 14, 26]. Hardware accelerators achieve high performance and energy efficiency thanks to specialization and spatial parallelism [7]. Reconfigurable hardware, like FPGA devices, is an attractive solution to democratize the use of such accelerators for different users [23].

Despite the progress in high-level synthesis (HLS) [21], we still face a large *semantic gap* between application experts and FPGA hardware architects for FPGA-based systems. This paper targets physicists and numerical experts for computational fluid dynamics (CFD). In this domain, the researchers must not only adapt the algorithms for a particular simulation but also face fragmented tools, integration tasks, complex libraries, and HLS directives for different targets. Integration tools can tackle such complexity by raising the abstraction level with language support, or by improving analysis and optimization to generate hardware from low-level code like C or Fortran.

Research on hardware generation from low-level code has a long history, with methods dating back to early auto-parallelising compilers [12]. Recent advances in code analysis and optimization reduce the manual effort required to produce HLS-friendly code, for instance, by inserting HLS pragmas [30] or by generating optimized systolic arrays [38]. Parallelism extraction is however sensitive to coding style. An alternative is to express a high-level specification with rich semantics in the form of a domain-specific language (DSL). High-performance DSLs have been successfully used to target CPUs and GPUs, e.g., for image processing [25], general tensor computations [15, 32], and deep learning [4, 34]. Similar flows have been proposed for FPGAs [16, 31]. Since most DSLs have high-level operators and data structures, compilers can decide shapes, layouts, and *schedules* to generate target-aware code.

In this paper we present a proof-of-concept of an end-to-end methodology that leverages the high-level semantics in DSLs to create FPGA-based systems and accelerate numerical kernels in CFD simulations (cf. Section III). As in other DSLs, we lower our specification into a polyhedral model, allowing us to leverage existing polyhedral transformations. Our key contributions are: (1) a study of code generation strategies to produce code that is amenable to commercial HLS (cf. Section IV), and (2) an approach to decouple the computational logic from management of on-chip data to improve the overall system efficiency and create composable architectures (cf. Section V). Decoupling computation from data management is particularly important for CFD and data-intensive applications to better coordinate data exchanges with the host CPU, hide/reduce the communication latency, and increase parallelism. Our decoupled approach is relevant also for other DSL-to-hardware compilation flows. We provide a preliminary evaluation for a fundamental CFD kernel, which helps identifying the potential and the challenges for upcoming FPGA nodes in HPC (cf. Section VI). For example, by exploiting memory sharing, we can fit more parallel kernel instances, increasing the speedup from $7.09\times$ to $12.58\times$ compared to ARM execution.

II. Background on Fluid Dynamics

A. Spectral Element Methods

In numerical mathematics, spectral element methods (SEM) are common in solving partial differential equations (PDEs),

```
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. 1. DSL code for the Inverse Helmholtz operator.

like the Navier-Stokes equations, which are impossible to solve analytically. SEM approximates the solution using functions, like the Fourier series, transforming the unknown physical quantities of the problem into spectral coefficients.

To reduce the numerical complexity, the simulated volume is divided into $N_e q$ smaller volumes. By partitioning the total space into several sub-spaces or *elements*, SEM reduces the numerical error introduced by the approximation. 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 for all spatial dimensions.

In this paper we focus on the Helmholtz equations, which are common in PDE solvers. Moreover, the Inverse Helmholtz operator is complex enough to subsume simpler operators (e.g., interpolation) which are similarly relevant in CFD simulations [13]. The operator can be formulated as:

$$t_{ijk,e} = \sum_{l=0}^{p} \sum_{m=0}^{p} \sum_{n=0}^{p} S^T_{li} \cdot S^T_{mj} \cdot S^T_{nk} \cdot u_{lmn,e} \tag{1a}$$

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

$$v_{ijk,e} = \sum_{l=0}^{p} \sum_{m=0}^{p} \sum_{n=0}^{p} S_{li} \cdot S_{mj} \cdot S_{nk} \cdot r_{lmn,e}\,. \tag{1c}$$

B. CFDlang DSL

In this paper we extend the CFDlang DSL for tensor operations [27]. CFDlang is target-agnostic and offers the user an interface that is close to the mathematical problem specification. The CFDlang notation is motivated by the tensor product notation often found in CFD applications. Equations 2a-2c are all equivalent to Equation 1a.

$$\mathbf{v} = \left(\mathbf{S}^T \otimes \mathbf{S}^T \otimes \mathbf{S}^T\right) \mathbf{u} \tag{2a}$$

$$= \left(\mathbf{S}^T \otimes \mathbf{S}^T \otimes \mathbf{S}^T \otimes \mathbf{u}\right)^{iljmkn}{}_{lmn} \tag{2b}$$

$$= \left(\mathbf{S} \otimes \mathbf{S} \otimes \mathbf{S} \otimes \mathbf{u}\right)^{limjnk}{}_{lmn} \tag{2c}$$

In CFDlang, Equation 2c could be represented as $S \# S \# S \# u\,.\,[[1\,6]\,[3\,7]\,[5\,8]]$. Here $S \# S \# S \# u$ is the outer product of all tensors involved in the contraction. The dimensions of this product tensor are numbered from 0-8. The index pairs in the square brackets then specify which dimensions are reduced in the contraction. In addition CFDlang supports most of the tensor operations typically used for CFD simulations such as *tensor contractions* (cf. 1a and 1c), *inner* and *outer products*, and *entry-wise multiplication* (cf. 1b).

