OverGen: Improving FPGA Usability through Domain-specific Overlay Generation

Sihao Liu[§*], Jian Weng[§*], Dylan Kupsh[*], Atefeh Sohrabizadeh[*], Zhengrong Wang[*], Licheng Guo[*], Jiuyang Liu[†]
Maxim Zhulin[*], Rishabh Mani[*], Lucheng Zhang[†], Jason Cong[*], Tony Nowatzki[*]
[*]*University of California, Los Angeles* [†]*Institute of Software, Chinese Academy of Sciences*
{*sihao,jian.weng,dkupsh,atefehsz,seanzw,lcguo,cong,tjn*}*@cs.ucla.edu*

***Abstract*—FPGAs have been proven to be powerful computational accelerators across many types of workloads. The mainstream programming approach is high level synthesis (HLS), which maps high-level languages (e.g. C + #pragmas) to hardware. Unfortunately, HLS leaves a significant programmability gap in terms of reconfigurability, customization and versatility: Although HLS compilation is fast, the downstream physical design takes hours to days; FPGA reconfiguration time limits the time-multiplexing ability of hardware, and tools do not reason about cross-workload flexibility. Overlay architectures mitigate the above by mapping a programmable design (e.g. CPU, GPU, etc.) on top of FPGAs. However, the abstraction gap between overlay and FPGA leads to low efficiency/utilization.**

Our essential idea is to develop a hardware generation framework targeting a highly-customizable overlay, so that the abstraction gap can be lowered by tuning the design instance to applications of interest. We leverage and extend prior work on customizable spatial architectures, SoC generation, accelerator compilers, and design space explorers to create an end-to-end FPGA acceleration system. Our novel techniques address inefficient networks between on-chip memories and processing elements, as well as improving DSE by reducing the amount of recompilation required.

Our framework, OverGen, is highly competitive with fixed-function HLS-based designs, even though the generated designs are programmable with fast reconfiguration. We compared to a state-of-the-art DSE-based HLS framework, AutoDSE. Without kernel-tuning for AutoDSE, OverGen gets 1.2× geomean performance, and even with manual kernel-tuning for the baseline, OverGen still gets 0.55× geomean performance – all while providing runtime flexibility across workloads.

***Keywords*-Reconfigurable architectures; Domain-specific Accelerators; FPGA; CGRA; Design Automation;**

I. Introduction

FPGAs have proven to be highly performant and flexible hardware accelerators for important data-processing workloads (e.g. [1–17]), and have garnered significant traction in industry (e.g. [18–20]). Unfortunately, FPGAs pose significant challenges for programmer productivity. With RTL programming at the extreme end of complexity/low-productivity, the pragmatic options are high-level synthesis (HLS) and overlays, described next.

In HLS, a high-level language code (e.g. C with #pragmas) is lowered to a hardware state machine, and then passed as RTL to a traditional FPGA synthesis flow. Pragmas specify hints about the optimal hardware structure (e.g. unroll factor, initiation interval), and state-of-the-art frameworks

§ Sihao Liu and Jian Weng are co-first authors.

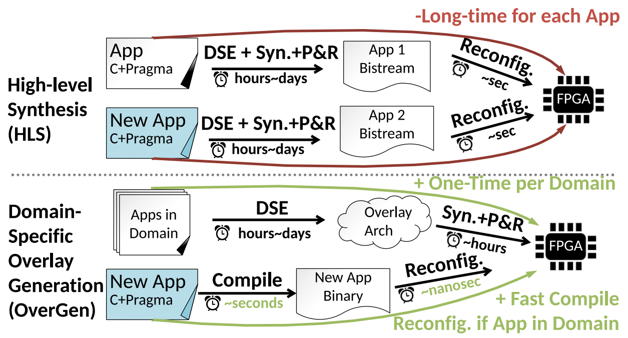


Figure 1: Overlay Generation Compared to HLS

like AutoDSE [21] explore these parameters on behalf of the programmer. While highly effective, HLS limits programmer productivity through high compilation/synthesis times. Also, multiplexing between applications by reflashing the FPGA bitstream takes significant time, taking more than a second to reconfigure modern FPGAs [10,22]. Moreover, HLS designs are specific to the input application: if any flexibility across applications is required, it must be programmed-in explicitly.

Alternatively, FPGA overlays map a coarser grain architecture (e.g. CPUs [23–26], GPUs [27–30], CGRAs [31–34]) on top of the FPGA's fine grain abstractions. While overlays reduce compilation/synthesis time and are more general, they experience quite high overheads due to the abstraction gap between general purpose architectures and the low-level fine-grain abstractions exposed by FPGAs. Overlays can be customized with domain-specific extensions [19,35], but this approach is highly time-consuming.

Vision and Requirements: Our vision is to use an HLS-like approach where the generated hardware is tuned to input applications, but which targets a highly-flexible overlay architecture instead of a fixed-function pipeline. Figure 1 gives the basic idea, where a set of applications are fed to a design-space exploration (DSE) step to determine the ISA and resource provisioning in the overlay, and compiling a new application (and reconfiguring) is extremely fast. Ideally, small application changes would not require FPGA synthesis.

We envision four requirements for overlay generation to be successful: 1. the overlay design space must include both system parameters and a broad accelerator design space, 2. it must balance generality versus specialization, depending on the degree of diversity in input applications, 3. the memory system itself should be highly specializable to the application, and 4. it has to get competitive performance with traditional HLS within a reasonable DSE time frame.

Approach: For the first two requirements, we leverage prior work on flexible multicore system generators (e.g. [36,37]) and spatial architecture synthesis (e.g. [38–46]). Multi-core system generators enable simple scaling in terms of cores, cache and network [36]. Spatial architecture[1] synthesis can provide accelerators for each core that are tuned to one or more applications. Spatial architectures provide a broad design space from fixed custom datapath accelerators to systolic arrays and vector architectures to coarse grain reconfigurable architectures (CGRAs). This flexibility comes from the graph-based representation of spatial architectures (nodes are PEs, switches, memory units, etc.).

To enable a highly application-specialized memory-system (requirement 3), our primary insight is that data-reuse structures (e.g. DMA engines, scratchpads) must be incorporated into the spatial architecture design space – i.e. enabling a custom topology connecting reuse structures to compute structures. For the DSE to make good decisions, this requires the compiler to analyze and expose data-reuse analysis to the spatial-scheduler intermediate representation. We refer to this technique as spatial-memory exploration.

Finally, for requirement 4, we notice that significant time is spent on recompiling workloads as the hardware definition changes. We develop novel techniques for modifying the hardware while preserving the validity of previous compilations. We call these schedule-preserving transformations.

Implementation and Implications: Our implementation is called OverGen, which integrates two open-source frameworks, the DSAGEN [38] spatial architecture generation framework and the ChipYard [36] SoC generator, and extends these with support for FPGA resource modeling at the system level, novel hardware design space extensions, and novel algorithms for DSE-time reduction.

While much of this work is about the integration of previous ideas and existing frameworks (with some novel extensions), the results are profound: Our evaluation suggests that domain-specific spatial overlays, and the OverGen framework specifically, have the potential to challenge HLS as the defacto FPGA design methodology. Our approach preserves a programmer-friendly interface with short compilation and reconfiguration times, and has competitive performance across many domains compared to the state-of-the-art HLS framework AutoDSE [21]. Across workload suites of DSP, Machsuite, and Vitis Vision, OverGen achieves geomean speedups of 1.21×, 1.13×, 1.25× speedups over baseline AutoDSE without kernel tuning, and it still reaches comparable performance, 0.71×, 0.37×, 0.65× respectively, with manual kernel tuning for AutoDSE. Our approach also enables overlays that support single or multiple workloads by *automatically* reasoning about the cross-workload flexibility.

[1]Spatial architectures are those that expose low-level aspects of hardware in their ISA, like resource assignment and scheduling of the operand network.

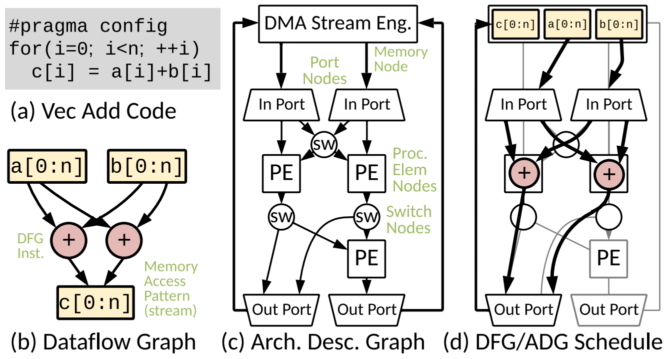

```
#pragma config
for(i=0; i<n; ++i)
  c[i] = a[i]+b[i]
```

Figure 2: Decoupled-Spatial Example of Vector Addition

Specifically, our contributions are:

- A full-stack domain-specific overlay generation framework verified on FPGA[2].
- Modeling techniques to codesign system parameters and accelerator design while balancing FPGA resources.
- Novel optimizations for integrating data-reuse into the spatial architecture DSE and reducing DSE design time.
- Evaluation demonstrating competitiveness against HLS and cutting edge DSE-for-HLS [21].

Paper Organization: Section II gives background on our decoupled-spatial accelerator design space. Section III overviews the approach and benefits. Section IV describes the spatial memory optimization. Section V covers the details of overlay design. Sections VI, VII, and VIII cover implementation, methodology and evaluation, and we conclude after discussing related work in Section IX.

II. Background: Spatial Architecture Synthesis

Our approach extends prior work on spatial architecture synthesis to create each overlay tile. This section first gives background on the accelerator execution model and then elaborates on design representation, the compilation and DSE techniques – all are enhanced in this work. We finish this section by describing the intellectual and practical limitations of existing spatial architecture synthesis techniques.

A. *Decoupled-Spatial Execution*

Execution Model: The accelerator execution model we use in this work is decoupled-spatial [47–55]. In this model, compute and memory accesses are expressed in a dataflow graph (DFG), and streams define coarse grain patterns of memory access, value generation, or communication. Streams and instructions execute when they receive all inputs required for one instance of their computation, as in ordered-dataflow [56]. An example transformation from source to dataflow graph (DFG) is shown in Figure 2(a) and 2(b); note the loop is unrolled by two iterations.

[2]Open-source repository: https://github.com/PolyArch/dsa-framework

Spatial Hardware and Mapping: Spatial architectures expose underlying hardware details to the ISA, like the capabilities and connectivity of hardware elements. Figure 2(c) shows an example represented as an architecture description graph (ADG). The ADG is composed of primitives like processing elements (PEs), switches for routing operands, DMA for generating memory addresses and requests, and ports for synchronizing between memory and compute. The compiler is responsible for mapping instructions, streams, and communication onto appropriate hardware units (e.g. PEs, stream engines, and switches); see the example in Figure 2(d).

