Python FPGA Programming with Data-Centric Multi-Level Design

Johannes de Fine Licht*, Tiziano De Matteis*, Tal Ben-Nun*, Andreas Kuster* Oliver Rausch*, Manuel Burger*, Carl-Johannes Johnsen[†]*, Torsten Hoefler*

*Department of Computer Science, ETH Zurich, [†]Department of Computer Science, University of Copenhagen

Abstract—Although high-level synthesis (HLS) tools have significantly improved programmer productivity over hardware description languages, developing for FPGAs remains tedious and error prone. Programmers must learn and implement a large set of vendor-specific syntax, patterns, and tricks to optimize (or even successfully compile) their applications, while dealing with ever-changing toolflows from the FPGA vendors. We propose a new way to develop, optimize, and compile FPGA programs. The Data-Centric parallel programming (DaCe) framework allows applications to be defined by their dataflow and control flow through the Stateful DataFlow multiGraph (SDFG) representation, capturing the abstract program characteristics, and exposing a plethora of optimization opportunities. In this work, we show how extending SDFGs with multi-level Library Nodes incorporates both domain-specific and platform-specific optimizations into the design flow, enabling knowledge transfer across application domains and FPGA vendors. We present the HLS-based FPGA code generation backend of DaCe, and show how SDFGs are code generated for either FPGA vendor, emitting efficient HLS code that is structured and annotated to implement the desired architecture.

1 INTRODUCTION

The widespread adoption of HLS tools have greatly improved programmer productivity when targeting FPGAs by allowing kernels to be developed in C++ or OpenCL [1]. Still, achieving efficient architectures remains challenging in practice, as the optimization space of hardware design is larger than for software; requiring new syntax, new ways to structure programs, and new transformations [2]. In this work, we propose employing the *Data-Centric parallel programming* (DaCe) framework and its *Stateful DataFlow multiGraph* (*SDFG*) [3] intermediate representation as an alternative way to develop FPGA programs. The data-centric representation enables explicit management of data location and movement, which remains the biggest performance factor in computing today [4].

SDFGs allow representing programs by their dataflow and control flow independent of the chosen FPGA backend, enable compatibility across FPGA vendors through code generation, and are amenable to optimizing transformations performed directly on the graph. SDFGs have been proven effective for load/store workloads in various domains, ranging from linear algebra kernels and graph algorithms [3] to numerical weather prediction [5] and supercomputer-scale quantum transport simulations [6]. When FPGA SDFGs are manually authored [3], their performance is on-par with state-of-the-art implementations and libraries.

Modern compiler techniques, such as polyhedral optimization, allow sophisticated automatic optimization of low-level IR by detecting and transforming loop constructs [7], but is restricted to the space of transformations that can be proven to be "safe". Rather than relying on fully automated optimization to exploit all available opportunities, DaCe exposes powerful performance analysis ca-



Fig. 1. Proposed development workflow using the DaCe framework.

pabilities and optimization tools that enable knowledgeable performance engineers to perform *guided* optimization of programs. Transformations are done via graph rewriting on the SDFG representation, expressed in terms of the general dataflow and control flow of the program, thus facilitating knowledge exchange between programs and domains.

Domain-specific languages can enable additional optimizations by restricting the input domain, allowing additional assumptions to be made on the program's behavior, but typically offer limited exchange of knowledge and engineering effort with other domains. In this work, we show how the "Library Node" extension to SDFGs, first prototyped by StencilFlow [5], enable a multi-level design methodology that exposes the best of both worlds within the same framework, enabling the application of both domainspecific and general purpose optimizations to DaCe programs.

Throughout this work, we describe how the proposed methodology enables productively developing fast FPGA programs by utilizing a multi-level approach:

- SDFGs natively expose key aspects of hardware, such as pipelining, streaming, and systolic arrays.
- DaCe's FPGA backend targets both Xilinx and Intel FPGAs with structured, annotated HLS code, en-



Fig. 2. Glossary of nodes and edges in the SDFG representation. This work extends SDFGs with Library Nodes, highlighted in green.

abling reuse between vendors.

- DSL frontends, such as BLAS, ONNX, or Stencil-Flow [5], allow expressing programs at a high abstraction level, comprehensible to non-FPGA experts.
- High-level domain-specific transformations allow applying domain knowledge to optimize programs.
- General mid-level transformations allow porting from CPU to FPGA, and perform key optimizations, such as replacing memory accesses with streaming.
- Low-level specialization allow employing vendorspecific tweaks, targeting non-universally supported features such as shift registers or accumulators.

2 **Representation and Code Generation**

In the following, we provide an overview of how SD-FGs represent FPGA programs, and how key concepts are translated into fast HLS code for the two major vendors: Xilinx, through the Vivado HLS [8] C++-based compiler; and Intel, through the Intel OpenCL SDK for FPGA [9]. The components of SDFGs that we will use are summarized in Figure 2, and we give a brief primer here.

Access nodes represent data containers in dataflow states of the SDFG (drawn as ovals with solid (arrays) or dashed (streams) borders), where they are accessed using memlets, annotated on dataflow edges. Computations are performed in small tasks called tasklets (drawn as hexagons), which can only access memory explicitly passed to them via dataflow edges, capturing all data movement in the program. Maps implement explicit parallelism in the graph, representing parametric replication of the content between map entry nodes (opening the scope) and map exit nodes (closing the scope), drawn as trapezoids. Coarse-grained control flow is represented by control flow graphs, where nodes are states that only contain pure dataflow. Control flow and dataflow graphs can be arbitrarily nested in each other using nested SDFGs to represent arbitrary program semantics, while still exposing as much analyzable data movement as possible. Operational semantics can be found in the initial paper [3].

This work uses **Library Nodes**, a special version of the tasklet representing *abstract behavior*, which is "expanded"/"specialized"/"lowered" (we will use these functionally equivalent terms interchangeably) into parametric *subgraphs* that implement the abstract behavior [5]. An example SDFG targeting FPGA is shown in Figure 3, which we will cover in detail in the following sections.

2.1 Code Generating SDFGs

SDFGs are constructed, manipulated, and compiled using the DaCe framework. DaCe follows the guiding principle that as many optimization opportunities as possible should be kept part of the representation — where they can be manipulated by the performance engineer — rather than happening as "magic" during code generation. Nevertheless, emitting functional and efficient code from SDFGs poses a significant design and engineering challenge, with numerous kinks and subtleties arising from moving from software to the hardware domain. The code generator must translate the final representation into structured HLS code that is easily digestible by the compiler, faithfully follows the functional semantics of the SDFG, and that successfully achieves all parallelism implied by the representation.