Figure 1 shows a description of the complete Inverse Helmholtz operator in CFDlang. Lines 1-6 describe all required tensors including the intermediate values for t and r. Lines 7 and 9 show a tensor contraction and line 8 contains a Hadamard product. In CFDlang the program does not determine the order of operations and the exact implementation, allowing the compiler to optimize the operations for a particular target.

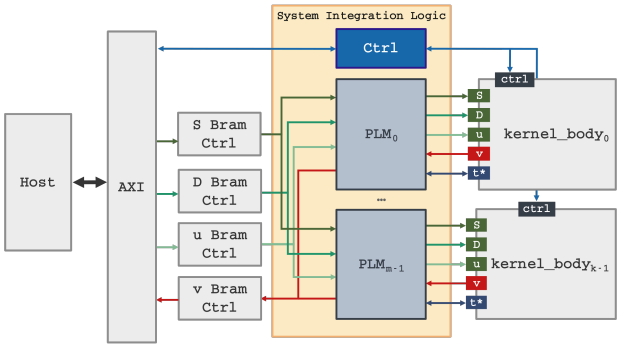


Fig. 2. Target system instance for the Inverse Helmholtz: we replicate the accelerator (along with its PLM) operator multiple times for parallel execution.

III. System Architecture and Methodology Overview

A. System-level FPGA-based Design for CFD Simulations

From the system-level perspective, the CFD simulation runs on the host, which sends the kernel data to the FPGA (`S`, `D`, and `u`) and retrieves the output (`v`) after the kernel execution. Since CFD simulations are massively parallel, we can parallelize multiple elements. We design each accelerator by combining HLS and Private Local Memory (PLM) optimization tools. This allows us to optimize the two parts independently and replicate them based on the amount of FPGA resources requested by HLS. Figure 2 shows an example where $m = k$ and so each PLM instance (for one element) is associated with the corresponding kernel. If $k < m$ (e.g., $m = 16$ and $k = 4$), the same accelerator operates on consecutive PLM elements.

B. Decoupled CFDlang-to-Bitstream Flow

We propose a modular tool flow that simplifies the creation of FPGA accelerators for numerical simulations directly from CFDlang. Concretely, it helps the user optimize intra-kernel and inter-kernel parallelism, and host-accelerator interfacing.

Figure 3 shows an overview of our flow. The CFDlang's representation gives us fine-grained control to rearrange data accesses or modify the number of parallel accesses. In this work we extend the CFDlang compiler infrastructure with FPGA-specific optimizations and hardware generation. We added a polyhedral engine, using libISL [9], for intra-kernel transformations for HLS and memory optimization.

The data layout and the kernel implementation generated by the compiler are then optimized separately. We use commercial HLS (currently Vivado HLS) to generate the accelerator

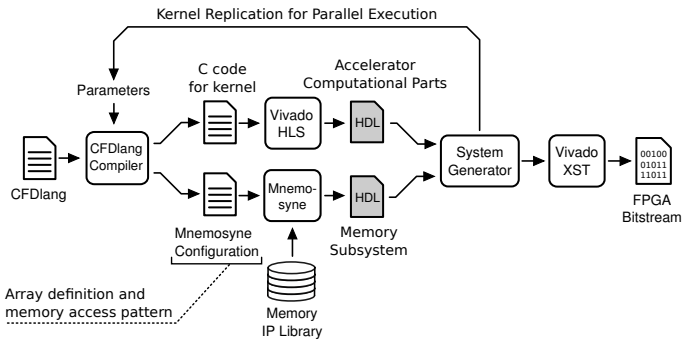


Fig. 3. Tool flow from CFDlang to FPGA bitstream generation.

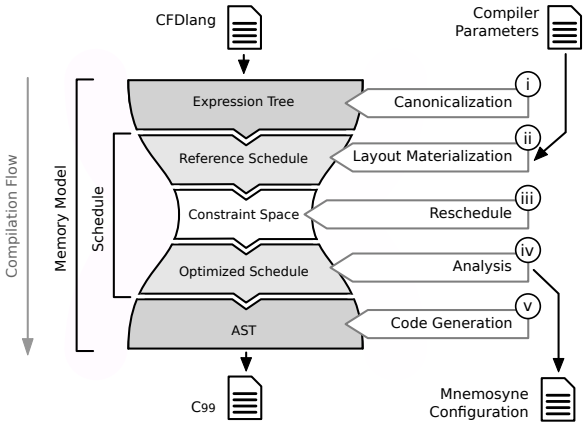


Fig. 4. The extended CFDlang compilation flow. This diagram shows the different levels of abstraction and the operations performed on them.

implementation, from C code generated from the compiler. We use high-level operator information in the DSL and leverage polyhedral analysis to fine-tune the generated code so that it is amenable for HSL (cf. Section IV). This source-to-source approach allows us to profit from excellent results from HLS tools for the computational part. Classic HLS tools, however, have limited support for implementing multi-bank, multi-port memories. For this reason we use Mnemosyne [22], which takes over the generation of the memory architecture for the accelerator and supports us in the effective use of FPGA BRAMs. We modified the CFDlang compiler to automatically create the Mnemosyne input metadata during the compilation (cf. Section IV-F). This is crucial since the compiler can support sophisticated partitioning or sharing of data among multiple memory banks through code analysis.

The system generator in Figure 3 automatically creates the logic for replicating the kernels (produced by HLS) and the memories (produced by Mnemosyne). The tool flow finally produces the artifacts for interfacing with bitstream generation and the corresponding host software to control the accelerators.