Spatial Design Space: The modular nature of spatial hardware and its representation as an ADG, as in Figure 2(c), lead to a wide design space. The parameters of each component form a rich space which enables tradeoffs among performance, flexibility, and hardware cost. The topology is also flexible, enabling designs from app-specific datapaths, to vector architectures, mesh-CGRAs and much in between.

B. Spatial Compilers and Pragma Hints

A compiler that bridges high-level programming language to the decoupled-spatial execution is required to improve the programming productivity. OverGen leverages and extends the DSAGEN C+pragma compiler [38,57]. For context, two pragma extensions hint transformation decisions:

```
#pragma dsa config
{
  #pragma dsa decouple
  for (i = 0; i < n; ++i)
    for (j = 0; j < n; ++j)
      c[i] = a[i+j] * b[j];
}
```

Listing 1: An FIR example annotated with pragmas

`#pragma dsa config` This pragma annotates the scope of decoupled-spatial specialization. All the offloaded code regions within the compound body will be concurrent on the spatial configuration.
`#pragma dsa decouple` This pragma indicates all the memory accesses under the annotated loop level are alias-free if they are accessed by different pointers.

Generic Transformation: The instructions within the innermost loop will be sliced into two subsets, computational and memory access [58]. All the instructions that transitively depend on the load/store instructions are considered address generation, and the remaining ones are computational [57]. The computational instructions will be represented in a dataflow graph and fed to a spatial scheduler; and the memory instructions will later be fed to the memory analyzer.

Idiomatic and Modular Transformation: The compiler's role is to transform the program according to the capability of a particular design instance. If a certain transformation demands a specific hardware feature that is unavailable, a fallback transformation will be applied to guarantee the success of compilation [57]. After the analysis and transformation, all mapped instructions to the decoupled-spatial execution will be replaced by the accelerator ISA.

C. Automated Spatial Accelerator Synthesis

Our work leverages the spatial accelerator synthesis algorithm proposed in prior work [38], where the goal is to find the best single-core accelerator for a set of input workloads. The algorithm essentially performs graph-based simulated annealing on the ADG, using entirely random modifications and an evaluation function based on a simple performance and area model. To make DSE fast, the algorithm can avoid recompiling a kernel if the random hardware changes do not affect that kernel ("schedule repair").

Limitations of prior work: We address three key limitations of prior spatial-accelerator synthesis works [38–43]:

- *Datapath Limited:* Prior work performs spatial synthesis on the datapath of a single-core only, avoiding specializing memories within a core (e.g. scratchpads and their topology), and ignoring the shared memory system.
- *Reuse Ignored:* Prior systems ignore data-reuse as a first-order design constraint. This information is required to make good decisions about how to provision memory and network bandwidths, and it must be captured and made available by the compiler.
- *ASIC Focused:* Prior works focus on developing spatial architectures for ASICs. FPGAs have new challenges for optimizing across multiple resources (BRAMs, LUTs, DSPs, etc.), and present a compelling use case.

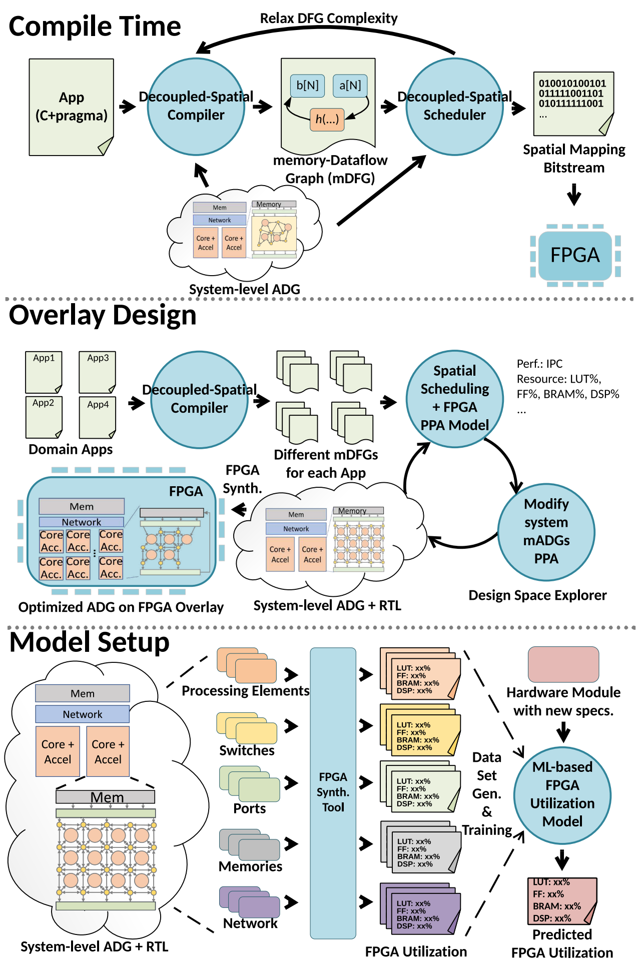
These limitations prevent prior systems from reasoning about critical design tradeoffs like core-count vs vector-width, scratchpad vs cache size, in-core reuse vs shared bandwidth. To address these limitations, OverGen extends the design space beyond a single core while considering FPGA resources (Section V) and adds reuse and memory access structures into the spatial/graph-level DSE (Section IV) — overall creating a full-stack overlay generation framework.

III. OverGen Overview & Tradeoffs

Here we discuss how OverGen spans compilation, design space exploration, and resource modeling, and then overview the design space and key tradeoffs.

A. Overview

Compilation: Figure 3 shows the overview of OverGen. We begin with the compilation flow, which takes the system-level ADG (sysADG) as input. The sysADG defines the spatial accelerator and system design spec, and is created during overlay generation (described later). The programming interface of OverGen is multithreaded C with aforementioned pragmas (details in Section VI-E). The LLVM-based compiler will attempt to create the highest-performance dataflow graph for the spatial accelerator using its knowledge of the available hardware features in the sysADG. The compiler then extracts



Compile Time
Relax DFG Complexity
App (C+pragma)
Decoupled-Spatial Compiler
memory-Dataflow Graph (mDFG)
Decoupled-Spatial Scheduler
Spatial Mapping Bitstream
FPGA
System-level ADG
Overlay Design
Domain Apps
Decoupled-Spatial Compiler
Different mDFGs for each App
Spatial Scheduling + FPGA PPA Model
Perf.: IPC
Resource: LUT%, FF%, BRAM%, DSP%
Modify system mADGs PPA
Design Space Explorer
FPGA Synth.
System-level ADG + RTL
Optimized ADG on FPGA Overlay
Model Setup
Processing Elements
Switches
Ports
Memories
Network
FPGA Synth. Tool
FPGA Utilization
Data Set Gen. & Training
Hardware Module with new specs.
ML-based FPGA Utilization Model
Predicted FPGA Utilization
System-level ADG + RTL

Figure 3: Overview of OverGen Framework

the memory access and computation from the program to construct a Memory-enhanced Dataflow Graph (mDFG); the mDFG is enhanced with information about array size, suitability for mapping arrays to scratchpads, and data reuse of each stream. The program represented as an mDFG is then mapped onto the ADG by the spatial scheduler, using the reuse information to make informed decisions. The mDFG could fail to map to the hardware; if so, the compiler will "relax" the DFG complexity by using less aggressive transformations (e.g. reduce unrolling degree [57]).

Overlay Generation: The input to the overlay generation is a set of workloads which forms the domain of interest. It is too inefficient to redo the compilation with each step of DSE. Therefore, the compiler generates a set of different mDFGs representing program versions that could be useful for different possible accelerators, and it incrementally recompiles these during DSE.

Compiled mDFGs are used to guide spatial-accelerator synthesis: all mDFGs are scheduled to an ADG, and the ADG is iteratively updated to maximize the objective (mean performance of the best-performing mDFG for each workload). There are four innovations over prior work: 1. the system and spatial accelerator are co-designed; 2. reuse and array information enables reasoning about memory and cache allocation at the spatial level, and 3. DSE balances FPGA resource utilization, and 4. the mDFG resource utilization guides ADG transformations with schedule-preserving transformations, described in Section V-B.

Finally, the chosen sysADG will then be lowered to synthesizable RTL for the FPGA, in part leveraging hardware generators from DSAGEN [38] and ChipYard [36]. DSAGEN's microarchitecture implementation is enhanced to enable pipelining on FPGAs with tight cycle-time constraints.

Model Setup: Our FPGA resource utilization model is based on per-hardware element models. Elements with a few parameters (e.g. core) can be exhaustively synthesized. For elements with many parameters, we use a machine-learning (ML) based model, trained from synthesizing a representative design space. Leveraging learned models means that this framework can more easily be ported to other FPGAs.

B. Overlay Design Space

System Design Space: Our target for the overlay is a homogenous multi-tile (i.e. multicore) where each tile contains an instance of the spatial accelerator, associated with a lightweight control core. Because we target highly-acceleratable workloads, the control cores are kept simple (single issue, small private cache), and are only provisioned for managing accelerator execution. The control cores and accelerators share access to a shared L2 cache over a crossbar-based NoC. Overall we explore the number of tiles, NoC bandwidth, L2 banks (for controlling L2 bandwidth), and L2 capacity.

Accelerator Design Space: The first order parameter of the accelerator is the number of processing elements (PEs), which determines the maximum compute bandwidth. The topology determines the flexibility, which can be characterized by the number and the radix of switches. We support a variety of functional units (FUs), with datatypes from 8 to 64-bit integer, and single/double precision float. PEs can have a wider bitwidth than each FU, in which case OverGen generates PEs supporting subword SIMD.

Streams for memory access and data manipulation execute on respective "stream engines":

- *DMA:* Memory engine for accessing shared L2.
- *Scratchpad:* Memory engine for private scratchpad.
- *Recurrence:* Communicating loop-carried dependencies.
- *Generate:* Generating affine value sequences.
- *Register:* Pulling data from accelerator to control core.

All stream engines have a parameterizable bandwidth. Memory stream engines have a capacity (scratchpad only) and parameter for whether parallel indirect access is supported (requires reordering hardware). Finally, *Ports* connect memory and compute units, enabling synchronization. Their width determines the maximum ingest/egest rates. *Ports* have a few additional parameters for supporting certain stream patterns (e.g. whether they support automatic padding for non-vector-width [56] and whether they support meta-data about whether the stream has computed a dimension of the loop [57]).

C. Key tradeoffs

OverGen opens a variety of tradeoffs that were previously difficult to explore and would have required manual effort:

Big tiles vs. More tiles: Many acceleratable workloads benefit from vectorization, while others are difficult to vectorize due to irregularity or loop dependencies. This leads to a tradeoff where some domains prefer more small accelerators (less pipeline/vector parallelism) or fewer large accelerators (more pipeline/vector parallelism).

L2 cache size vs. Scratchpad capacity: Some workloads have regular access to private data that can map to scratchpads, while less regular codes often benefit from hardware managed caches. Each domain requires a tailored allocation.