The FPGA backend of DaCe is modularized into a generic part, which orchestrates the traversal of the SDFG, and two lower level specialized backends for Xilinx and Intel, which are responsible for emitting vendor specific code for Vivado HLS (C++) and the Intel OpenCL compiler, respectively. The generic backend contains the most sophistication in terms of interpreting the representation and delegating code generation tasks, whereas the two specialized components are primarily concerned with vendorspecific semantics (e.g., how processing elements and memory interfaces are expressed) and syntax (e.g., vector data types and stream objects). In particular, the highly restricted syntax supported by OpenCL requires more verbose syntax to be emitted than for backends that support C++. DaCe also supports tasklets with embedded RTL code: in this case, both HLS and RTL kernels are generated, and then combined together in the final bitstream [10].

2.2 Parallelism, Pipelining, and Unrolling with Maps

Representation. Parallel sections in SDFGs are expressed via the *map* construct, appearing as a pair of entry/exit nodes opening/closing scopes (one map is present in the black box, and two in the purple box in Figure 3). In software, these scopes can be used to target multi-core and SIMD parallelism for both CPUs and GPUs. In hardware, we distinguish between two ways of exploiting the parallelism implied by maps: *pipelined* maps, where iterations are



Fig. 3. Kernel state with four processing elements (right), with pre- and post-states (left) copying memory between host and device.



Fig. 4. Processing elements are function calls in a toplevel function in the Vivado HLS paradigm. host code in the Intel OpenCL paradigm. cluding a parametrically sized systolic array.

executed in sequence, but exploit pipeline parallelism in the mapped computation; and *unrolled* maps, which represent parametrically replicated hardware, such as systolic arrays (see Section 2.6) or SIMD-style vectorization. The purple box in Figure 3 contains an inner map, which will be generated as a pipelined inner loop, and an outer map over tiles, orchestrating the buffering behavior.

Code generation. In an SDFG with any number of nested map scopes, code generation follows the philosophy that anything that is not explicitly unrolled should be pipelined. The Intel OpenCL compiler does this automatically, but for the Xilinx backend, the graph is traversed from outermost to innermost nesting to detect the innermost map *that is not unrolled*, where a pipeline pragma will be injected inside the generated loop. Furthermore, loop coalescing pragmas are automatically injected whenever loops generated from maps are perfectly nested, and when necessary, pragmas to ignore dependencies (see Section 2.7). Maps designated as being unrolled will annotate the generated loops with the vendor-specific unroll directive.

2.3 Representing FPGA Kernels

Representation. The pure dataflow representation of SDFG states is a natural fit for mapping to streaming dataflow kernels on FPGA. When traversing the SDFG, the framework detects states that only access memory situated on the FPGA, designating these as FPGA kernels. Although kernels are always mapped from pure dataflow states, coarse-grained control flow is still achievable by embedding *nested SDFGs* in the dataflow state. Moving data between the host and device is represented as memory copies in the representation. Access nodes are annotated with a data location. When connected by direct data-to-data edges in a dataflow state, this will result in the appropriate copy operation depending on both source and destination. Streaming transfers can be natively represented using stream access nodes, but due to the OpenCL abstraction adopted by both the Xilinx and Intel toolflows, the backend currently only supports bulk transfers. Host/device streaming will be introduced once either backend exposes sufficient support to end-users (e.g., using the QDMA [11] subsystem for Xilinx FPGAs).

Code generation. When the code generation traversal encounters an FPGA kernel according to the aforementioned

storage and execution predicate, the dataflow section is dispatched to the FPGA backend. Before continuing the traversal to generate the hardware itself, the kernel "boundary" is generated by inferring the necessary arguments that must be passed to the resulting OpenCL kernel launch(es). Interaction with the OpenCL API is wrapped in the interface provided by the hlslib [12] C++ library, as shown in Figure 5.

2.4 Processing Elements

Representation. The notion of partitioning the functionality of a kernel into multiple independently-scheduled modules, commonly referred to as *processing elements* (PEs), is central to designing large FPGA architectures. Native support for this concept is thus a core consideration in the SDFG representation. At the same time, this should not introduce new FPGA-specific concepts to the representation.

SDFG states imply pure dataflow by representing data movement and data dependencies (e.g., everything contained in a blue rectangle Figure 3 or Figure 6), the latter of which must be respected by the code generating backend. When a dataflow graph contains more than one *weakly* connected component (i.e., at least two subgraphs G_0 and G_1 with no dataflow edge (u, v) connecting any node $u \in G_0$ with any node $v \in G_1$), the backend has the liberty to *schedule* each weakly connected component in parallel. For software backends, this can enable launching multiple concurrent GPU kernels, or running different concurrent tasks on multiple CPU threads. When appearing within an FPGA kernel, these are also scheduled as independent "tasks", exposing the concept of processing elements to the programmer. In the example in Figure 3, each of the four connected components represent an independent processing element scheduled in parallel: the components in the red and the black box are memory reader/prefetcher modules, which read from arrays (solid borders) in off-chip memory into data streams (dashed borders). The red box is a simple copy of the full array dimensions, implemented by a single dataflow edge, where the black box repeats multiple reads of the array, using a map to generate the desired access pattern. The blue box inversely writes from a stream back to memory.

Code generation. In the Vivado HLS toolflow, processing elements are expressed by annotating a scope in the C++ code with the DATAFLOW pragma, resulting in

every loop and function call in the scope to be scheduled as a distinct processing element. This requires a toplevel "entry" function that contains the processing elements, is annotated with additional pragmas that designate the hardware interfaces used by the kernel to interact with the FPGA shell, and instantiates the on-chip streams (i.e., FIFOs) that facilitate inter-PE communication, shown for an example in Figure 4. The Xilinx backend uses the simulation extensions from hlslib [12] (providing the HLSLIB_DATAFLOW_FUNCTION macro wrappers and the thread-safe and bounded hlslib::Stream class) to achieve actually concurrent simulation of parallel processing elements, including support for feedback/back-edges in the dataflow. The Intel OpenCL flow takes a different approach: rather than being contained in a top-level function, every processing element must be expressed as a separate OpenCL kernel in the top-level scope, where they are connected using global channel objects. Launching each processing element is thus done from the host code, shown in Figure 5. These two methods of expressing kernels thus affect both the host code (which kernels are generated and launched) and the kernel code (one vs. multiple top-level kernels, global channel objects vs. local stream objects). If a generated OpenCL kernel has no arguments, it will be generated as an "autorun" kernel, which is always active and will run whenever data is available on the connected channels, and thus does not need to be invoked from the host code.

2.5 Channels/Streams/FIFOs

Representation. Streams are a native data container construct in the SDFG representation, representing first-in, firstout queues, which can be used to communicate between subgraphs in dataflow sections. In CPU and GPU codes, these are employed as single- or multi-producer queues: for example, a breadth-first search kernel can produce tasks to a queue that is consumed by multiple workers, dynamically distributing work. Stream semantics are the same in FPGA kernels, but with additional constraints due to the underlying hardware implementation that they imply: streams cannot be unbounded, and must be single-producer, singleconsumer. Streams facilitate communication between processing elements, while simultaneously acting as synchronization primitives between kernels through the producerconsumer relationship. Even though the components in Figure 3 do not have dataflow edges between them, they synchronize by pushing/popping the same stream data container.