From the perspective of an application developer, we enable a seamless integration of the CFDlang in Fortran or C++ code. The kernel with the respective accelerator is then called via a predefined function handle from the surrounding application.

IV. DSL Lowering

A. CFDlang Compiler Extension: Overview

Figure 4 shows an overview of our current CFDlang compiler. As discussed in Section II, the compiler accepts tensor programs such as the Inverse Helmholtz kernel (cf. Figure 1).

The CFDlang frontend creates a simple intermediate representation (IR) that models each statement by constructing an expression tree for the right-hand side (RHS). In this representation, the compiler can detect the independence of reduction dimensions in contraction expressions to exploit associativity. This allows transforming Equation (2c) into an equivalent expression that computes multiple reductions of lower ranks:

$$\mathbf{t} = \Big(\mathbf{S} \otimes \Big(\mathbf{S} \otimes (\mathbf{S} \otimes \mathbf{u})^{cz}{}_{xyz}\Big)^{by}{}_{cxy}\Big)^{ax}{}_{bcx}$$

These transformations operate entirely on the IR and are the basis for existing CFDlang optimizations.

As shown in Figure 4, we extended the compiler by gradual abstraction, incrementally lowering the input to a more flexible and concise constraint-based description. We apply existing transforms during step ⓘ before introducing any new abstractions. Our flow can further specialize this abstract representation to achieve more HLS-friendly kernels. The process stops once a rigid C99 implementation has been reached.

B. Modelling Tensor Values

After having produced a pseudo-SSA form in step ⓘ, all expressions are fixed. The order in which individual elements of these expressions are computed is still undefined, and the IR does not reflect the per-element dependencies. Therefore, we introduce a value-based abstraction of tensor expressions, which differs from memory-based methods, such as the typical `memref`-based usage of the `linalg` dialect in MLIR [17].

As the language only knows statically shaped non-aliasing tensor values, we can reference any particular element of any tensor using an index tuple. Since we build on *isl*, we use its notation. For example, the set of index tuples for the tensor $\mathbf{t}$ in the kernel from Figure 1 is written as:

$$\{\mathbf{t}[\underbrace{i,j,k}_{\text{index tuple}}] : 0 \le i < 11 \text{ and } 0 \le j < 11 \text{ and } 0 \le k < 11\}$$

Every tensor spans its own, unique index space with its rank setting the number of dimensions. This also applies to scalars, which are modelled as 0-dimensional, and thus have exactly one valid "index" each. When reasoning about types, we refer to any tensor index space using the shorthand *tensor*[...].

Every expression in the IR defines all elements of a unique tensor via the RHS expression. There are named tensors that appear on the left-hand side of an assignment, which may either be part of the kernel interface (cf. **input** and **output** in Figure 1) or local temporaries like $\mathbf{t}$. All other expressions define transient (a.k.a. virtual) tensors without an explicit name.

We can examine assignments via mappings of data dependencies *tensor*[...] $\to \bigcup$ *tensor*[...] from output to operand tensor

elements. For the Hadamard product in Line 8, $\mathbf{r} = \mathbf{D} \circ \mathbf{t}$, we obtain elements mappings from $\mathbf{r}$ to $\mathbf{D}$ and $\mathbf{t}$:

$$\mathbf{r}[i,j,k] \mapsto \mathbf{D}[i,j,k] \cup \mathbf{t}[i,j,k]$$

We compute this mapping, called the *operand map*, for every tensor expression by transitive application.

Consider the contraction expression on Line 7, the Equation 2c. From the internal structure of the reduction, we can define an inner domain for the expression that includes the reduction indices $\{\mathbf{t}[i,j,k,\alpha,\beta,\gamma] : \ldots\}$, from which we then construct an inner operand map:

$$\mathbf{t}[i,j,k,\alpha,\beta,\gamma] \mapsto \mathbf{S}[i,\alpha] \cup \mathbf{S}[j,\beta] \cup \mathbf{S}[k,\gamma] \cup \mathbf{u}[\alpha,\beta,\gamma]$$

To obtain the composable mapping over the outer, output tensor domain, we project out these indices.

C. Computing a Reference Schedule

In a polyhedral model, a statement $stmt[\ldots]$ is a space over some integer control variables, and its points are called *instances*. A schedule $S : stmt[\ldots] \to [\ldots]$ maps these instances to the schedule space, which is an anonymous integer tuple space that reflects an executable loop program structure. These tuples impose a total ordering via lexicographical comparison, enabling a mathematical abstraction for code transformations.

We promote every assignment to a statement, allowing us to leverage existing schedule optimizations for tensor expressions. The order of the domain elements is not fixed by the CFDlang program, but there is an implicit reference schedule that defines what orders are valid. We construct the reference schedule from the assignments and their operand maps to enable layout-aware transformations. During construction, we use the operand maps to avoid materializing transients, and inner domain maps to lower reductions into schedule space.

D. Layout Materialization

In step (ii) of Figure 4, we use the reference schedule to concretize tensor memory layouts as pre-optimization. This allows us to adapt to external constraints, such as the host memory layout, and to make use of array partitions during rescheduling. This differs from typical polyhedral approaches that perform the layout independently from their scheduling. Instead, we use a model-driven construction of the layouts through command-line options, and modify our schedule accordingly. Such options include *layout expressions* which map tensors to arrays. An array is a one-dimensional index space $array[i]$, later implemented using concrete platform memory. For example, the C99 standard innermost dimension layout of $\mathbf{t}$ reads $\mathbf{t}[i,j,k] \mapsto \mathtt{t}[121i + 11j + k]$. Every tensor must have an affine layout, and we default to the row major layout. These expressions can also be used to implement implicit reshaping as is commonly done in host-device interfaces.