Balancing Bandwidths: The overall design space has essentially three levels of memory hierarchy, from shared cache to spatially distributed scratchpads, and reuse in the computation units. Allocating bandwidths across these levels requires understanding the compute bandwidth and data reuse possible in the chosen workloads — these decisions are tightly coupled with accelerator size and number of tiles.

Compute Density vs. Generality: If the goal of the overlay is to support either many workloads or dissimilar workloads, a more general overlay is required. This tradeoff can be made by constructing a flexible datapath at the cost of more resources, thus affecting all of the above tradeoffs.

IV. Spatial Memory Exploration

A. Motivating Spatial Memory DSE

Prior spatial architecture synthesis algorithms assume that all memory elements (scratchpads/DMAs) can communicate with all computation elements. While this simplifies the design space and spatial scheduling, it also prevents the DSE from exploring the best way to connect memories and processing elements together. Figure 4(a) shows an example design where memory stream engines communicate over essentially a crossbar to the spatial compute units. Figure 4(b) shows the potential of a system that allows spatial memories, where these engines have local communication with a smaller subset of elements. Similarly, extending this design space enables the possibility of deciding between multiple smaller scratchpads or a single unified scratchpad.

Making these decisions with existing DFG abstractions is difficult, as they lack two key pieces of information: 1. the relationship between access patterns and *data structures*, and 2. the size and reuse of these data structures. Together, these can enable reasoning about the validity and performance of spatial memory optimizations.

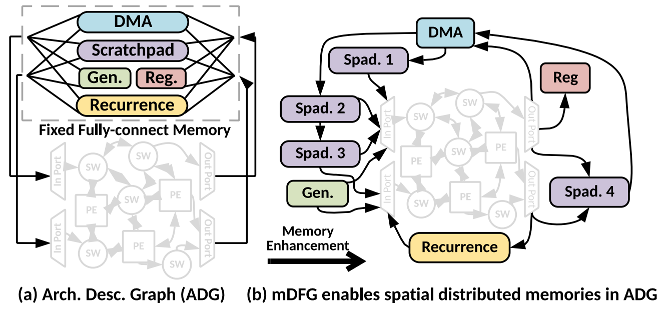

DMA
Scratchpad
Gen.
Reg.
Recurrence
Fixed Fully-connect Memory
In Port
Out Port
SW
PE
Spad. 1
Spad. 2
Spad. 3
Spad. 4
Memory Enhancement
(a) Arch. Desc. Graph (ADG)
(b) mDFG enables spatial distributed memories in ADG

Figure 4: Spatial Memory Enhancement for ADG

Memory-enhanced DFGs (mDFG): We enhance DFGs with data structure and reuse information by introducing *array nodes*, creating what we call a memory-enhanced DFG (mDFG). Array nodes have edges to streams that consume or produce those arrays, and we include reuse properties on streams. An example is in Figure 5 for a simplified version of `FIR`. Here, the input array `a` is stored in scratchpad for higher bandwidth requirement and reuse. The size parameter describes the total size allocated in either DRAM or scratchpad. If it is in scratchpad, the additional space of double-buffering is included. Also, streams are annotated with additional information for computing the reuse factor, including data traffic, data footprint, *stationary reuse*, and *recurrent reuse* (see the "Reuse Analysis" paragraph).

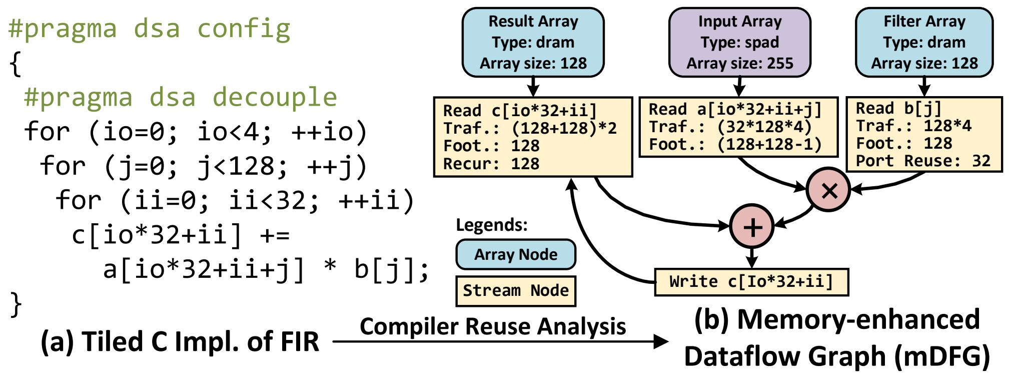
There is now sufficient information in the mDFG to decide which scratchpad to use — i.e., if data can be routed between the scratchpad node in the ADG and PE nodes that consume this data, and if there is enough remaining space in the scratchpad. If there is ever a limited capacity, the reuse information can help determine *which* array node in the mDFG should be mapped to a scratchpad node; for example, if an array has a stream with *stationary* reuse at the port, the benefit of exploiting reuse at the scratchpad level could be less than another array without stationary reuse. Note that the reuse information in the mDFG will also be used in the DSE for making system-level design decisions (Section V).

B. Software Support for Spatial Memory

To implement spatial memories, we extract array and reuse information from the program, and embed this in the mDFG to utilize during spatial scheduling.

Array Node Extraction: As it was discussed in Section II, all memory operations under the `stream` pragma are "restricted" (alias free), so we can extract the arrays involved in the dataflow graph by analyzing the pointer expressions. Specifically, we extract all the array pointers that are transitively used by all the decoupled memory operations. Consider the example in Figure 5(a): `a`, `b`, and `c` are extracted as array nodes. An array node has three attributes: pointer, footprint, data traffic, and memory reuse.

Reuse Analysis: Being aware of memory behaviors that can be captured by hardware specializations helps both compiler



```
#pragma dsa config
{
 #pragma dsa decouple
 for (io=0; io<4; ++io)
  for (j=0; j<128; ++j)
   for (ii=0; ii<32; ++ii)
    c[io*32+ii] +=
      a[io*32+ii+j] * b[j];
}
```

Figure 5: Memory Reuse Enhancement for DFG

optimization and DSE. Our compiler recognizes these patterns and annotates them on the associated stream nodes. Next, we discuss three typical reuse patterns, *general*, *stationary* and *recurrent* through the example in Figure 5(a) and (b).

General Reuse refers to when a memory stream repeatedly accesses a set of data within a program region. Scratchpad is often favored to exploit this reuse, provided there is sufficient capacity. Reuse can be identified by finding a discrepancy between data footprint (array or tile size) and traffic (number of uses). Consider the operand `a[i*32+ii+j]` from the innermost loop; the compiler recursively analyzes and joins the memory boundaries touched by each loop, and finally computes that 255 elements are in the memory footprint. To compute the data traffic, the compiler notes that every loop variable is involved in this pointer expression, which means a different element is accessed in each iteration. Thus, the data traffic of this operand is computed by multiplying all loop trip counts, i.e. $32 \times 128 \times 32 = 16384$. This indicates that each element is reused an average of $\frac{16384}{255}$ times. Indirect memory access, e.g. `a[b[·]]`, can also be analyzed similarly. To simplify, we assume: 1. `b[·]` is linear and can be analyzed by the above techniques; 2. no memory access will overflow, and the indirect memory access is a uniform distribution over array `a`. Therefore, data traffic is calculated by multiplying loop trip counts, and the data footprint is the size of array `a`.

Stationary Reuse refers to an operand repeatedly reused across the innermost loop so that this operand can be *stationary* in the compute substrate (e.g. the port FIFO). Consider the `b[j]` operand: Because the innermost loop `ii` does not involve the pointer expression, this value is reused across loop `ii` 32 times. Even though `b[j]` also has *general reuse*, it does not provide as much value to map to scratchpad, because much of the reuse is captured as stationary reuse (i.e. in the port).

Recurrent Reuse refers to when a pair of memory streams repeatedly update a set of data. When this set of data can concurrently fit in the data path pipeline and port FIFO, this pair of streams are favored to use the recurrence stream engine to avoid memory traffic. Consider the `c[io*32+ii]`: it repeatedly reads and writes a set of memory touched by `ii` (i.e. 32 concurrent instances) along with `j` (i.e. 32 recurrences). Therefore, when there is enough on-chip buffer for these 32 concurrent instances, this pair of streams will be mapped to the recurrence stream engine.

To sum up, reuse behavior captured by scratchpad, port FIFO, and the recurrence stream engine will all be considered as the reuse factor; this factor is used to calculate the bandwidth pressure of each stream in the DSE performance model (Section V-C).

mDFG Scheduling: Enhancing any spatial-scheduling algorithm to support mDFGs is straightforward. The principle is to treat array nodes (the ones representing the data structure), as any other node which must be scheduled onto the ADG, but with unique scheduling constraints. Intuitively, an array node can be mapped to a memory stream engine if:

1) There is sufficient remaining space (for a scratchpad).
2) There is a legal route from producers to consumers.
3) The access pattern of all streams for the array node is supported by the stream engine (e.g. indirect access).

The stream engines in our implementation allow more than one array each (as they support multiple concurrent streams), provided there is sufficient capacity. The tradeoff is that the bandwidth must be shared between any associated streams. Thus, even if it is legal to map more than one array to a scratchpad, it is sometimes beneficial to avoid sharing by using a different scratchpad or even just placing the array node onto a DMA stream engine; this can help maximize the utilization of available bandwidth.

Having reuse info on streams can help resolve these choices. For example, array nodes with stationary reuse at ports (e.g., read the same value *X* times) provide less benefit when mapped to scratchpads than those array nodes without stationary reuse – this is because their bandwidth consumption is already reduced. These factors must be considered during spatial scheduling; thus, we modify the objective of the spatial scheduler to use the projected performance of the mDFG, which factors in reuse and bandwidth bottlenecks. Because this is a critical portion of the system-level DSE, we explain the performance model in the next section (Section V-C).

V. Unified System & Accelerator Design Space Exploration

The goal of DSE in OverGen is to codesign the system parameters and accelerator features/topology to maximize FPGA performance on the set of input applications. Here we first give an overview, then discuss a novel technique to use prior schedules to guide spatial DSE, and finally discuss the performance and area modeling techniques.

A. *Overlay Design Exploration*

Logically, one iteration of the DSE involves proposing a new ADG for the hardware, recompiling all the workloads to it, and evaluating an objective (performance and FPGA resource use) to guide the next step of DSE — repeat until convergence. We use three main strategies to reduce the time for each DSE iteration.

First, we attempt to avoid recompilation as much as possible. During standard compilation, the compiler will iteratively back-off from aggressive transformations that require more resources than available (e.g. reduce the vector width and recompile). To avoid this during DSE, the compiler pre-generates different mDFGs for each program region which each use different transformations (different unrolling degrees, use a recurrence stream instead of accumulation, etc.). These different mDFGs are maintained during DSE, and ultimately only one of them needs to be used (only one has to schedule correctly to the ADG). While this increases the up-front cost for the first DSE iteration, it eliminates from-scratch recompilation during DSE.