Because all data movement is explicitly captured in the SDFG, programmers can benefit from the information annotated on dataflow edges to verify the correctness of producer/consumer relationships, which are automatically inferred by the tool based on the access pattern expressed by map scopes in the graph. Figure 7 shows the annotation of a dataflow edge written by the processing element prefetching the matrix B from Figure 6 into the stream object B_pipe, and the corresponding read from B_pipe within the processing element. The matrix of size $K \times M$ is read N/P times, where P is the tile size introduced by the systolic array (see Section 2.6), resulting in a data volume of $K \cdot M \cdot \frac{N}{P}$ annotated on the dataflow edge/memlet.



Fig. 7. Data movement volume annotations on the producer (left) and the consumer (right) of a stream, used to verify correctness of the program.

Code generation. Due to the distinct methods of expressing processing elements, the semantics of allocating streams varies significantly between the Xilinx and Intel backends. When generating Xilinx code, streams are emitted in the top-level kernel function as local objects, where they must be passed as arguments to the producer and consumer accessing them (see Figure 4). For Intel OpenCL codes, they must be emitted to the global kernel scope, where the appropriate producer and consumer will read them directly (i.e., rather than receiving them as arguments).

2.6 Parametric Processing Elements: Systolic Arrays

Representation. Systolic arrays [13] are a powerful pattern to express parametric parallelism through deep pipelines, and are the most potent source of parallelism on modern FPGAs [2] when applicable. SDFGs expose this pattern through unrolled maps in the outermost FPGA kernel scope, with a parametric — but compile-time specialized — number of iterations, coupled with *arrays* of stream objects. When such a map is unrolled, each instance semantically becomes a weakly connected component in the state, resulting in them being instantiated as separate processing elements according to the semantics in Section 2.4. This is equivalent to any other map construct in the SDFG: namely, they represent independently executable replications of the contained subgraph (unrolled maps can occur at any level of nesting in the program), but are recognized as a special case in the top-level scope of an FPGA kernel.

An SDFG implementing a one-dimensional systolic array for matrix multiplication ($\mathbf{C} = \mathbf{A} \times \mathbf{B}$) is shown in Figure 6, where the map nodes annotated by red borders instantiate the systolic array. Each element implements the same content (highlighted), but reads from a distinct index in three arrays of stream objects ("pipes") for A, B, and C, respectively. Since every processing element is only connected to the previous and the next, they must pass data along the chain [14] from the head towards the tail. The processing elements implement a simply buffering scheme where each element stores one element of A in a local buffer, then streams over the full B matrix, before writing back a complete output tile of C. This simple SDFG already yields 364 and 188 GOp/s with $8k \times 8k$ matrices when compiled for an Intel Stratix 10 and a Xilinx Alveo U250 board, respectively, with much potential for additional optimization [14].

Code generation. Systolic array code generation varies between vendors due to the different ways of expressing processing elements. In Xilinx codes, it is sufficient to unroll a loop in the C++ kernel code with bounds known at compile time, letting constant propagation fix all the indices in each instantiation to lay out the systolic array, as shown



Fig. 8. Multiple levels of nested Library Node expansions.

in Figure 4. For Intel, the OpenCL kernel itself is replicated and specialized directly in the code generator (see Figure 5).

2.7 Memory Hierarchy

Representation. Not all data movement is born equal: dataflow can have significantly different performance impact depending on the location and storage type of the source and destination, even though the number of bytes moved are the same. The FPGA backend exposes global memory, which represents data present in off-chip, memorymapped storage such as DDR or HBM; local memory, representing any on-chip memory implementation such as registers, BRAM/M20K, LUTRAM, or UltraRAM (left up to the HLS compiler); registers, which is a subset of local memory, but forces the HLS compiler to allow fully parallel read/write access to every entry of the container; and experimental support for shift registers, implementing cyclic buffering patterns with multiple access points (natively supported by Intel OpenCL). Combining these allows implementing highly specialized memory hierarchies, as well as host/device interaction, in a way that is compatible with both Xilinx and Intel devices.

Code generation. Local memories can be emitted as regular C arrays directly in the kernel code, while offchip memory is allocated with API calls in the host code and passed to the kernel arguments. The FPGA backend gives the underlying HLS compilers additional scheduling freedom by generating a distinct pointer argument for every access to the same DRAM memory container present in the kernel and marking them as restrict in OpenCL, such that every read and write can be performed independently, which is safe due to SDFG semantics.

Whenever both reads and writes are emitted to local memory, and the write is not marked as a potential conflict, the generated code is annotated with pragmas to instruct the compiler to ignore dependencies (HLS DEPENDENCE for Xilinx and ivdep for Intel). This is implied by SDFG semantics, where these accesses are either in dataflow sections (where conflicts must be annotated), or in control flow scopes, that are inherently sequentialized.

3 MULTI-LEVEL DESIGN WITH LIBRARY NODES

SDFGs provide a data-centric view of the implemented application, enabling a wide range of optimization opportunities [3], which can be exploited using graph-rewriting

transformations on the SDFG. Some optimizations, however, arise from knowledge about the underlying application domain (for example, algebraic identities), which are difficult or impossible to express generally without encoding domain-specific knowledge into the representation. To accommodate such domain-specific optimizations, we will use Library Nodes to represent an abstract behavior (the "what") on the incoming/outgoing data connectors (as opposed to a concrete implementation of this behavior, the "how"). Library Nodes are expanded by replacing them with a subgraph, "lowering" them towards a concrete implementation of their behavior. During this expansion, Library Nodes can inspect their context using the surrounding memlets and nodes, which may change the structure of the expanded subgraph, e.g., by checking if inputs or outputs are streams or if the use vector types.

For example, in a neural network program, a Library Node can represent a convolution applied to a given input format and producing a given output format. However, *which* convolution is applied, and how/where it is executed, can be deferred, exploiting the high level of abstraction to detect and apply domain-specific transformations. Before an SDFG can be used to generate code, all Library Nodes must be fully expanded to native SDFG constructs, but can go through several levels of lowering [15] before reaching a fully expanded state.

Domain-specific transformations are not the only opportunity enabled by the Library Node abstraction. The ability to *nest* levels of decreasing abstraction make Library Nodes a versatile tool to achieve a number of tasks. An example is included in Figure 8: a linear algebra-aware frontend produces an SDFG containing a generic matrix multiplication operator Library Node (top); once expanded, the Library Node can delegate itself to a number of different linear algebra operations, depending on the dimensionality of the two operands; finally, the performance engineer can choose either a generic implementation, or one specialized for a specific backend. For example, they may choose an FPGAspecific streaming implementation specialized for Xilinx FPGAs (or a generic FPGA implementation).

In the following, we use two composite linear algebra kernels from the extended set of BLAS subprograms proposed by Blackford et al. [16] to demonstrate the multi-level design process using SDFGs and the DaCe framework.