Options include also *partitioning maps* which map arrays to arrays. These mappings can declare relations of the very general type $\bigcup array[i] \to \bigcup array[o]$, provided that their union has an injective fixpoint. This means that they can, in fact, split *and* merge arrays, despite the name. This allows non-surjective mappings, which can be used to implement explicit address-space sharing if the transformation is legal (cf. Section V-A2).

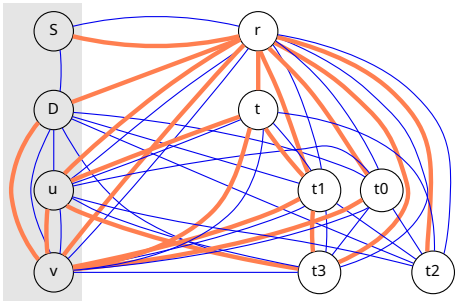


Fig. 5. Graph of the memory-interface and address-space compatibilities for Inverse Helmholtz kernel. Interface arrays are grouped on the left.

The reference schedule reflects these mappings after their application by transforming the statement data dependencies, and splitting the statements to operate over disjoint sets of array partitions. As a result, the subsequent rescheduling process can independently schedule computations in different array partitions regardless of the original expression structure.

E. Rescheduling and Code Generation

In step (iii), we use isl's Pluto scheduler to compute schedules from the constraints derived from the reference schedule. We obtain these constraints through layout-aware dataflow analysis. We use read-after-write (RAW) dependencies as cost function in the isl rescheduler to reduce the dependence distance and thus the live intervals. Read-after-read (RAR) dependencies also feed a cost function that attempts to place the statements at coincident schedule space points. This helps reduce the pressure on temporary storage.

Finally, step (v) calls isl's code generator to produce a C99 program that implements the computed schedule. Our precomputed operand maps simplify the generation of the expressions for each element.

F. Liveness Analysis

To optimize the memory architecture, Mnemosyne needs external information on the memory interface. Based on this, it applies sharing transformations based on a memory compatibility graph, which we can easily compute from the CFDlang program for any given schedule. We perform these analyses and generate such metadata in step (iv).

Figure 5 shows a memory compatibility graph derived from a valid schedule of the kernel in Figure 1. In this graph, nodes represent arrays, with the edges indicating sharing potential. Two arrays are address-space compatible if their lifetimes do not overlap for the entire execution of the accelerator. Two arrays are memory-interface compatible if it is possible to define a total temporal ordering of the memory operations such that the same type (either read or write) never happens at the same time on both arrays.

Dataflow analysis returns RAW dependencies in the form:

$$\mathtt{RAW} : array[i] \to [write[\ldots] \to \mathrm{read}[\ldots]]$$

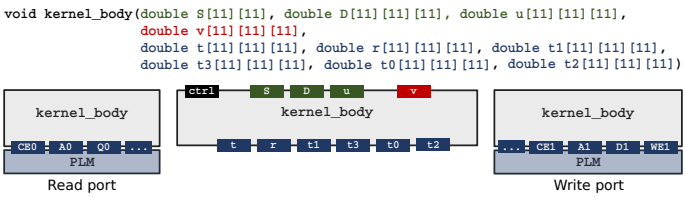


Fig. 6. Generation of accelerator kernel from C code: optimized PLM units store the arrays and are accessed with standard memory ports. For readability, the interface uses multi-dimensional arrays instead of flattened 1-D arrays.

This relation maps *array* elements that transport a value from the statement *write* to the statement *read*. By definition, the value at the specified array element is live at all schedule space points between these statements. By applying the schedule S to both statements, we transform the RAW dependencies into the liveness intervals $I = (S \times S) \circ \texttt{RAW}$ over schedule space tuples:

$$I : array[i] \rightarrow [[\ldots] \rightarrow [\ldots]]$$

Correctly inferring the liveness of **input** and **output** arrays requires a modified virtual schedule. In this schedule, two statements `first` and `last` are defined, modelling writes to inputs and reads from outputs.

Since the schedule space tuples are lexicographically ordered, we can define a second-order helper function `ge_le` that turns a mapping from one tuple to another into a set of all tuples between them. Finally, we obtain $L = \texttt{ge\_le} \circ I$, mapping every array element to the set of schedule tuples at which it carries a live value.

$$\texttt{ge\_le} : [[\ldots] \rightarrow [\ldots]] \rightarrow [\ldots]$$
$$L : array[i] \rightarrow [\ldots]$$

To determine whether two arrays are address-space compatible, one must now simply determine whether their images in L are disjoint.

V. Generation of Hardware Architecture

Our compiler-based flow can generate optimized FPGA architectures with several accelerators executing in parallel on different elements (cf. Section II). To do so, we divide the creation of the target system (cf. Section III-A) in two steps. In the first step, we generate the accelerator logic (*kernel body*) and the memory subsystem for a single kernel starting from the artifacts generated by the DSL compiler (cf. Section V-A). In the second step, we create a parallel architecture by replicating the memories and the kernels as many times as they can fit into the given FPGA (cf. Section V-B). Then, we generate the logic for coordinating the execution and the memory accesses to the different memory instances, along with the corresponding software counterpart for configuring and executing the entire CFD simulation.