Next, we also try to avoid the expensive spatial-scheduling stage of compilation by reusing the mDFG-to-ADG schedules from the prior iteration of DSE. A simple approach is to only re-schedule the portions of the DFG mapped to ADG elements that were modified (i.e. schedule repair [38]). In addition, we can use information about prior schedules to make a more informed decision about how to modify the ADG (see Section V-B).

Finally, we leverage the disparity between spatial scheduling time (very high) and system-level design-space exploration (quite low). Rather than explore both ADG design (spatial DSE) and system parameters (system DSE) at the same level of the DSE, it is relatively inexpensive to nest system DSE inside of spatial DSE – i.e. run a full exploration of system parameters every time we modify the ADG. This improves the convergence of the overall DSE.

DSE Flow Summary: The overall DSE flow is in Figure 6. At the beginning of each DSE iteration, the spatial DSE will propose a new ADG named ADG*. ADG* is constructed using a combination of random and schedule-preserving transformations (Section V-B). Then, mDFGs are rescheduled onto ADG*, leveraging the prior schedules for any unchanged portions of the ADG. If any program region has no successfully scheduled mDFGs, then ADG* is abandoned,

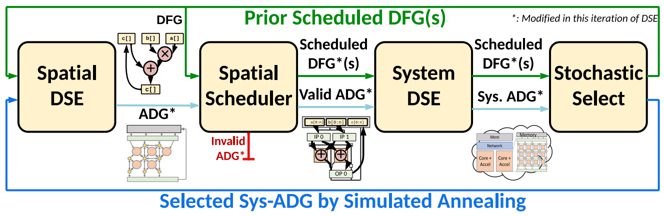


Figure 6: OverGen's Unified DSE Flow

and a new iteration begins. If not, then the system-level exploration (system DSE) exhaustively searches for the best system-design parameters for ADG* (creating sysADG*) based on estimated performance and resource constraints.

The objective function favors estimated performance first (Section V-C), followed by estimated resources-per-accelerator (Section V-D). This secondary objective encourages the spatial DSE to prune unneeded resources in the ADG, even if it does not lead to more cores or higher performance in the current DSE iteration. The final step is to choose whether to continue with this ADG*, which is done stochastically through a simulated annealing approach.

B. *Schedule Preserving Transformations*

During each DSE iteration where the Spatial DSE randomly modifies the hardware, it is common that some of the compiled DFGs can become invalidated due to hardware deletions or resource reduction. While this can sometimes be rectified by repairing the schedule to use other resources, it often cannot be. In these cases, the DSE algorithm either has to use a lower-performance schedule, less-vectorized DFG. The repair itself also takes a significant amount of time. This is unfortunate, because this can even happen when deleting units that are not necessary: e.g. a switch that is only used to pass through a value without requiring flexible routing.

Thus, we introduce the concept of schedule-preserving transformations, which use prior DFG schedules to guide hardware modifications that preserve their validity. Schedule preserving transformations are defined as hardware modifications that simplify the ADG while adding back the minimum capability to support the existing schedules. Thus, in essence, schedule-preserving transformations increase hardware utilization, providing further incentives for the removal of hardware units that provide less value. Specifically, we identified three such transformations:

Node Collapsing, as shown in Figure 7(a), occurs when a unit which performs routing (e.g. a switch) is deleted. Here, after the routing node is deleted, any routes on existing schedules that went through the node are used to define new direct hardware connections from their source to their destination. Thus, this transformation preserves prior schedules by ensuring a valid path for routes through a deleted unit.

Edge Delay Preservation, as shown in Figure 7(b), preserves the pipeline depth of all operands for a PE when an

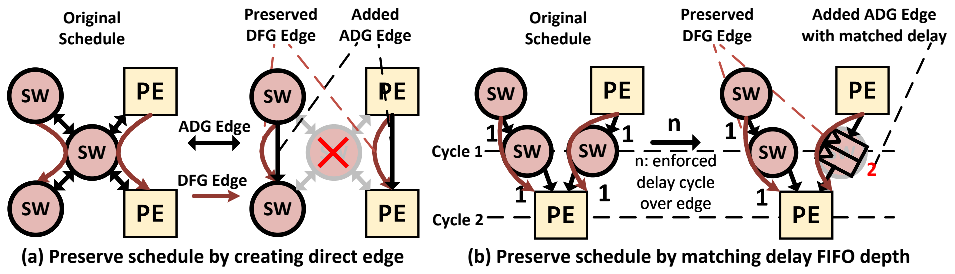

Original Schedule
SW
PE
SW
SW
PE
ADG Edge
DFG Edge
Preserved DFG Edge
Added ADG Edge
SW
PE
SW
PE
(a) Preserve schedule by creating direct edge
Original Schedule
SW
PE
Cycle 1
Cycle 2
SW
SW
PE
n
n: enforced delay cycle over edge
Preserved DFG Edge
Added ADG Edge with matched delay
SW
PE
SW
PE
(b) Preserve schedule by matching delay FIFO depth

Figure 7: Schedule-preserving Transformation Examples

intervening routing node is deleted. A balanced pipeline depth ensures that all operands arrive at the same time to avoid pipeline bubbles; these bubbles can lower the throughput of the spatial accelerator [59]. Our approach is to increase per-operand FIFO-depths in the PE (called delay-fifos) whenever this imbalance can be observed on an existing schedule.

Module-Capability Pruning prunes excess module capabilities, and associated hardware, that are not needed by mapped schedules. Without this transformation, the DSE oftentimes does not have enough incentive to remove some costly capabilities, and more frequently removes capabilities that are actually useful.

C. Performance Model with Spatial Memory

To estimate performance, we implemented a bottleneck-based analysis that captures system-level design parameters, memory bandwidth at different layers, and computational bandwidth. Specifically, the overall performance is calculated as the weighted geometric mean of the estimated IPC for each mDFG. An mDFG's IPC is calculated by multiplying the maximum instruction bandwidth (mDFG Insts) by the number of tiles and by the lowest bottleneck factor of all levels of the memory system:

$$\text{Perf} = (\text{mDFG Insts}) \cdot (\text{\# of Tiles}) \cdot \min_{L_1 \dots L_3} \left(\frac{\text{R}_{\text{Production}}}{\text{R}_{\text{Consumption}}} \right) \qquad (1)$$

The *mDFG Insts* factor captures vectorization degree, allowing the DSE to explore tradeoffs between higher vectorization degrees and number of tiles. Memory operations, namely load and store operations, are included within the estimated IPC to ensure that vectorization of pure data-movement DFGs is incentivized.

While counting loads and instructions estimates the ideal IPC, memory bandwidth limitations reduce the observed IPC, as memory subsystems cannot always supply enough data to fulfill computational requirements. We compute the most-bottlenecked performance reduction over L_1, L_2, and L_3, corresponding to the Scratchpad, L2 Cache, and DRAM. This bottleneck factor is calculated by dividing the production and consumption rate (as $\frac{R_{\text{Production}}}{R_{\text{Consumption}}}$ in the previous equation). These factors are calculated as follows, taking into account stream reuse factors (see Section IV-B):

$$\text{R}_{\text{Production}} = BW_{L_N, L_{N+1}} \cdot (\text{\# of Banks})$$
$$\text{R}_{\text{Consumption}} = \sum_{i=1} \left(\frac{\text{BW}(\text{Stream}_i)}{\text{Reuse}(\text{Stream}_i)} \right) \cdot (\text{\# of Shared Tiles}) \qquad (2)$$

In the above equations, the production rate is computed by multiplying the bandwidth and bank count at each memory level. The consumption rate, or data needed to satisfy compute bandwidth, is the sum of compute data required by a single tile, multiplied by the number of tiles at that memory hierarchy level. The single-tile required data is computed as the summation of all stream bandwidths divided by their associated reuse rates. We describe how the bandwidth (BW) and reuse factors are computed at each level:

Scratchpad Bandwidth: With scratchpads replicated across tiles, the *# of Shared Tiles* factor is one, making the bandwidth only depend on vectorization degree. Also, the bandwidth is calculated separately for the read and write port.

L2 Bandwidth: As L2 Bandwidth is shared amongst tiles, the consumption rate increases with respect to tile count, requiring more banks. Accesses to L2 cache occur when a stream pattern cannot be supported by port reuse or recurrent data stream, without which the required data production rate will be increased – thus demanding more L2 Banks.

DRAM Bandwidth: Similar to L2 bandwidth, the consumption rate is dependent on both reuse and tile count; however, the total FPGA's DRAM bandwidth is fixed.

D. ML-based FPGA resource model

To rapidly predict FPGA resources, the DSE leverages a machine-learning (ML) resource prediction model, which estimates resources on a component-level basis. To generate the ML model, we perform out-of-context synthesis on variations of each hardware unit, shown in Table I, to train an ML-based FPGA resource model. The component-level ML model implements a 3-layer multi-layer perceptron (MLP), with an 80%/10%/10% test, train, and validation data split. As the FPGA resource model was synthesized out-of-context with no synthesis optimization passes being performed, our model behaves pessimistically – the projected design point is larger than the actual post-PnR result.

Hardware Unit	Total Synthesized
Processing Elements	100,000
Switches	56,700
Input Port	34,412
Output Port	25,796

Table I: Number of Hardware Modules Synthesized

VI. OverGen Microarchitecture & Implementation

Our implementation of OverGen integrates prior frameworks for spatial architecture generation [38] and SoC generation [36] – both implemented in Chisel [60]. We also extend these frameworks with an implementation of spatial memories and a high-utilization memory pipeline suitable for FPGAs. In this section, we first discuss the microarchitecture of generated accelerators, show how they interact with the rest of the system and introduce our implementation of two key components of OverGen: the *stream dispatcher* and *stream engine*.

A. Implementation Overview

Figure 8 shows a high-level block diagram of an example dual-tile OverGen overlay architecture mapped to a Xilinx VCU118. Each OverGen tile (colored in light-blue) is composed of a RISC-V Rocket [61] control core (colored in orange) and one instance of the spatial accelerator. The control-core sends commands and synchronizes with the accelerator over the RoCC interface [62], and the spatial accelerator uses a TileLink [63] DMA for memory access.

Within the spatial accelerator, the *stream dispatcher* connects the control-core to the spatial memory system, and coordinates stream execution. All units inside the spatial memory system are *stream engines*, whose responsibility is data movement to and from ports and memories. Stream engines share a common pipelined implementation.

In the remainder of this section, we discuss how we achieve high utilization in the stream-dispatcher and stream-engines.

B. Stream Dispatcher Microarchitecture

The stream dispatcher is designed to connect the control core to the spatial memory system, and manage stream execution for an arbitrary number of stream engines. Figure 9 shows the hardware organization, and we explain intuitively by describing streams' execution over their lifetime. Each stream's lifetime has three steps: `stream config`, `stream instantiation` and `stream synchronization`.