3.1 High-Level Domain-Specific Frontends

To develop programs using the SDFG representation, programmers can use high-level frontends, rather than using the low-level graph API to create SDFGs directly. For example, the DaCe framework itself exposes a Python frontend supporting NumPy [17], and with BLAS extensions. Calling BLAS routines or using linear algebra operators on NumPy arrays will emit BLAS Library Nodes in the resulting SDFG, which can later be expanded to the desired implementation: e.g., direct function calls to MKL, cuBLAS, or OpenBLAS, or specialized SDFG subgraphs targeting a specific architecture.

Figure 9 shows AXPYDOT, a small composite BLAS kernel, summing two input vectors, then taking the dot product with the resulting vector and a third input vector. The



Fig. 9. Implementation of AXPYDOT using the standard Fig. 10. Generic SDFG Fig. 11. The SDFG from Figure 10 automatically trans-DaCe Python frontend and BLAS library calls. computing AXPYDOT.

formed for FPGA execution.

SDFG emitted by the frontend for this code is shown in Figure 10. The BLAS operators are instantiated as the axpy and dot Library Nodes, reading and writing from arrays. The two kernels exchange data through the array z, which will be first written by axpy and then read by dot, in sequence. Kernels that are composed of BLAS level 1 and 2 routines, such as AXPYDOT, are fully memory bound, but expose a promising opportunity for streaming computation by pipelining temporaries directly between subroutines [18] on the FPGA, which we will exploit in the following.

3.2 Mid-Level FPGA Transformations

SDFGs can be specifically engineered to target FPGAs, by writing them using the graph API. Alternatively, DSLs that target FPGAs, such as StencilFlow [5], can directly emit FPGA-specific graphs. Finally, existing SDFGs can be transformed from a generic implementation to an FPGA implementation using graph transformations. Any of these approaches will result in graphs that can be further optimized using general-purpose transformations available in the DaCe toolbox. This includes platform-agnostic transformations such as map tiling, inserting fast memory buffers, or removing redundant memory accesses [3]; and more FPGA-oriented transformations, which we describe here.

Transforming a Subgraph into an FPGA Kernel 3.2.1

If the input is a generic graph that has not yet been targeted to FPGAs, programmers can automatically offload a full SDFG or a specific subgraph for FPGA execution using the FPGATransformSDFG and FPGATransformState transformations, respectively, provided in the DaCe framework. These detect all DRAM accesses in the target graph or subgraph, then create additional pre- and post-states performing memory transfers between host and device. The memories accessed by the transformed subgraph are replaced with their FPGA equivalents.

Figure 11 shows the AXPYDOT example from Figure 10 after applying the FPGATransformSDFG transformation. Occurrences of the DRAM memories x, y, and w and replaced with corresponding FPGA memories fpga_x, fpga_y, and fpga_w in the kernel graph, the memories are copied to the FPGA before the kernel is executed in the state pre_axpy, and the output array result is copied back in the state post_axpy. This program can already be generated and compiled for both Xilinx and Intel boards.

3.2.2 Memory Access Extraction

When the memory access pattern of a certain computation is known, it is often beneficial to stream the data into the FPGA

processing elements. Creating streaming accessors has many benefits [2], including making use of burst-mode in memory controllers, tailored buffering for pipelined execution, broadcasting off-chip memory to multiple processing elements, or customizing caching mechanisms.

In DaCe, extracting a streaming pattern from an existing memory access to a streaming access is performed via the StreamingMemory transformation. The transformation processes the outgoing (or incoming) memlets of a certain data access node, finding all recurring access patterns of unique symbolic expressions. If the range consists of one scalar or vector element, the transformation can be applied. It then extracts the read (write) out of the computation by introducing another component that accesses the memory in the same order as the computation, and pushes it onto a stream (or pops computation outputs and stores results). The corresponding outgoing/incoming memlets are replaced by memlets that access the stream instead. If more than one PE uses the same memory access order, the transformation generates a single streaming component that connects one array node to multiple streams. In order to avoid deadlocks, the transformation also detects dependencies by computing reachability from the destinations/sources of the memlet paths (inherently given by the construction of the SDFG). If accesses are dependent, separate components are created, even for the same access pattern.

3.2.3 Pipeline Fusion

Following streaming the endpoints of a computation, we also consider streaming composition of consecutive computations. In unoptimized SDFGs, intermediate data is represented as data access nodes, pointing to off-chip memory by default. This round-trip is undesirable, and in certain computations can be completely avoided by fusing the two underlying pipelines into one.

The StreamingComposition transformation is similar in structure to memory access extraction, but checks for array nodes with in-degree and out-degree of one, which contain equivalent access orders that can be composed. To do so, the transformation traces the memlet path through map/pipeline scopes and nested SDFGs, canonicalizing the memlets' symbolic expressions by remapping symbol names to indices. If the ranges of the iteration spaces match exactly, and the symbolic expressions are equal, the result of the first computation can be streamed into the second. Similarly to StreamingMemory, we replace the memory access nodes and neighboring memlets with streams, converting global memory arrays into local streams.



Fig. 12. The AXPYDOT program after automatically extracting memory accesses into processing elements and streaming between operators. Streams are color-coded by name for pipeline visualization.

3.2.4 Putting it All Together

An example that uses all mid-level transformations can be seen in Figure 12. Manually applying FPGATransformSDFG, StreamingMemory on x, y, wand StreamingComposition on AXPYDOT yields fully pipelined execution. The same transformations can be applied automatically, in a scheme which we perform for our applications in Sections 4–6. However, the transformations must be attempted at a certain order in order to apply. First, the input SDFG must be transformed to FPGA-based computation (FPGATransformSDFG). Then, the data can be vectorized to the desired length (using Vectorization), which the Library Nodes use to control unrolling and accumulation factors upon expansion. After expanding the Library Nodes, the memory access patterns of each computation is exposed, and StreamingMemory and StreamingComposition can be applied. Lastly, the memory assignment to banks can be tweaked by inspecting the dataflow of the SDFG.

The resulting SDFG is capable of generating separate modules for efficiently reading/writing off-chip memory, using the stream construct as a FIFO queue to connect pipeline stages. In the rest of this section, we describe how the Library Nodes and can be further specialized depending to the target FPGA vendor.

3.3 Platform Specialization

While it is possible (and often sufficient) to implement operators with a generic SDFG subgraph, it can be desirable to specialize the implementation of an operation to a specific target. This can simply mean calling an external high performance implementation, such as cuBLAS, or it can mean a specialized native DaCe graph expansion. Because of the differences in capabilities between Intel and Xilinx architectures, such specializations prove useful in practice.

3.3.1 Floating Point Accumulation

For computations that need to perform accumulation, such as the DOT operator used in Figure 10, it is beneficial to specialize the computation based on whether the underlying architecture supports accumulation on the given data type.



Fig. 13. AXPYDOT expanded for Xilinx (left), using a partial sum and reduce phase, and for Intel (right), accumulating into a single register.