A. Kernel Generation

To separate the generation of the computational part and the PLM units we export all memory elements from the accelerator. The compiler transforms each memory element (e.g., array or tensor) into an interface parameter of the code to be synthesized. Figure 6 shows an example of resulting C prototype and the corresponding hardware interface generated by HLS. We implement each array with a PLM unit, i.e., a set of BRAMs that can store the corresponding data (e.g., 121 64-bit elements for array `S`) and the logic and ports to implement the required read and write accesses. Mnemosyne creates shared PLM units exploiting compiler information (cf. Section V-A2).

1) High-Level Synthesis: We use commercial HLS tools to generate the RTL code from the C code produced by the compiler. When using uni- and multi-dimensional arrays as input parameters, existing HLS tools assume the memory is outside the component. They generate a standard memory interface, assuming fixed latency when scheduling memory accesses. We can apply state-of-the-art HLS optimizations (i.e., loop unrolling and pipelining) since they are independent of the memory interface. Array partitioning can be also applied to increase the parallelism, demanding multi-port memories that we manage during memory architecture generation.

2) Memory Architecture Generation: Each memory element is implemented outside the accelerator on BRAMs. We apply memory sharing to reduce the BRAM requirements of each kernel. To this end, we exploit the information computed during liveness analysis (cf. Section IV-F). Mnemosyne uses this information to generate zero-conflict memory architectures while guaranteeing fixed latency of the memory accesses. It can also create multi-port, multi-bank architectures based on the requested HLS optimizations.

B. System Generation

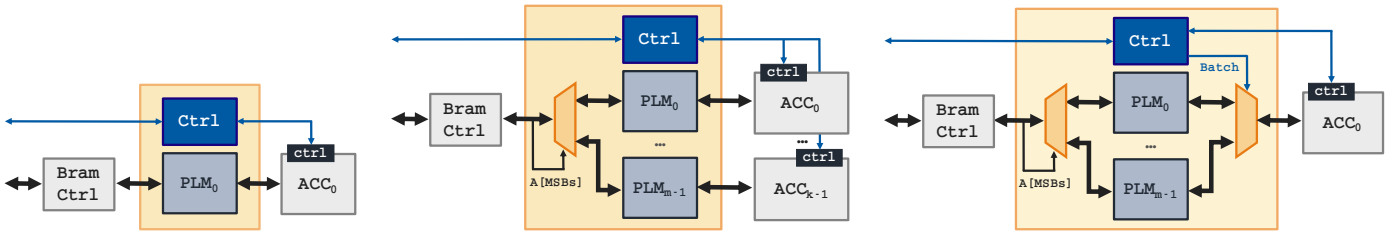
After generating the accelerator and the corresponding optimized memory subsystem, we can compute how many replicas can fit into the given FPGA. After reserving FGPA resources for interfaces (e.g., AXI controllers), which can be easily pre-characterized, we can define the set of resources A available for the accelerators and extra routing logic. We can then estimate the resource requirements of the HLS accelerator (H) and its memory (M) from the reports. So, our system must respect the following equation:

$$[H] \cdot k + [M] \cdot m \leq [A] \tag{3}$$

We assume $m \geq k$, since accelerators can only execute in parallel if they each have a memory architecture to work with. To simplify the logic around the accelerators and the PLM units, m must be a power-of-two multiple of k. This constraint greatly simplifies the system integration logic.

We developed a tool to read the kernel and memory interfaces, the CDFlang metadata, and the board information to automatically create 1) the accelerator instances, 2) the logic to drive the data from the host to the different PLM units and vice versa, and 3) the system description ready for logic synthesis along with the corresponding software host code.

Given the number of accelerator (k) and memory (m) replica, we determine how many times to execute each accelerator (parameter $batch = m/k$) and, in turn, how to connect the



(a) $m = k = 1$ (b) $m = k > 1$ (c) $m > k$

Fig. 7. System architecture variations based on m and k.

modules. If $k = m$, the memory ports are simply connected between accelerators and memories of equal index, as shown in Figure 7b. Each accelerator operates on a single PLM element. If $k < m$, each kernel operates on $batch$ memories, as shown in Figure 7c. For instance, if $k = 2$ and $m = 4$, we have that $batch = 2$ and each accelerator (ACC) operates on two PLMs. In the first execution, ACC_0 will access PLM_0 and ACC_1 will access PLM_2. On the second execution ACC_0 will access PLM_1 and ACC_1 will access PLM_3. This architecture can amortize long setup times for the data transfers.

The CPU host communicates with the accelerator using an AXI-lite memory mapped interface and an interrupt line. Each of the k accelerators use the `ap_ctrl` interface which consists of a `ap_start` control input, and `ap_done`, `ap_idle`, `ap_ready` status outputs. To be able to control k accelerators using a single AXI-lite interface to the CPU, we implemented an AXI-lite peripheral to receive the AXI transactions and update the memory mapped registers as if the CPU was interacting with a single kernel generated by HLS with an AXI-lite control interface. To start execution, the host writes a `start` command to a memory mapped register. When all of the k accelerators are ready to begin, the AXI-lite peripheral broadcasts the `start` signal to all accelerators. Once each of the k accelerators has signaled that it is done processing, the AXI-lite peripheral raises the interrupt line back to the CPU. When a round is completed, the `batch` counter is incremented up to m/k. The `batch` counter is then forwarded to the memory integration logic, as shown in Figure 7c.