1. The control core communicates stream parameters and commands that finalize stream creation to the *stream dispatcher*. The stream dispatcher holds a register file for these parameters so that they can be reused if unchanged across streams. The `stream config` step updates this register file.
2. When a stream finalization command is sent, the `stream instantiation` step will decode the register values and create an elaborated stream entry for its corresponding stream engine in the stream dispatch queue.
3. The stream dispatch queue uses a basic Tomasulo algorithm [64] at `stream synchronization` to see whether its required resource (stream engines, ports, etc.) are currently being used by other stream entries. If yes, the newly created stream entry will be put into a dispatch queue, waiting for the required resources to be idle, and then it will be dispatched to its destination. Dispatch is out-of-order, but respects per-port request order. The `stream synchronization` step is done by stream barrier queue, where the stream synchronization commands are queued. The stream synchronization command encodes certain ports / stream engines resource. It blocks streams to be dispatched if the ports / stream engines it specified are busy. By doing so, stream barrier queue can synchronize between

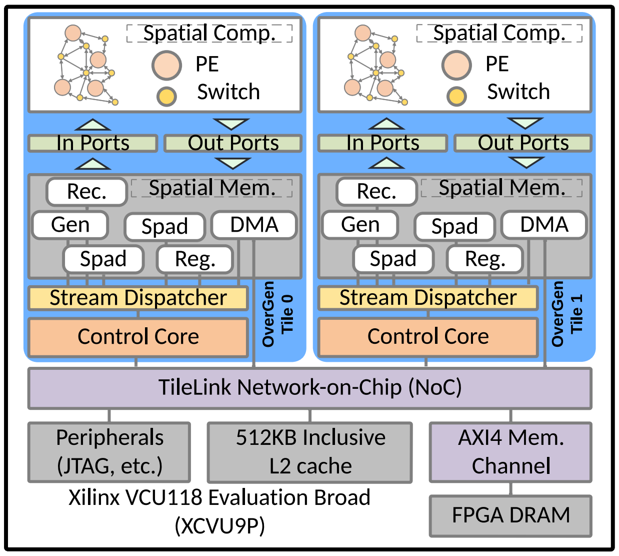


Figure 8: Example Dual-tile OverGen Overlay

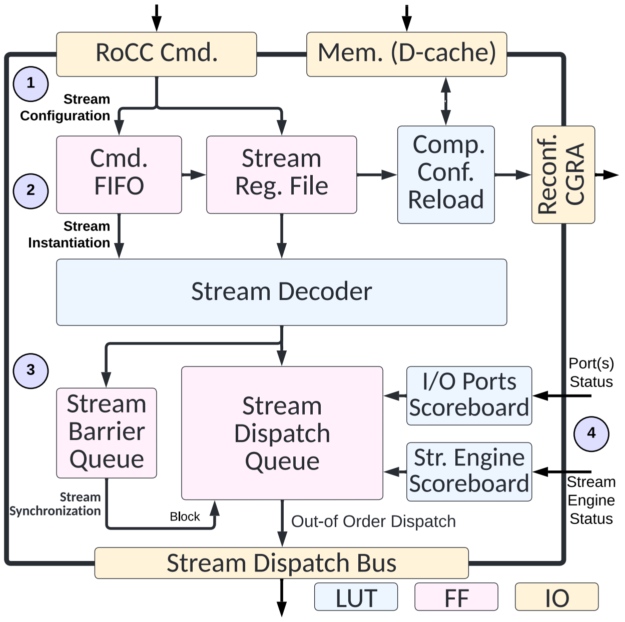


Figure 9: Stream Dispatcher Microarchitecture

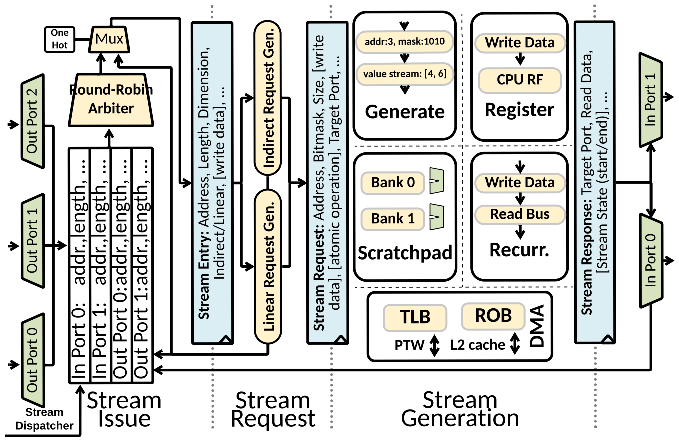

One Hot
Mux
Round-Robin Arbiter
Out Port 2
Out Port 1
Out Port 0
Stream Dispatcher
In Port 0: addr.,length, ...
In Port 1: addr.,length, ...
Out Port 0:addr.,length, ...
Out Port 1:addr.,length, ...
Stream Entry: Address, Length, Dimension, Indirect/Linear, [write data], ...
Indirect Request Gen.
Linear Request Gen.
Stream Request: Address, Bitmask, Size, [write data], [atomic operation], Target Port, ...
addr:3, mask:1010
value stream: [4, 6]
Generate
Write Data
CPU RF
Register
Bank 0
Bank 1
Scratchpad
Write Data
Read Bus
Recurr.
TLB
ROB
PTW
L2 cache
DMA
Stream Response: Target Port, Read Data, [Stream State (start/end)], ...
In Port 1
In Port 0
Stream Issue
Stream Request
Stream Generation

Figure 10: Stream Engine Microarchitecture & Pipeline

different stream entries, preventing data hazard.

4 *Stream engine* performs stream memory access once the elaborated stream entry is dispatched. It frees the resource in the scoreboards on stream completion.

Performance Characteristics: This unit can dispatch one stream per cycle, and up to N number of streams can complete per cycle (N = number of total stream engines). The minimum latency of RISC-V instruction completion to stream dispatch is 2 cycles (one for parameter configuration, one for dispatch if no resource conflict found).

System Integration & Reconfiguration: The *stream dispatcher* is also responsible for bridging other interfaces to the accelerator side for the purpose of system integration. OverGen-generated accelerators attach to the Network-on-Chip (NoC) directly to coherently share a banked inclusive L2 cache with other cores. The accelerator shares the page table walker (PTW) of control-core for local TLB support.

The accelerator also uses D-cache to load its configuration bitstream to re-program the computing substrate (Figure 9 right edge). Such reconfiguration bitstream reload is triggered by a write to *bitstream address* and *bitstream size* registers in stream register file. When this bitstream is returned from D-cache, it passes through a customized network to perform reconfiguration on the spatial computing network.

C. Stream Engine Microarchitecture

Our goal is to design a modular stream engine generator that works for the combinations of supported stream patterns (1D/2D/3D $\times$ Affine/Indirect $\times$ DMA/Scratchpad). Each feature can be turned off individually to save resources if not needed in a given domain. Figure 10 shows the micro-architecture and pipeline design for stream engines.

Overview of Common Stages: The first stage, `Stream Issue`, is where the *stream table* receives decoded stream entries from the *stream dispatcher*. The *stream table* selects one stream entry to be sent to the `Stream Request` stage for every cycle, together with its data payload (e.g. write data, indirect index value).

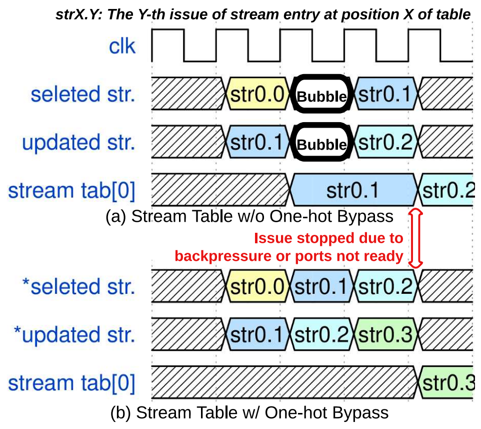

strX.Y: The Y-th issue of stream entry at position X of table
clk
seleted str.
updated str.
stream tab[0]
str0.0
Bubble
str0.1
str0.1
Bubble
str0.2
str0.1
str0.2
(a) Stream Table w/o One-hot Bypass
Issue stopped due to backpressure or ports not ready
*seleted str.
*updated str.
stream tab[0]
str0.0
str0.1
str0.2
str0.1
str0.2
str0.3
str0.3
(b) Stream Table w/ One-hot Bypass

Figure 11: Stream Table One-hot Bypass

The `Stream Request` will generate the memory request packet based on the elaborated stream entry. After accessing memory blocks or obtaining the expected value at the `Stream Generation` stage, the responses (only for memory read or recurrence) will be forwarded to a Re-order Buffer (only for DMA and indirect scratchpad), where the responses are re-ordered in request order. The responses will eventually be sent to its destination input ports and eventually consumed by In Port(s).

Stream Issue: After being dispatched, the stream entry first arrives at the *stream table*, where the meta information for each stream are recorded. The *stream table* aggregates the readiness across all valid streams and selects one to be issued to the stream request generator. The readiness of each stream is determined by whether any associated input ports have enough space (read stream) and output ports have enough data to consume (write stream).

The stream table is designed to be fully pipelined. The difficult case is when there is only one active stream. Since the stream table needs to hold the outstanding meta info for each stream, it is designed to be flip-flop based. Thus, the updated stream entry can only be reflected in the next cycle and issued in the subsequent cycle, creating a pipeline bubble. To fully-pipeline a single stream, a one-hot detector is added beside the stream table, where if only one stream is active, the stream table will be bypassed by the updated stream entry from the stream request stage. Figure 11(a) shows the bubble in the waveform without the bypass, where the issue rate is one every two cycles. Figure 11(b) shows that adding the bypass doubles the issue rate.

Stream Request: After being issued from the *stream table*, stream entries will be converted to memory request packets (address, mask, read/write, etc.) in the `Stream Request` stage. For affine stream patterns, the task of the request generator is to generate a memory access bitmask based on the address and number of bytes to be accessed, described in TileLink protocol [63]; As for indirect stream patterns, an indirect request generator containing a set of adders is introduced. Such adders are used to add the start address of stream `a` with its multiple index values (values of stream `b`) to calculate the actual addresses of indirect streams like `a[b[i]]`. The `Stream Request` stage is also responsible for calculating the next-cycle stream state that will be written back to the *stream table* for the next request (or bypassed when there is only one stream).

Stream Generation: The `Stream Issue` and `Stream Request` stages create continuous requests that will eventually produce data that will be consumed by the computing fabric in the `Stream Generation` stage, which is specific to each stream engine.

DMA accesses virtual memory, where memory requests from `Stream Request` will 1. reserve an entry in the ROB; 2. access a private TLB and PTW shared with the control-core; 3. Memory request interface connects directly to NoC, which allows accelerator access to L2 cache (LLC) directly; 4. Memory response will be sent to ROB to complete the memory transaction.