Intel Arria 10 and Stratix 10 architectures supports native 32-bit floating point accumulation, which allows a stream of floats to be directly summed into an output register. Contemporary Xilinx FPGAs, such as the Alveo U250, do not have native 32-bit floating point units, and cannot directly accumulate floating point numbers into a single register, as this results in a loop-carried dependency induced by the multiple-cycle latency of the addition operation. For 64-bit floating point, neither Xilinx nor Intel support accumulation, and both must address the issue of loopcarried dependencies.

To avoid the loop-carried dependency for DOT, we can perform accumulation interleaving [2] by summing the incoming data into a number of partial sums, stored in a buffer of a size larger than the latency of the addition operation. In Figure 13, the AXPY and DOT operators have both been expanded for Xilinx (left) and Intel (right). AXPY uses a generic implementation (identical to the CPU implementation), while DOT is implemented using specialized expansions depending on the target architecture. The Xilinxtargeted expansion uses a partial sum strategy to resolve the loop-carried dependency, using two unrolled maps. The first ("unroll") sums up all entries of the vector containing the product of contributions from x and y using a fully unrolled circuit (i.e., W-1 adders, where W is the vectorization width), resulting in a single element contribution. This contribution is added into the partial sum buffer, accessed with a cyclic index. The second unrolled map ("reduce") is performed after the main streaming phase, and sums up all values in the partial sums buffer into a single output, which is written to the output (again consuming W-1 adders. In a resource-constrained scenario, this could be reduced to a single adder without impacting the asymptotic runtime). The Intel specialization instead accumulates into a single register, saving the partial buffer and additional reduction.

3.3.2 Shift Registers

For Intel FPGA, we can use shift registers to achieve the sliding window-style buffering pattern required for stencil computations. This is a powerful abstraction that is desirable to exploit by the Intel FPGA backend, but does not (as of writing) exist in Vivado HLS. In Section 6.2, we show

 TABLE 1

 Performance of AXPYDOT on the Alveo U250 (Xilinx) FPGA.

Naïve HLS in DaCe	Streaming Transformations
$3.57\pm0.15~\mathrm{GB/s}$	$9.34\pm0.03~\mathrm{GB/s}$

how — with some additional effort — the shift register pattern can be imitated for stencil computations on Xilinx, by implementing a Xilinx-specific expansion using explicit buffers between each access point.

3.3.3 Specializing SDFGs vs. Hand-Written Code

Targeting low-level aspects of the HLS tools, such as floating point accumulation and shift registers, raises the question of whether there is even any benefit of using the Library Node expansion abstraction versus writing these specializations by hand. While embedding hand-written HLS code inside SDFGs is indeed supported natively by the framework, implementing the specializations as graph expansions come with a significant benefit of *malleability*, analyzability, and potential for reuse. Malleability, because the expanded subgraph can be further transformed by the DaCe framework: for example, to tile the internal maps, or to add/change sizes and constants, such as input size and vectorization width. Analyzability, because the graph view gives an overview of the computational structure, and allows inspecting data volumes on dataflow edges (e.g., to detect producer/consumer mismatches). Potential for reuse, because even specialized codes can potentially be reused between platforms or vendors - for example, using the partial sums implementation of DOT to reduce 64-bit floating point numbers on an Intel platform, without writing any additional code. Finally, staying within the SDFG representation lets the program benefit from the powerful code generating backends, which are continuously improved and updated to work seamlessly with the newest versions of the HLS toolflows.

4 CASE STUDY: LINEAR ALGEBRA

We have seen the multi-level design methodology applied to the AXPYDOT example throughout Section 3. We additionally show how this flow can reproduce the composite BLAS kernel GEMVER evaluated by FBLAS [18], using our multi-level SDFG design. In the following, we evaluate kernels on a Xilinx Alveo U250 accelerator board, compiled using Vitis 2020.2 for the xilinx_u250_xdma_201830_2 shell, and using the Xilinx Runtime (XRT) version 2.5.309. Results were measured 10 times, the median and 95% nonparametric confidence intervals are listed as errors.

4.1 AXPYDOT Evaluation

The result of the AXPY/DOT BLAS operation composition are listed in Table 1. As the program is bandwidth-bound, we list the attained bandwidth of AXPYDOT when run with an input buffer of 800 MiB (209,715,200 elements). The table shows that the streaming transformations, which are applied automatically, are able to stream all memory accesses, and fuse the AXPY and DOT pipelines. This is a promising result for FPGA programs in general, since the transformations detected the access patterns directly and did not employ application-specific knowledge. As for generated code length, the naïve version generates one module (processing element) and 139 lines of code, whereas the streamed version generates 5 separate modules and 207 lines of code. All in all, this contributes to a $2.6 \times$ speedup of the streamed version over the original input graph.

4.2 GEMVER Optimization and Evaluation

The GEMVER application is a composition of several BLAS operations used in solving systems of equations. Its SDFG is shown in Figure 14. Specifically, GEMVER performs two rank-1 updates (GER), a transposed matrix-vector multiplication (GEMV^T), and another matrix-vector multiplication (GEMV), both using different access patterns.

There are several approaches

that can be taken to layout this complex composition [18], stemming from the difference in access patterns between the two GEMVs, and the inability to directly stream the result of the two rank-1 updates into both matrix-vector multiplications at the same time. Thus, malleability plays a key role for GEMVER optimization.

In DaCe, Library Nodes can be expanded in multiple ways, depending on the target platform and the parameters of the node (i.e., whether an array is transposed). For GER, vectorized expansion would only apply if the

matrix and the first or second vector are vectorized, but not both. In order to compose the operators GEMVER, the performance engineer must match the tiling schemes between them: for the transposed GEMV, a scheme that streams in tiles by columns matches the output of GER, while the second GEMV can use a row-major scheme. Once the access patterns match, the array z can automatically be streamed by the mid-level transformations.

Data movement for the two consumers of C can be significantly reduced by *both* pipelining *and* storing it in offchip memory for later use. As mentioned in Section 3.2.3, streaming composition only works if there are no other uses of the array. However, the performance engineer can manually replicate C (interactively or programmatically) following Library Node expansion, creating the possibility to apply pipeline fusion once more, which would remove one of the replicas of C in favor of a stream.

TABLE 2 Performance of GEMVER on the Alveo U250 (Xilinx) FPGA.

Version	Runtime [s]	Off-Chip Volume
Naïve SDFG	$1.10{\pm}0.029$	6.0 GiB (—)
Manual memory banks	1.52 ± 0.000	6.0 GiB (1×)
Streaming composition	$0.88 {\pm} 0.004$	4.0 GiB (1.5×)
Manual composition	$0.74 {\pm} 0.005$	3.0 GiB (2×)



Fig. 14. GEMVER SDFG.



Fig. 15. LeNet-5: PyTorch module (left) and corresponding SDFG (right, array nodes are hidden for brevity).