The CPU host code executes the accelerator for the total number of elements in the CFD simulation (N_e), requiring N_e/m main loop iterations. This loop includes the input data transfers, execution, and output data transfers. The CPU transfers the input array data for m points through the AXI interface. m instances of each array are transferred to power-of-two aligned addresses. Then, in a loop which executes m/k times, the start command is sent over the AXI-lite control interface, triggering the execution of k accelerators. The CPU waits for the done interrupt. After this loop is finished, m points are complete and ready in the output memories. The data is transferred in m output arrays available to the CPU.

VI. Evaluation

We implemented a prototype of our DSL-to-FPGA tool flow for the CFD simulation targeting the Xilinx Zynq UltraScale+ MPSoC ZCU106 board. This system allows us to get preliminary feedback on the challenges of FPGA acceleration for such workloads. The board features a quad-core ARM Cortex-A53 and a xczu7evffvc1156-2 FPGA, which has 504K system logic cells (around 230K LUTs and 460K FFs) and 312 block RAMs. We use Xilinx Vivado HLS 2019.2 for kernel synthesis and Mnemosyne for the optimization of the accelerator's memory. We developed an in-house tool for the generation of the system integration logic, the FPGA system description, and the host code. We added hardware timers to measure the execution time of the kernel computation with and without the data transfers. We demonstrate the flow on the Inverse Helmholtz operator with a polynomial degree equal to $p = 11$.

We used CFDlang to generate different C variants having alternative shapes, layout, and compatibility information. The CFD accelerator kernel requires around 2,314 LUTs, 2,999 FFs, and 15 DSPs. We generated the FPGA systems ignoring sharing compatibilities and using Mnemosyne only as PLM generator. The PLM units for one kernel require 31 BRAMs so we can fit up to $m = 8$ units and so $k = 8$ kernels. However, when enabling compatibilities obtained from liveness analysis (cf. Section IV-F), we can fit up to 16 PLM units and kernels (The PLM units for one kernel now require only 18 BRAMs). Resource usage in all cases (from $m = k = 1$ to $m = k = 16$, when possible) are reported in Table I (including the rest of the architecture), while Figure 9 provides a detailed view on BRAM utilization in all cases. We also generated FPGA systems where the temporary arrays were left inside the HLS accelerator. In these cases, the memory system used 9 BRAMs and the accelerator used 24, for a total of 33 BRAMs, showing that exporting the temporary arrays to allow control over their implementation does allow for better optimization. All kernels are synthesized at the target frequency of 200 MHz. We executed a prototypical CFD simulation of 50,000 elements with all data in DRAM.

First, we tested $k < m$ variants to determine if larger data transfers can reduce communication latency. These experiments did not show much improvements due to limitations in the current implementations of the data transfers. So, we performed all remaining tests with $k = m$.

Figure 9 shows the speed up that we achieved with our parallel architectures. Since memory sharing is transparent to accelerator execution, the values are the same for the two sets

TABLE I
RESOURCE UTILIZATION FOR NO MEMORY SHARING AND MEMORY SHARING ARCHITECTURES

	m, k	LUT		FF		DSP	
No Sharing	1	11,318	(4.9%)	9,523	(2.1%)	15	(0.9%)
	2	15,929	(6.9%)	12,583	(2.7%)	30	(1.7%)
	4	25,728	(11.2%)	18,663	(4.1%)	60	(3.5%)
	8	42,679	(18.5%)	30,795	(6.7%)	120	(6.9%)
Sharing	1	11,292	(4.9%)	9,533	(2.1%)	15	(0.9%)
	2	15,572	(6.8%)	12,596	(2.7%)	30	(1.7%)
	4	24,480	(10.6%)	18,663	(4.1%)	60	(3.5%)
	8	42,141	(18.3%)	30,782	(6.7%)	120	(6.9%)
	16	77,235	(33.5%)	55,053	(12.0%)	240	(13.9%)

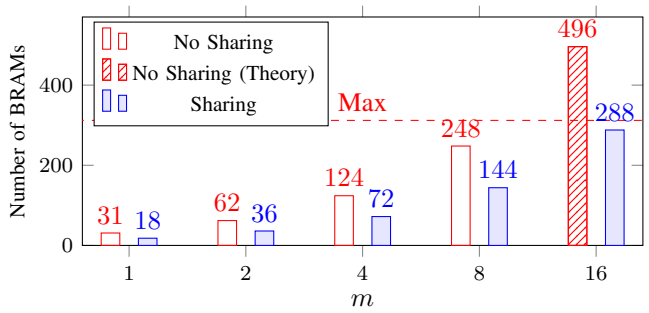


Fig. 8. BRAM utilization of parallel accelerators w/- and w/o memory sharing.

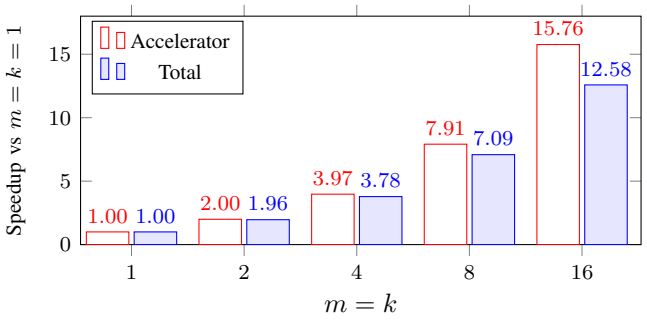


Fig. 9. Accelerator and total speedup for parallel architectures.