Other units are intuitive: The Generate Engine generates affine value sequences, similar to the patterns supported by affine memory streams. The Recurrence Engine forwards write data payload from output ports directly to input ports. The Register Engine enables scalar value collection from an output port to control-core directly.

D. OverGen Implementation

Specific design constraints should be followed to maximize the FPGA resource utilization (for larger or more accelerators) and minimize the critical path (for higher frequency). Besides resource constraints, FPGA requires careful consideration on timing since Configurable Logic Blocks (CLB) are pre-placed and clock sources are pre-defined. Bad implementations (e.g. large combinational logics) or complex designs (e.g. multi-clock region) can significantly hurt the frequency. Therefore, OverGen follows two design principles to attempt to solve these two challenges:

1) Conservatively design and add extra pipeline stages while maintaining fully-pipelined execution.
2) Build each module by using pre-built FPGA IP for higher frequency and better utilization.

Conservative pipeline: One challenge is the added delay of cross-die interconnects on the latest FPGAs [65]. These are present on Xilinx FPGAs, which use multiple dies connected by silicon interposers in order to increase the number of logic elements on a single device. In addition, specialized IP blocks such as the DRAM controllers or HBM controllers have fixed locations, and interacting modules are also more constrained in their layout. Together, these factors lower the final clock frequency.

To mitigate the timing degradation caused by die-crossing delays, we conservatively insert additional pipelines. We explicitly add extra stages between the *stream dispatcher* and all *stream engines* to relax the timing budget on the stream dispatch bus. Moreover, the DMA engine is responsible for main memory access, which requires it to closely interact with the NoC and then the DRAM controller. The fixed-location of the DRAM channel on the FPGA encourages per-tile DMA engines to be placed near the DRAM controller, as shown in Figure 12, so we also add extra pipeline stages inside DMA read/write ports that connect to the NoC. The *stream engines* are also conservatively pipelined, as shown in Figure 10, because they bridge other accelerator pieces: stream dispatcher, in/out ports, and compute fabric.

FPGA IP Optimization: We adapt the overlay design to make use of the specialized memory and DSP blocks available on the FPGA fabric. For example, using the Block RAM hard blocks for scratchpad and ROB. Also, mapping floating point computation to dedicated DSPs significantly increases the achievable frequency compared to mapping with LUTs.

E. Limitations & Future works

Threading Interface: The current pthread-like programming interface assumes a one-to-one mapping of threads to tiles, where threads run to completion uninterrupted. Also, the performance models assume that all tiles are parallelizing the same code region, and this is our convention when implementing kernels. We also do not manage the interaction between host and FPGA in terms of offloading or data movement. A more sophisticated programming interface, task model (e.g. [66–69]), and analytical models could significantly expand usability.

Processing Elements: Our current implementation of processing elements only supports a dedicated instruction execution model; in contrast, the use of *shared* PEs (either static [70,71] or dynamically scheduled [56,72,73]) can potentially support kernels with larger code regions and get higher utilization for kernels with more complex control flow.

Compilation Support: Although our processing elements already support a predication-based control lookup table for conditional execution, our compiler has only limited support for converting arbitrary control flow to predication based dataflow execution. A more general dataflow control flow model (e.g. [74,75]) is future work. Meanwhile, our compiler only supports data parallel loop unrolling when exploring DFG resource occupation (i.e. DFG size). When it comes to exploiting overlapping data reuse between subsequent loop iterations, we still require manual unrolling to take advantage. This can be improved by integrating prior work on reuse distance analysis [76]. Also, our reuse analysis relies on strong assumptions on compilation-time determined loop trip count and array shape. One of our future directions is to support dynamical array shape and loops.

	Workload	Size	Type	#ivp	#ovp	#arr	#m,a,d
DSP	cholesky	48^2	`f64`	7	3	2	5,4,2
	fft	2^{12}	`f32x2`	3	1	2	4,8,0
	fir	$2^{10} \times 199$	`f64`	4	2	2	4,4,0
	solver	48^2	`f64`	4	2	2	4,4,1
	mm	32^3	`f64`	4	3	3	4,4,0
MachSuite	stencil-3d	$34^3 \times 8$	`i64`	7	1	2	4,12,0
	crs	494×4	`f64`	6	5	6	1,0
	gemm	64^2	`i64`	4	2	3	8,8,0
	stencil-2d	$66^2 \times 3^2$	`i64`	3	1	2	9,11,0
	ellpack	494×4	`f64`	4	3	4	4,4,0
Vision	channel-ext	$128^2 \times 4$	`i16`	1	1	2	0,0,0
	bgr2grey	$128^2 \times 4$	`i16`	3	1	2	16,32,4
	blur	$128^2 \times 4$	`i16`	3	1	2	0,52,8
	accumulate	$128^2 \times 4$	`i16`	2	1	2	0,16,0
	acc-sqr	$128^2 \times 4$	`i16`	2	1	2	16,16,0
	vecmax	$128^2 \times 4$	`i16`	2	1	3	0,16,0
	acc-weight	$128^2 \times 4$	`i16`	5	1	2	32,16,4
	convert-bit	$128^2 \times 4$	`i16`	3	1	2	0,32,0
	derivative	$130^2 \times 4$	`i16`	3	1	2	16,32,4

Table II: Workload specification: size, data type, input/output ports, and multiply, add, div ops in the best DFG.

VII. Methodology

Benchmarks: We selected 19 workloads from different domains: 9 from Xilinx Vitis computer vision library, 5 from the *digital signal processing (DSP)* domain targeted by REVEL [56], and 5 from MachSuite [77] for commonly-accelerated workloads. The data size and data type are shown in Table II.

Baseline: We evaluate OverGen in terms of speedup, compilation, DSE time and device reprogram time. We compare against the state-of-the-art HLS technology, AutoDSE [21], as our baseline by using Merlin Compiler (2020.3) and Xilinx Vivado (2020.2). Because AutoDSE benefits significantly from manual kernel tuning, we evaluate both against non-tuned and tuned code versions for AutoDSE.

Compiler support: We augment the open-source DSAGEN [38] compiler with spatial memory support. An extended Clang and LLVM compiler transform the pragma-annotated program into RISC-V assembly, and the RISC-V GNU toolchain is modified for binary generation.

Hardware Generation & Verification: OverGen augments the Chisel-based DSAGEN hardware generator [38] by extending it to full system-level with a modular spatial memory system as described in Section IV. After obtaining RTL from hardware generation, we further verify the functional completeness as a full system with RISC-V binaries on RTL cycle-level by using Synopsys VCS before FPGA verification.

System-Level Integration & Experiment Platform: Each accelerator is integrated into ChipYard [36] as an RoCC accelerator to a small RISC-V Core (Rocket Core). All designs use an 8-way associative directory-based inclusive L2. The generated RTL is further synthesized to Xilinx VCU118 Evaluation board by using Vivado 2021.2. All data for each kernel begin offchip and are loaded from FPGA DRAM.

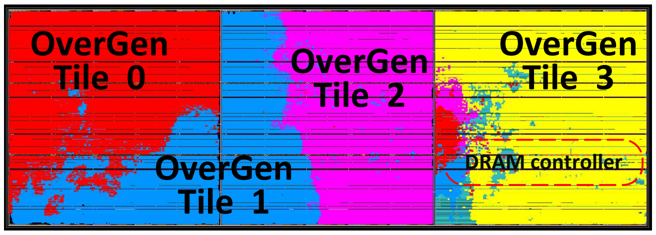


Figure 12: Quad-Core OverGen FPGA Floorplan

Because of FPGA implementation difficulties, we were not able to run on our FPGA when multiple DRAM channels were enabled. Thus, we use a single DRAM channel for most experiments, and study the effect of multiple DRAM channels separately using VCS RTL simulation (Eval. Q7).

Figure 12 shows the floorplan of a Quad-tile General OverGen design at 92.87MHz, including the DRAM controller's location. The critical path is around the L2 MSHR logic, and optimizing is beyond the scope of this work.

VIII. Evaluation

The goal of our evaluation is to provide perspective on the opportunities of synthesized spatial overlays as compared to state-of-the-art automated HLS (AutoDSE). This section is organized around 8 key questions, with the takeaways being:

- OverGen is able to generate reconfigurable designs that can outperform baseline AutoDSE (without kernel tuning) by mean 1.2×, even though the generated designs are more flexible.
- HLS benefits more heavily from kernel tuning, while OverGen's execution model and compiler can handle many code patterns natively without software effort.
- New applications within the same domain can be easily deployed on an existing overlay with only modest performance degradation, due to overlay flexibility.

Q1: How performant are generated overlays?

Figure 13 shows the overall performance of OverGen across all workloads, normalized to AutoDSE without kernel tuning. We demonstrate three different kinds of overlays:

- **General Overlay** (second bar): A single hand-designed mesh-based accelerator overlay targeting all workloads with maximum vectorization width (512 bit).
- **Suite Overlay** (third bar): An overlay specialized to each workload suite. Table III shows the specs of each.
- **Workload Overlay** (fourth bar): An overlay specialized only to a single workload.

We first compare against AutoDSE without manual kernel tuning. The general overlay achieves comparable performance to AutoDSE on the DSP suite and MachSuite, and mean 68% of the performance on vision suite. This is because it can only fit at most 4 general tiles, due to the high overhead of the general overlay's datapath and FUs (about

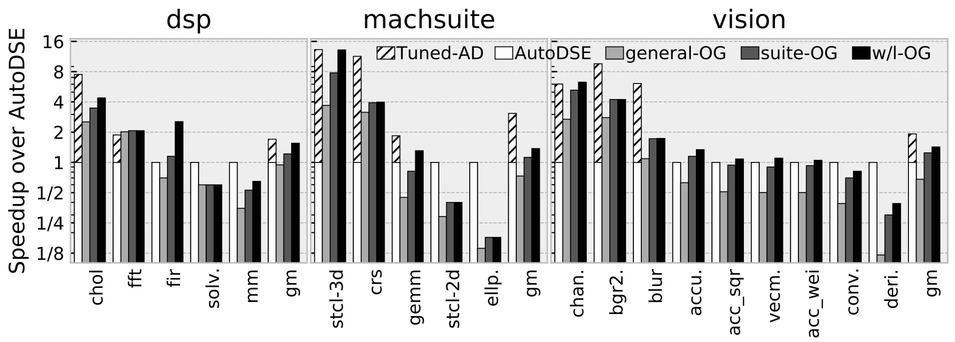


Figure 13: Overall Performance Comparison

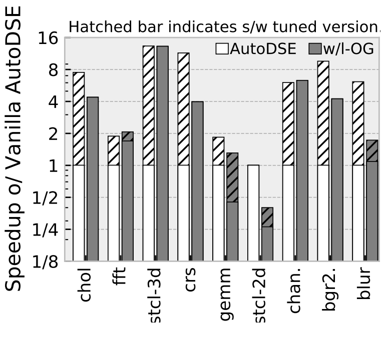


Figure 14: Effect of tuned kernels

	Spec.	Mach.	Vitis	DSP	General
System	Tile Count	10	13	7	4
	L2 #Bank	16	16	8	4
	NoC B/W (Byte)	64	64	64	32
Accelerator	PEs	20	16	10	24
	Switches	17	11	27	35
	Avg. Radix	2.9	2.61	2.85	4.69
	Int $+/\times/\div$	16/14/0	16/15/13	0/0/0	24/24/24
	Flt. $+/\times/\div/\sqrt{x}$	4/4/0/0	0/0/0/0	6/6/5/2	24/24/24/24
	Spad. Cap. (KB)	64	-	8, 32	32
	Spad. B/W (B/cyc)	32	-	32, 32	32
	Spad. Indirect?	Yes	-	No, No	Yes
	GEN/REC/REG	0/0/0	0/0/0	0/1/0	1/1/1
	In Ports B/W (B)	160	112	152	224
	Out Ports B/W (B)	96	48	104	160

Table III: Specification of Suite Specific Overlays

52% in LUT). The per-suite specialized overlays outperform baseline AutoDSE by a mean 1.2×, primarily due to having 2-3× more tiles (i.e. due to specialized network, FUs, and memories). Additionally, the DSP overlay uses two scratchpads to increase bandwidth without requiring more expensive wider accelerator datapaths. The per-workload specialized designs can outperform AutoDSE without kernel tuning by mean 1.45× for similar reasons; the relative improvement over suite-specialized is modest, especially for Vitis, due to the strong similarity between workloads.