Results of GEMVER with N=16,384 are shown in Table 2. As our baseline already uses vector width of 16 elements, tiled computation, and the matching Library Node implementations, the differences in performance are not substantial. The example shows the importance of streaming composition: when memory banks are manually chosen to maximize bandwidth the program is slower. Performance only improves when the accesses are streamed and composed. Using the manual replication of C yields an additional improvement in performance: a $2\times$ reduction in memory movement and $1.49\times$ reduction in overall runtime.

5 CASE STUDY: MACHINE LEARNING

We present the multi-level design methodology for FPGAs in the context of a machine learning application, utilizing the ONNX frontend of DaCe.

5.1 ONNX Domain-Specific Frontend

DaCeML¹ is a data-centric machine learning framework [19] based on DaCe. The framework exposes the operators of the Open Neural Network eXchange (ONNX) IR as Library Nodes, and provides a collection of implementations for each operator, specialized for CPUs, GPUs, and FPGAs. As a case study, we present the process of lowering a PyTorch [20] model, implementing Deep Learning inference with the LeNet-5 convolutional model [21], to an SDFG that can be executed on either FPGA vendors. Using DaCeML, we generate ONNX files and SDFGs from PyTorch neural network modules with a single-line Python decorator (Figure 15).

DaCeML includes domain-specific transformations that are difficult or impossible to implement at the lower levels of the SDFG IR. To optimize for FPGAs, we develop and employ the InputToConstant transformation. DaCeML SDFGs typically receive both their inputs and parameters as arrays. For inference, those parameters can be fixed in hardware. This FPGA specific transformation converts a parameter array of the model to a compile-time constant. The transformation first verifies that the parameter array is never written to, then removes the input by traversing the edges, removing all corresponding memlets and access nodes. This domain-specific transformation requires knowl-edge of the parameter values, which are obtained from PyTorch.

5.2 Lowering to FPGA

After applying the FPGATransformSDFG (see Section 3.2.1) for offloading the execution of the SDFG on FPGA, each of the Library Node is expanded to a nested SDFG optimized for spatial architectures. These expansions employ a range of optimizations. Convolutions ("Conv") are implemented by using the image to column (*im2col*) approach [22]. Therefore, the convolution and GEMM expansions rely heavily on the systolic matrix multiplication shown in Section 2.6. The activation function ("ReLU") is an element-wise operation, and can be expressed by nested maps. The MaxPool implementation uses a sliding window, implemented using shift registers. All the network operators operate on single precision floating point, and can accept data either from off-chip memory or streamed in from on-chip memory. The latter opens up to the possibility to streaming among operators, such as between convolution, activation and sub-sampling (blue dashed boxes in Figure 15), allowing a reduction of I/O through pipelined execution of sections of the graph. Figure 16 show the program SDFG obtained by applying the general StreamingComposition transformation (described in Section 3.2.3 to automatically achieve this).

5.3 Evaluation

For the evaluation, we target the BittWare 520N PCIe attached board, with an Intel GX 2800 Stratix 10 processor, equipped with 4× DDR4 memory banks. The OpenCL code generated from DaCe is compiled with version 20.3 of the Intel FPGA OpenCL SDK and 19.4 of the Quartus compiler. We consider a batch size of 1,000 for our experiments. As reference, we use the inference time of running the input PyTorch model on CPU. We target a 36C/72T Intel Xeon Gold 6154 and PyTorch v1.6.0. Inference is measured at 56.2 ± 0.0433 ms and 14.4 ± 0.0447 ms for 1 core and 36 cores, respectively. We evaluate the different transformations applied to the SDFG in Table 3, showing a speedup



Fig. 16. The LeNet program after transforming to stream between operators. Streams are color-coded by name for pipeline visualization. Library Node expansions are collapsed for readability.

of 1.87x over the single core CPU implementation. The InputToConstant transformation yields a speedup over the original SDFG graph of $3.2 \times$ and a reduction of data movement. By applying the StreamingComposition to replace intermediate memories with streams, we further reduce the accessed data volume, and the speedup is increased to $8.8 \times$. This shows how combining domain-specific and general transformations can yield significant benefits, and how general transformations are reused across domains (see examples for applying the transformation to linear algebra in Sections 3.2.3 and 4).

 TABLE 3

 Performance of LeNet-5 on the Stratix 10 (Intel) FPGA with inference batch size 1000.

Version	Runtime [ms]	Off-Chip Volume
Naïve SDFG	265.8±3.502	0.28 GiB (—)
Input to constant	81.3±1.570	0.22 GiB (1.2×)
Streaming composition	30.1±0.703	0.16 GiB (1.7×)

6 CASE STUDY: STENCILFLOW

StencilFlow [5] is a domain-specific framework built on top of the DaCe framework, utilizing the full multi-level design to emit fully pipelined and deadlock free stencil architectures for complex input programs. In the following, we describe how each of the levels of optimizations described in Section 3 are applied in the context of StencilFlow, then further exploit the specialization capabilities of Library Nodes to extend the framework to target Xilinx FPGAs from the same input programs, even without access to shift registers.

6.1 Stencil Language and Transformations

StencilFlow defines a specialized stencil DSL, expressed in a JSON input format, allowing input programs with heterogeneous operators, reading from different input data containers, and with dependencies between them. An analysis tool parses the operators and maps the dependencies between them, computes the buffers required to fully pipeline each operator, then computes the delays between operators to insert *delay buffers* between them to prevent deadlocks that can otherwise be induced by fork/joins in the dependency graph. An simple example program, implementing two iterations of the diffusion 2D stencil, is shown in Figure 17.

6.2 Intel and Xilinx Stencil Specialization

StencilFlow presents results for an Intel Stratix 10 FPGA. The Intel OpenCL compiler is suitable to target stencil computations, due to the shift register abstraction allowing easy instantiation of more complex cyclic buffering patterns. For this work, we show how the StencilFlow stack can be extended to target Xilinx FPGAs instead, simply by providing a new stencil node expansion to the stencil Library Node, with no further changes to the surrounding infrastructure.

Intel OpenCL's shift register abstraction leaves buffer management for stencil programs up to the compiler. The stencil node expansion subgraph for a 2D 4-point stencil with 4-way vectorization is shown in the *left* graph in Figure 18. At every iteration, the shift register buffer is "shifted" forward by an amount equivalent to the vector length (blue box, top). Then, a new input vector of data from the wavefront is written to the front of the shift register buffer (red box, middle), resulting in the shift register containing all data necessary to perform the stencil computation. Finally, the computation is performed in an unrolled map over the vector width (black box, bottom), loading the 4×4 appropriate entries from the shift register (four accesses in each iteration of the size-4 unrolled map, although some will overlap in practice), and the result is written to the output stream.