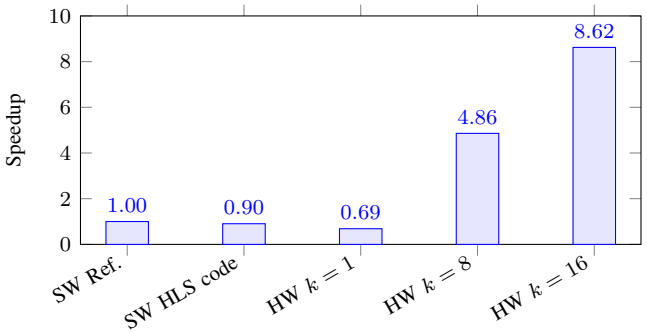


Fig. 10. Speedup compared to software execution on ARM A53.

of experiments, except in case $m = k = 16$ which is possible only with memory sharing. We can see that, as expected, the speedup for accelerator execution is nearly the ideal, k. The total speedup is lower due to the communication overhead but can reach up to 12.58× in case of 16 kernels.

For a fair comparison with software execution, we executed a reference implementation of the operator on the ARM A53 CPU available on the ZCU106 (*SW Ref.* in Figure 10). This is the same CPU which performs the data transfers in the hardware execution tests and is configured to run at 1.2 GHz, which is 6× faster than the kernels running on FPGA. Figure 10 shows the comparison for all experiments. The C code given as input to HLS (*SW HLS Code*) is slower on CPU. We also compared the software with the hardware execution with a variable number of kernels (*HW* $k = 1$, *HW* $k = 8$, *HW* $k = 16$). In case of *HW* $k = 1$, the code has 30% slowdown compared to the software execution because of the faster ARM frequency and the CPU-FPGA data transfers. The best architecture, *HW* $k = 16$, executes up to 8.62× faster than the CPU. From the CFD expert viewpoint, all results have been achieved by writing only 9 lines of DSL (Figure 1) and no particular hardware knowledge (except from board resources).

VII. RELATED WORK

We focused our work on spectral element methods which use tensor expressions. Some of existing DSLs such as Tensor Flow Eager [2] are based on software libraries like Tensor Flow or Theano [1, 3]. These approaches raise the abstraction level, but they offer limited flexibility in the back-end. DSLs such as Tensor Comprehension (TC) or TVM, on the other hand, were developed to optimize for various platforms. While TC specializes in GPUs, TVM also supports FPGA back-ends. However, this back-end is based on a template architecture and does not offer flexibility for custom tensor expression or for replicating the respective kernels, as also in [11].

Halide is a representative DSL for stencil operations [25], while Halide-HLS [24] and HeteroHalide [18] offer hardware back-ends. Halide separates computation and scheduling for image processing. This allows applications to be flexibly suited to the target architectures, but also shifts the burden of understanding the platform to the application developer.

We leverage the polyhedral model for the scheduling of tensor operators, which was a useful approach in many domains [8, 36]. We use the polyhedral model to determine access patterns, lifetime ranges, and streaming constraints [37, 35]. An automatic flow to generate systolic arrays from the polyhedral model has been proposed in [38]. Polyhedral models can support the interplay of polyhedral optimizations and hardware back-ends with the memory subsystem.

We extended the CFDlang compiler [28, 32] to target hardware generation. The IR uses primitive operations without any notion of domain semantics that can be found instead in ML-specific approaches such as TPP [10]. Modern frameworks such as MLIR [17] can ease the combination of independent DSL frontends, tensor middle-ends, and hardware backends. Via adapters such as Teckyl [33], one can already construct flows that consume TC [34] and process them entirely within MLIR. The MLIR community is missing a value-based tensor dialect, as opposed to the memory-reference based `linalg` dialect. We foresee a similar dialect, possibly based on TeIL.

The design of domain-specific accelerators for scientific

simulations demands an efficient distributed computing model. Approaches such as [29] use DSLs to explicitly encode inter-kernel pipelining and parallelism on a device. When evaluated for CFD codes in [20], partitioning and communication planning have impact on the scalability of these implementations. We focus instead on increasing the utilization and throughput of a single device guided by kernel resource usage.

While HLS simplifies the creation of hardware accelerators [21], memory optimization, system-level integration, and programmability are still open challenges. Several HLS optimizations can improve the use of local memories or physical banks during computation [6, 22]. Indeed, PLMs dominate the resource requirements, especially in data-intensive accelerators [22]. While independent accelerators can naturally share physical banks, compiler-level analysis can help extracting relevant information about intra-kernel compatibilities. Many existing FPGA architectures, like IBM CloudFPGA [39] and ESP [19], separate the computational parts from the interconnection logic. For example, the ESP *services* allow designers to integrate accelerators without any impact to the rest of the system. We extend the same concepts to the local memory architecture. We also enable the creation of parallel architectures. The two approaches are orthogonal.

VIII. Conclusions and Future Work

We presented an end-to-end tool flow to accelerate CFD simulations, combining DSL compiler, commercial HLS and memory optimization tools to seamlessly create a custom accelerator to exploit the intrinsic parallelism of the application. On a Xilinx Zynq Ultrascale+ ZCU106, we deployed 16 parallel kernels, achieving a speed-up of 12.58× over the single-kernel execution and 8.62× over the ARM CPU (which runs 6× faster). These are promising preliminary results that motivate our future work on more advanced DSL tranformations, MLIR integration, better data transfer strategies, and scaling-up to clusters of larger FPGA boards. This is an important step to make FPGA acceleration viable for fluid dynamics.

Acknowledgment

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”. In: *CoRR* abs/1605.08695 (2016).