Compared to AutoDSE with manual kernel tuning, OverGen is able to achieve 0.71×, 0.37×, 0.65× of performance for DSP, Machsuite and Vision respectively, while still maintaining workload-flexibility. This is sensible, as hardware structures for preserving generality and programmability reduce the maximum resource efficiency; Q2 goes into depth on why kernel tuning is more critical for AutoDSE.

While most suite overlays were at least half the performance of the AutoDSE designs, there were a few outliers. Both `stencil-2d` and `derivative` both apply aggressive reuse optimization through a sliding window, which can be well specialized by line buffer architecture on HLS [78]. For `ellpack`, we have to load a vector to the scratchpads of all cores, but we currently lack broadcast support from DRAM to scratchpad, which wastes significant bandwidth; incorporating stream-based multicast [79] would be helpful.

Q2: Impact of kernel tuning across frameworks?

We studied 9 workloads that benefit from kernel tuning, as shown in Figure 14. There are 7 workloads where AutoDSE (and its underlying HLS technology) does not handle some code patterns well, leading to lower performance because of increased initiation interval (II: number of cycles between pipeline compute instances). In general, these patterns are more easily supported on OverGen's ISA/compiler. To substantiate this, we manually transform these 7 workloads to improve their II for AutoDSE, and we found 4 opportunities for kernel tuning in OverGen.

AutoDSE Kernel Tuning: We find that two main manual transformations are useful in these workloads: eliminating variable loop trip counts, and strength reduction for strided access patterns. Table IV shows the II's before and after these transformations, and the hatched bar in Figure 13 and Figure 14 shows the tuned workloads' performance. Note that all other workloads achieve II=1, and OverGen *always* achieves II=1. We next discuss each transformation and the affected workloads.

Causes	Var. Loop TC			Inefficient Strided Access			
Workload	chol.	crs	fft	bgr2.	blur	chan.	stcl-3d
Untuned II	10	4	2	9	6	8	6
Tuned II	5	2	1	1	1	1	1

Table IV: HLS Initiation Interval (II) Optimization

Variable Loop Trip Count: HLS prefers a perfect loop nest with fixed trip-count [78], but `cholesky`, `fft`, and `crs` all have variable trip counts or imperfect loop bodies. To transform these programs, we replace variable trip counts with a fixed maximum, and push outer-loop computation into the inner loop. We then guard the conditional execution with if-statements within the inner loop. OverGen supports variable trip-count streams natively (using REVEL's ISA [56]).

Inefficient Strided Access: AutoDSE's toolchain has trouble efficiently performing strided memory access with small strides (including accesses that appear strided when observing only the innermost dimension of the access pattern). Such patterns can limit AutoDSE's ability to exploit memory parallelism, either at the BRAM level with multiple ports, or at the DRAM level with memory request coalescing. To help the underlying HLS tools understand the access pattern better, the solution is to perform a strength reduction on any strided accesses using the innermost induction variable (e.g. instead of using `i * 4`, increment `i` by `4` in each iteration). OverGen's compiler natively supports strided streams and coalescing adjacent streams.

Prebuilt Database: AutoDSE has a pre-built database that records the best explorer configuration of AutoDSE for common workloads. `gemm` is optimized using this database.

OverGen Kernel Tuning: These software behaviors of interest are more easily captured by the OverGen compiler, so only 4 workloads benefit from source code transformation on OverGen. For `fft`, we peel the last several iterations, so that strided scalar access can be coalesced to fully utilize the memory bandwidth [38,57]. For `gemm`, to minimize I/O traffic into the accelerator and improve reuse, we unroll across two inner-loop dimensions (similar to tensorization [80]). For `stencil-2d` and `blur`, our compiler has limited support for exploiting reuse from overlapped data access between subsequent iterations. Therefore, we manually unrolled the iterations to reuse the overlapped data.

Overall, while kernel tuning is a helpful avenue for performance improvement in AutoDSE's HLS-based approach, it also more often requires programmer effort to get competitive performance than OverGen for this set of workloads.

Q3: How fast is OverGen's DSE?

Figure 15 shows the DSE and synthesis time comparison between AutoDSE (first bars in each suite) and the suite-wise OverGen overlay (right-most hatched bar). Comparing AutoDSE's combined time of synthesizing each application, our DSE constructs a more general accelerator while using only 47% of the time.

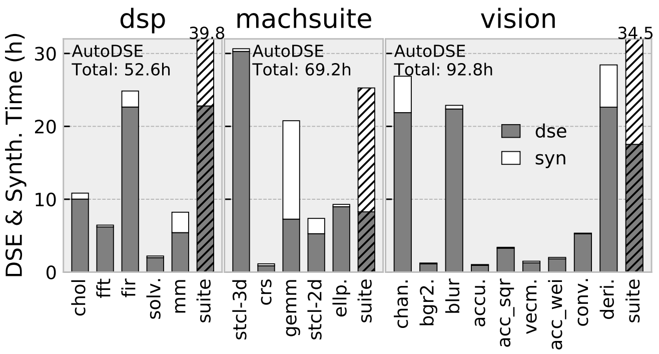
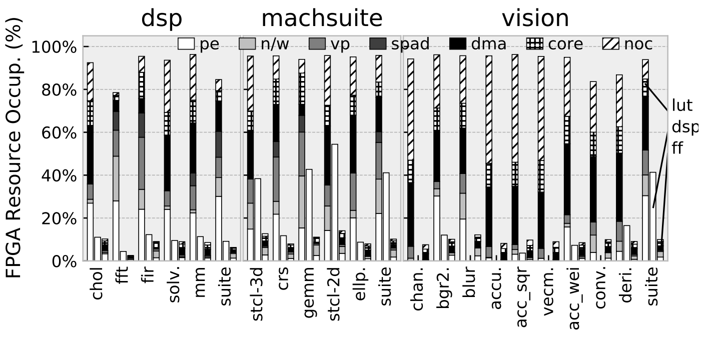


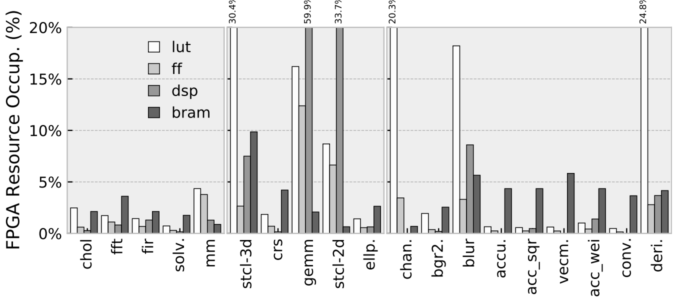
Figure 15: DSE and synthesis time comparison.

Q4: What are the limiting FPGA resources?

Figure 16 shows the resource breakdown of each component, normalized by the total FPGA resources available for both overlay and AutoDSE designs (with kernel tuning). All the generated overlay designs (both per-workload and suite) consume from 81% to 97% of LUTs, which is the limiting factor. Because we would like to preserve some



(a) Overlay Designs



(b) AutoDSE Design

Figure 16: FPGA Resource Breakdown

generality for potential future workloads, our DSE greedily consumes as many resources as possible, even if there is no parallelism or when we are memory bandwidth bound. One of the biggest components in terms of LUTs is the NoC, due to its crossbar-based implementation (prior work observed similar overheads [42]). AutoDSE tends to consume fewer resources as it favors utilizing less hardware when memory bound or parallelism bound, as generality is not a goal.

Q5: Can additional workloads be mapped to an overlay?

We perform a "leave-one-out" experiment to study the overlay flexibility. Specifically, we generate an overlay for all but one workload in a suite, then try to map the remaining workload. If that workload can map with relatively high performance, that indicates a more robust design.

The results are shown for MachSuite in Figure 17. Most of the workloads can be mapped to the corresponding leave-one-out accelerator, with mean 49.5% performance degradation. Performance loss is caused by datapath specialization, which prevents the optimal spatial mapping; generally, a less-vectorized version is used, which has commensurately less performance. The modest performance loss may be acceptable to an FPGA programmer making incremental changes. We imagine that the compiler could inform the user when a significant performance improvement is expected, to signal when to perform DSE again.

We use the same setup to evaluate the compile/reconfiguration time, as compilation time is most meaningful on an overlay that was not specifically designed for that workload.

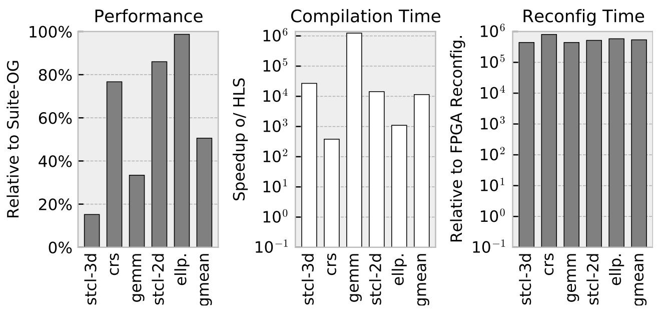


Figure 17: "Leave-one-out" Flexibility Evaluation

Comparing against AutoDSE-based HLS, our spatial overlay compilation is 10000× faster. Also, reconfiguration is much faster by mean 54000×. This is useful if the desired FPGA functionality changes rapidly, enabling efficient temporal multiplexing at very fine time scales.

Q6: How does overlay-generality affect performance?