Xilinx does not expose a scalable shift register abstraction. Instead, the buffers between each stencil access must be deduced, instantiated, and accessed explicitly, which is challenging due to vectorization resulting in non-aligned accesses into the vector strides. The *right* graph in Figure 18 shows a Xilinx expansion, achieving the exact same computation as the left graph, but without the aid of shift registers. The 4-point stencil has four access points, which with 4-way vectorization requires 14 unique offset. Offsets are computed as the distance from the "earliest access" relative to the iteration pattern. These offsets are translated into major indices (which buffer is accessed, according to the vector stride) and minor indices (indices into each accessed vector). The major indices become the "access points" into the vectorized buffers, resulting in 4 buffers for this example. At each iteration, the buffers are read at a cyclic index along with the value from the wavefront. Because the kernel is vectorized, a full vector must be read from each buffer first, after which the individual scalar elements can be extracted by the kernel (implemented by the 14 dataflow edges between the buffers and the computation). Finally, each buffer is updated with the value from the *following* access point, and the front access point (i.e., highest flattened index) is updated from the wavefront.

While it required a non-trivial effort to implement a Library Node expansion for Xilinx that implements a stencil buffering pattern, the payoff is significant: the StencilFlow frontend and analysis framework, as well as the remaining SDFG (the memory readers/writers, and the interconnection between stencils) remains unchanged, and we can now

```
1{"dimensions": [4096, 4096], "vectorization": 8,
2 "outputs": ["d"], "inputs": {
   "a": {"data_type": "float32", "input_dims": ["j","k"]},
"c0": {"data_type": "float32", "input_dims": []},
3
5
   "c4": {"data_type": "float32", "input_dims": []}},
6
  "program": {
7
   "b" • {
8
    "data_type": "float32",
9
    "boundary": {"a": {"type": "constant", "value": 0}},
10
    "computation": "b = c0*a[j,k] + c1*a[j-1,k] +
11
     "d": {
12
    "data_type": "float32",
13
    "boundary": {"b": {"type": "constant", "value": 0}},
14
    "computation": "c = c0*b[j,k] + c1*b[j-1,k]
15
    16 \} \} \}
```

Fig. 17. JSON-based StencilFlow program description.



Fig. 18. Intel (left) and Xilinx (right) stencil node expansions.

directly compile it for Xilinx instead.

6.3 Evaluation

We evaluate the Xilinx stencil expansion on the Alveo U250 accelerator board from kernels produced by the StencilFlow framework. While the Alveo U250 is a flagship UltraScale+ device, it suffers relative to Intel chips in not having native floating point units, and from its chiplet-based design, which limits connectivity between parts of the device. Still, the large amount of general purpose logic available allows instantiating large stencil programs.

We benchmark the vectorized Jacobi 3D, diffusion 2D, and diffusion 3D stencil programs on the Xilinx U250 board with 32-bit floating point types, using long and narrow $2^{17} \times 4096$ and $2^{15} \times 128 \times 128$ domains for 2D and 3D, respectively, to emulate time tiled stencils. We plot the results along with benchmarks on a Stratix 10 board evaluated by StencilFlow in Figure 19. The plot includes benchmarks both with and without accessing DRAM, as the Alveo board was observed to deliver significantly less than the expected memory bandwidth, despite the burst-friendly access pattern. The U250 yields up to $373 \,\mathrm{GOp/s}$ ($300 \,\mathrm{GOp/s}$) without (and with) memory, but falls short of the much larger floating point capabilities of the Stratix 10. The Xilinx stencil performance yields a $3.2 \times$ improvement over the stencil architecture evaluated in the original work on DaCe [3] (which used a KU115 board), and a $2.8 \times$ over the SODA [23] framework (ADM-PCIE-KU3 board). Although both these works use previous generation FPGAs, the domain-specific StencilFlow stack is responsible for a significant benefit.

Finally, we include preliminary results on the U250 for executing *horizontal diffusion*, a large real-life weather simulation stencil program [5], which contains a multitude of heterogeneous stencil computations and buffering patterns with complex dependencies between them. The program is run in a fully pipeline parallel fashion, where every stencil operation is performed in parallel in a streaming fashion, requiring careful buffering on channel connections to avoid



Fig. 19. Performance of StencilFlow across Intel and Xilinx platforms. *From StencilFlow [5] paper. [†]Preliminary results: not yet validated.

deadlocks. Because the buffering is done at the SDFG-level, this can be directly compiled for the Xilinx architecture along with the stencil expansion. The program is applied on a $128 \times 128 \times 80$ grid using 32-bit floating point types. However, both memory and compute utilization is poor, pending further investigation into why the U250's resources are not sufficiently exploited.

7 RELATED WORK

HeteroCL [24] proposes a programming model based on a high-level DSL extended from TVM [25], allowing the user to express programs through a set of computational primitives, some of which are specialized to specific backends: a stencil backend [23] and a tensor multiplication backend [26] framework). Optimizations can be done by the programmer by setting parameters of the employed patterns, following the approach of Halide [27]. The SDFGbased method presented here takes a somewhat opposite approach: rather than exposing existing backends through a unified DSL, we unify the full optimization flow and code generation process within a single representation, which can be targeted and customized by any number of internal or external frontends and DSLs. The benefit of HeteroCL is a more "push-button" approach: by constraining the input to certain patterns supported by the specialized backends, less intervention is needed by the engineer to achieve fast programs. In PyLog [28], the programmer writes python code, with the framework being responsible of applying optimization passes and generating C++ annotated code targeting Xilinx devices (through PYNQ [29]). Compared with both HeteroCL and PyLog, the benefit of SDFGs and the DaCe framework is malleability: every step of the design and optimization process is provided as a tool to the programmer, and every component can be customized, extended, or even hacked with low-level code. Domainspecific constructs and transformations exist within the same space as general constructs and transformations, and can be applied interchangeably. This consequently yields out-of-the-box cross-platform compatibility, as any frontend, DSL, or application based on SDFGs has access to either backend, even if is desirable to tweak details to a specific platform.

Spatial [30] is a language for programming spatial systems through a combination of parallel patterns that constrain the input, and an abstraction of the underlying hardware that these patterns can be efficiently mapped to. The constrained input space is exploited to perform design-space exploration of the hardware mapping, focusing on

automated performance tuning of the design. In contrast, the DaCe framework focuses on interactive design based on the data-centric model, which requires more guidance by the performance engineer, but shares its knowledge and toolbox between different systems, and is built to be extensible at every level of the stack.

The multi-level approach taken here is similar to that of MLIR [15], which introduces multi-level design to compiler IR, where *progressive lowering* allows going from highlevel domain-specific constructs [31], through any number of intermediate formats, down to traditional compiler IR such as LLVM. MLIR targets an automated compiler-based approach, while SDFGs are an interactive format that is progressively transformed based on the provided complete view of the program's data movement.

Kenter et al. [32] propose a macro-based approach to writing HLS programs that are portable between Xilinx and Intel OpenCL flows. DaCe lifts cross-compatibility into a full code generator, allowing a superset of opportunities to emit highly targeted and structured code for either vendor.