[2] A. Agrawal et al. *TensorFlow Eager: A Multi-Stage, Python-Embedded DSL for Machine Learning*. 2019. arXiv: 1903.01855.

[3] J. Bergstra et al. “Theano: a CPU and GPU Math Expression Compiler”. In: *Proc. of SciPy*. 2010, pp. 3–10.

[4] T. Chen et al. “TVM: An automated end-to-end optimizing compiler for deep learning”. In: *Proc. of OSDI*. 2018, pp. 578–594.

[5] T. Chen et al. “Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning”. In: *ACM SIGARCH Computer Architecture News* 42.1 (2014), pp. 269–284.

[6] J. Cong et al. “Bandwidth Optimization Through On-Chip Memory Restructuring for HLS”. In: *Proc. of DAC*. 2017.

[7] W. J. Dally et al. “Domain-Specific Hardware Accelerators”. In: *Commun. ACM* 63.7 (June 2020), pp. 48–57.

[8] A. Darte et al. “Lattice-based memory allocation”. In: *IEEE Transactions on Computers* 54.10 (Oct. 2005), pp. 1242–1257.

[9] “isl: An Integer Set Library for the Polyhedral Model”. In: *Proc. of ICMS*. Ed. by K. Fukuda et al. Vol. 6327. 2010, pp. 299–302.

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

[11] D. Giri et al. “ESP4ML: platform-based design of systems-on-chip for embedded machine learning”. In: *Proc. of DATE*. 2020, pp. 1–6.

[12] M. Girkar et al. “Automatic extraction of functional parallelism from ordinary programs”. In: *IEEE Transactions on Parallel and Distributed Systems* 3.2 (1992), pp. 166–178.

[13] 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.

[14] N. P. Jouppi et al. “In-datacenter performance analysis of a tensor processing unit”. In: *Proc. of ISCA*. IEEE. 2017, pp. 1–12.

[15] F. Kjolstad et al. “The tensor algebra compiler”. In: *Proc. of OOPSLA* 1 (2017), pp. 1–29.

[16] D. Koeplinger et al. “Spatial: A Language and Compiler for Application Accelerators”. In: *Proc. of PLDI*. 2018, pp. 296–311.

[17] C. Lattner et al. “Mlir: Scaling compiler infrastructure for domain specific computation”. In: *Proc. of CGO*. IEEE. 2021, pp. 2–14.

[18] J. Li et al. “HeteroHalide: From Image Processing DSL to Efficient FPGA Acceleration”. In: *Proc. of FPGA*. 2020, pp. 51–57.

[19] P. Mantovani et al. “Agile SoC Development with Open ESP”. In: *Proc. of ICCAD*. 2020.

[20] A. Mondigo et al. “Scalability Analysis of Deeply Pipelined Tsunami Simulation with Multiple FPGAs”. In: *IEICE Transactions on Information and Systems* E102.D.5 (2019), pp. 1029–1036.

[21] 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.

[22] 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), pp. 435–448.

[23] C. Pilato et al. “EVEREST: A design environment for extreme-scale big data analytics on heterogeneous platforms”. In: *Proc. of DATE*. 2021, pp. 1–6.

[24] J. Pu et al. “Programming Heterogeneous Systems from an Image Processing DSL”. In: *ACM Transactions on Architecture and Code Optimization* 14.3 (Aug. 2017).

[25] J. Ragan-Kelley et al. “Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines”. In: *Proc. of PLDI*. PLDI ’13. ACM, 2013, pp. 519–530.

[26] A. Reuther et al. “Survey of Machine Learning Accelerators”. In: *Proc. of HPEC*. 2020, pp. 1–12.

[27] 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.

[28] N. A. Rink et al. “TeIL: a type-safe imperative tensor intermediate language”. In: *Proc. of ARRAY*. 2019, pp. 57–68.

[29] K. Sano. “DSL-based Design Space Exploration for Temporal and Spatial Parallelism of Custom Stream Computing”. In: *arXiv:1509.00040 [cs]* (Aug. 27, 2015). arXiv: 1509.00040.

[30] T. Santos et al. “Automatic Selection and Insertion of HLS Directives Via a Source-to-Source Compiler”. In: *Proc. of FPT*. 2020.

[31] N. Srivastava et al. “T2S-Tensor: Productively Generating High-Performance Spatial Hardware for Dense Tensor Computations”. In: *Proc. of FCCM*. 2019, pp. 181–189.

[32] A. Susungi et al. “Meta-programming for Cross-Domain Tensor Optimizations”. In: *Proc. of GPCE*. Nov. 2018, pp. 79–92.

[33] *Teckyl: An MLIR frontend for Tensor Operations*. https://github.com/andidr/teckyl. Accessed: 2021-05-28.

[34] N. Vasilache et al. *Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions*. 2018. arXiv: 1802.04730.

[35] S. Verdoolaege et al. “Consecutivity in the isl Polyhedral Scheduler”. In: (2017). Publisher: 10330.

[36] S. Verdoolaege et al. “Polyhedral Parallel Code Generation for CUDA”. In: *ACM Trans. Archit. Code Optim.* 9.4 (Jan. 2013).

[37] S. Verdoolaege et al. *Scheduling for PPCG*. June 23, 2017.

[38] J. Wang et al. “AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA”. In: *Proc. of FPGA*. 2021, pp. 93–104.

[39] J. Weerasinghe et al. “Network-attached FPGAs for data center applications”. In: *Proc. of FPT*. 2016, pp. 36–43.