8 CONCLUSION

We proposed leveraging the Stateful DataFlow multiGraphs (SDFGs) representation as a programming model for spatial computing systems. SDFGs express programs by the dataflow and control flow, exposing all data movement to the performance engineer. We showed how the representation is code generated to efficient architectures for either FPGA vendor, and how Library Nodes allow embedding abstract domain-specific behavior into the representation, allowing SDFGs to be harnessed using a multi-level design methodology. We showed how programs from linear algebra, machine learning, and stencil domains can be written using expressive high-level frontends, and optimized with both domain-specific transformations, and general purpose transformations reused across domains, then specialized further to exploit platform features. The multi-level methodology promotes the invention of novel transformations and abstractions that further increase productivity and performance not only for the application at hand, but for any future SDFG amenable to the same optimization strategy.

ACKNOWLEDGEMENTS



This project received funding from the European Research Council (ERC) grant PSAP, grant agreement No. 101002047, and the European Union's Horizon Europe programme DEEP-

SEA, grant agreement No. 955606. T.B.N. is supported by the Swiss National Science Foundation (Ambizione Project #185778). The project was also sponsored by the Paderborn University, under the DaceML-FPGA project.

REFERENCES

 J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 30, no. 4, 2011.

- [2] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler, "Transformations of high-level synthesis codes for high-performance computing," *IEEE Transactions on Parallel and Distributed Systems* (*TPDS*), vol. 32, pp. 1014–1029, May 2021.
- [3] T. Ben-Nun, J. de Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler, "Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'19)*, 2019.
- [4] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham et al., "Trends in Data Locality Abstractions for HPC Systems," *IEEE Transactions* on Parallel and Distributed Systems (TPDS), vol. 28, no. 10, Oct. 2017.
- [5] J. de Fine Licht, A. Kuster, T. De Matteis, T. Ben-Nun, D. Hofer, and T. Hoefler, "StencilFlow: Mapping large stencil programs to distributed spatial computing systems," *Proceedings of the 19th* ACM/IEEE International Symposium on Code Generation and Optimization (CGO'21), 2021.
- [6] A. N. Ziogas, T. Ben-Nun, G. I. Fernández, T. Schneider, M. Luisier, and T. Hoefler, "A data-centric approach to extreme-scale ab initio dissipative quantum transport simulations," in *Proceedings of the International Conference for High Performance Computing*, Networking, Storage and Analysis, 2019, pp. 1–13.
- [7] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, 2008.
- [8] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, "AutoPilot: A platform-based ESL synthesis system," in *High-Level Synthesis*. Springer, 2008, pp. 99–112.
- [9] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, and M. o. Kinsner, "From OpenCL to high-performance hardware on FPGAs," in 22nd International Conference on Field Programmable Logic and Applications (FPL'12). IEEE, 2012, pp. 531–534.
- [10] C.-J. Johnsen, T. D. Matteis, T. Ben-Nun, J. de Fine Licht, and T. Hoefler, "Temporal Vectorization: A Compiler Approach to Automatic Multi-Pumping," in 2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD), Oct. 2022.
- [11] Xilinx. QDMA Subsystem for PCI Express v4.0, PG302 (v4.0) July 1, 2020. Accessed: Dec 2022.
- [12] J. de Fine Licht and T. Hoefler, "hlslib: Software engineering for hardware design," arXiv:1910.04436, 2019.
- [13] H. T. Kung and C. E. Leiserson, "Systolic arrays for (VLSI)," Carnegie-Mellon University, Tech. Rep., April 1978.
- [14] J. de Fine Licht, G. Kwasniewski, and T. Hoefler, "Flexible communication avoiding matrix multiplication on FPGA with high-level synthesis," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'20)*, 2020.
- [15] C. Lattner, J. Pienaar, M. Amini, U. Bondhugula, R. Riddle et al., "MLIR: A compiler infrastructure for the end of Moore's law," arXiv:2002.11054, 2020.
- [16] S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling et al., "An updated set of basic linear algebra subprograms (BLAS)," ACM Transactions of Mathematical Software, vol. 28, no. 2, pp. 135– 151, June 2002.
- [17] A. N. Ziogas, T. Schneider, T. Ben-Nun, A. Calotoiu, T. De Matteis, J. de Fine Licht, L. Lavarini, and T. Hoefler, "Productivity, portability, performance: Data-centric python," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis,* ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021.
- [18] T. De Matteis, J. de Fine Licht, and T. Hoefler, "FBLAS: Streaming linear algebra kernels on FPGA," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (SC'19), 2019.
- [19] O. Rausch, T. Ben-Nun, N. Dryden, A. Ivanov, S. Li, and T. Hoefler, "DaCeML: A data-centric optimization framework for machine learning," in *Proceedings of the 36th ACM International Conference* on Supercomputing, ser. ICS '22, 2022.
- [20] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan et al., "PyTorch: An imperative style, high-performance deep learning library," in Advances in Neural Information Processing Systems, vol. 32, 2019, pp. 8026–8037.
- [21] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard et al., "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, 1989.
- [22] K. Chellapilla, S. Puri, and P. Simard, "High Performance Convolutional Neural Networks for Document Processing," in *Tenth*

International Workshop on Frontiers in Handwriting Recognition, G. Lorette, Ed. Suvisoft, Oct. 2006.

- [23] Y. Chi, J. Cong, P. Wei, and P. Zhou, "SODA: stencil with optimized dataflow architecture," in Proceedings of the 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'18), 2018.
- [24] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu et al., "HeteroCL: A multi-paradigm programming infrastructure for softwaredefined reconfigurable computing," in Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'19), 2019, pp. 242-251.
- [25] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan et al., "TVM: endto-end optimization stack for deep learning," arXiv:1802.04799, 2018.
- [26] J. Cong and J. Wang, "PolySA: polyhedral-based systolic array auto-compilation," in Proceedings of the 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'18), 2018.
- [27] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson et al., "Programming heterogeneous systems from an image processing DSL," ACM Transactions on Architecture and Code Optimization (TACO), vol. 14, no. 3, p. 26, 2017.
- [28] S. Huang, K. Wu, H. Jeong, C. Wang, D. Chen, and W.-M. Hwu, "Pylog: An algorithm-centric python-based fpga programming and synthesis flow," IEEE Transactions on Computers, vol. 70, no. 12, pp. 2015–2028, 2021. "Xilinx PYNQ," http://www.pynq.io/, accessed: Dec 2022.
- [29]
- [30] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis et al., "Spatial: A language and compiler for application accelerators," in Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18), 2018, pp. 296–311. [31] T. Gysi, C. Müller, O. Zinenko, S. Herhut, E. Davis *et al.*, "Domain-
- specific multi-level IR rewriting for GPU," arXiv:2005.13014, 2020.
- [32] T. Kenter, J. Förstner, and C. Plessl, "Flexible FPGA design for FDTD using OpenCL," in *Proceedings of the 27th International* Conference on Field Programmable Logic and Applications, 2017.