OverGen can be used to generate increasingly general designs by incrementally adding more target workloads. Figure 18 shows the results of such an experiment, where we incrementally add workloads and rerun the DSE to analyze how the number of tiles and resource usage changes. We witness the overall datapath (PE + Port + network) use per tile increases as new workloads are added to the target set, because the datapath becomes more general. To compensate, the number of tiles decreases from 15 to 10. Because some of the workloads are memory bound, it only costs mean 8% performance to support all workloads in this suite.

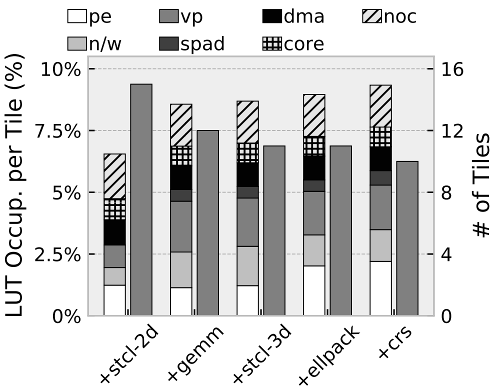


Figure 18: Incremental Design Optimization.

Q7: How do more DRAM channels affect performance?

Figure 19 shows the performance with varying DRAM channel count, normalized to single-channel DRAM for each design. For AutoDSE, most MachSuite kernels can benefit from multiple DRAMs by mean 25%. Element-wise memory-intensive workloads like `mm`, `gemm`[3], `vecm.`, `accu.`, `acc_sqr`, `acc_wei` and `deri.` can also benefit

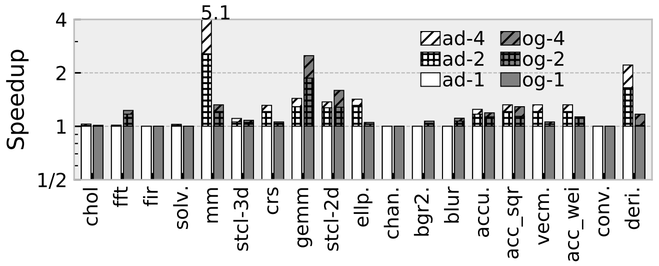


Figure 19: Effects of DRAM channels

from multiple DRAM channels. The OverGen Workload Overlays see benefits on a similar set of workloads by mean 19%.

Q8: Do schedule-preserving transforms improve DSE?

Figure 20 compares the DSE algorithm with and without schedule-preserving transformations. Here the x-axis is time in hours, and the y-axis is the DSE's estimated IPC for the whole FPGA. Schedule-preserving transformations help the DSE converge faster to designs that are more-specialized to the workload datapath topologies. Overall, DSE time is reduced by mean 15%, and the estimated IPC is improved by 1.09× (running on the FPGA confirms 1.08x speedup).

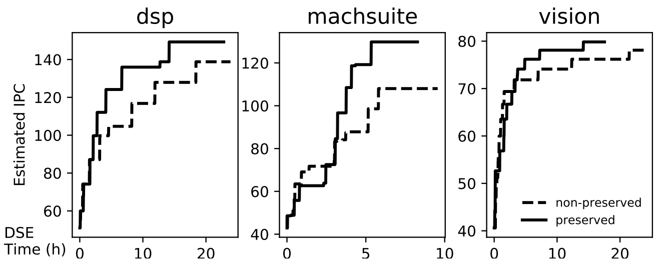


Figure 20: The effects of schedule-preserving transforms.

IX. Related work

Overlay Architectures: We highlight significant and recent overlay approaches; Li et al. provide an in-depth survey [81].

Soft CPU: FPGA vendors provide soft processor implementations, e.g. Xilinx MicroBlaze [82] and Intel Nios II [83], and there are also many open source works, e.g. SPREE [84], iDEA [85,86], and OpenRISC [87]. Some alternatives provide higher-performance microarchitectures, such as multi-issue (e.g. Leon3 [88], FPGA-Nehalem [26]), multi-thread (e.g. Octavo [24], CUSTARD [89], MT-MB [90]), multicore with scalable networks (e.g. Heracles [23], Kumar et al. [91]), vector operations (e.g. SIMD-Octavo [92], MXP [25]), as well as VLIW (e.g. TILT [93]).

Soft GPGPU: FlexGrip [30] and MIAOW [94] are single compute-unit (CU) overlays based on Nvidia and AMD GPU architectures, respectively. FGPU [28] was able to synthesize multiple CUs on a single FPGA board, with a follow-up work specialized for persistent deep learning [35].

[3]`gemm` is a tiled (blocked) implementation of matrix multiply, `mm` is not

Reconfigurable Architectures: QUKU [95] is an early example of a 2D-mesh style CGRA overlay. reMORPH is another 2D mesh-based overlay that is built around the FPGA's DSP blocks as primitives [31]. VDR [96] is a CGRA overlay which can map short program traces for JIT-based compilation. The DySER heterogeneous core/CGRA architecture was also mapped to FPGA [97,98]. ZUMA is an example of an FPGA-on-FPGA overlay [99].

Customizable Overlays: Interestingly, some overlays allow architecture customization. For example, CREMA [100,101] and Quickdough [102,103] leverage templates to customize PEs for each application and speedup the design process. CGRA-ME [104–107] and AHA [44,45,108] further introduced architecture description languages for arbitrary topologies and DSE with CGRA mapper involvement. Mocarabe [109] introduces the communication cost as a first-class citizen in the compiler to obtain a design with high frequency while still meeting the targeted II. SCRATCH [29] is a GPU-based overlay based on MIAOW [94], which automatically identifies the application-specific demands regarding the instruction set and computing unit capability, and generates a trimmed down GPU design.

Key Difference to prior Overlays: As compared to these prior frameworks, our overlay-synthesis approach attempts to perform application specialization automatically and across many aspects of the overlay architecture (instructions/topology/execution model/provisioning).

FPGA Programming: While the overlay approach improves the programmability by providing another layer of abstraction, there are also efforts to directly tackle this problem with new programming languages with lower-level abstractions. As an example, Dahila [110] generates predictable HLS designs by incorporating time-sensitive affine types into the language. On the other hand, Reticle [111] proposes an intermediate representation and low-level assembly that explicitly expresses special resources on FPGAs, e.g. LUTs and DSPs. Spatial [112] is a language designed for implementing accelerators based on parallel patterns. Although these techniques improve the programmability, they do not tackle reconfiguration overheads. Just-in-time compilation frameworks can also reduce the burden of FPGA synthesis [113,114].

A recent approach integrates separate compilation into an FPGA design flow to enable better usability [115,116]. These works leverage faster compilation/reconfiguration to subregions of the FPGA, and enable linking through a packet-switched network. The RapidStream framework [117–119] also partitions a large design for parallel implementation and final re-assembly, but instead uses customized point-to-point and pipelined channels to address the high area and limited bandwidth of packet-switched NoC's in prior work.

X. Conclusion

While FPGAs have proven to be extremely effective computational accelerators, their usability is not ideal. The heart of the problem is the limited design space of existing HLS tools, which is inflexible and requires frequent re-synthesis. In this work, we develop and evaluate the idea of an alternate HLS paradigm where a highly-flexible overlay is the target architecture. Surprisingly, even though the generated designs are programmable, the overall performance is on-par with state-of-the-art HLS tools.

Yet there is much more to be explored, and OverGen should be seen as a proof-of-concept for the potential of multicore spatial overlays. Many aspects of the design space can be further specialized to the chosen applications, leveraging the extreme flexibility of FPGAs. Examples include the NoC topology [120], NoC protocol [121,122], cache policies [123], coherence protocol [124], and synchronization [125] to name a few. One broad, underexplored aspect is *heterogeneity*: including heterogeneous cores, caches, networks, and memories. While our current framework assumes pure single-program parallelization, real systems (e.g. mobile SoCs [126], datacenters, VR [127] and even brain computer interfaces [128]) often require heterogeneous mixes of workloads with different throughput and latency requirements on the same fabric — this opens up vast potential for these different forms of architecture and microarchitecture heterogeneity. Supporting heterogeneity is challenging both because it adds another dimension to design-space exploration, and because it requires novel system support in virtualization and runtime management of heterogeneous resources.

Overall, we see spatial overlay synthesis as a potentially disruptive approach for FPGA HLS, and OverGen as springboard for future spatial architecture research.

Acknowledgment

This work was supported by NSF awards CCF-1751400 and CCF-1937599, as well as gift funding from VMware.

References

[1] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on fpgas," in *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 63–74.

[2] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A high memory bandwidth FPGA accelerator for sparse matrix-vector multiplication," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2014, pp. 36–43.

[3] Y. Wang, J. C. Hoe, and E. Nurvitadhi, "Processor assisted worklist scheduling for FPGA accelerated graph processing on a shared-memory platform," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 136–144.

[4] S. Zhou, R. Kannan, V. K. Prasanna, G. Seetharaman, and Q. Wu, "Hitgraph: High-throughput graph processing framework on FPGA," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 10, pp. 2249–2264, 2019.

[5] S. Salamat and T. Rosing, "FPGA acceleration of sequence alignment: a survey," *arXiv preprint arXiv:2002.02394*, 2020.

[6] W. Qiao, J. Oh, L. Guo, M.-C. F. Chang, and J. Cong, "FANS: FPGA-accelerated near-storage sorting," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2021, pp. 106–114.

[7] S. U. Hussain, B. D. Rouhani, M. Ghasemzadeh, and F. Koushanfar, "MAXelerator: FPGA accelerator for privacy preserving multiply-accumulate (mac) on cloud servers," in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC '18. New York, NY, USA: Association for Computing Machinery, 2018.

[8] Y. Chen, J. He, X. Zhang, C. Hao, and D. Chen, "Cloud-DNN: An open framework for mapping DNN models to cloud FPGAs," in *Proceedings of the 2019 ACM/SIGDA international symposium on field-programmable gate arrays*, 2019, pp. 73–82.

[9] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 16–25.

[10] N. Samardzic, W. Qiao, V. Aggarwal, M.-C. F. Chang, and J. Cong, "Bonsai: High-performance adaptive merge tree sorting," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20. IEEE Press, 2020, p. 282–294.

[11] A. Sohrabizadeh, J. Wang, and J. Cong, "End-to-end optimization of deep learning applications," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 133–139.

[12] J. Wang, L. Guo, and J. Cong, "AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 93–104.

[13] L. Song, Y. Chi, A. Sohrabizadeh, Y.-k. Choi, J. Lau, and J. Cong, "Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 65–77.

[14] L. Song, Y. Chi, L. Guo, and J. Cong, "Serpens: A high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication," in *Proceedings of the 59th Annual Design Automation Conference (DAC)*, 2022.

[15] Y. Hu, Y. Du, E. Ustun, and Z. Zhang, "GraphLily: Accelerating graph linear algebra on HBM-equipped FPGAs," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021, pp. 1–9.

[16] L. Guo, J. Lau, Z. Ruan, P. Wei, and J. Cong, "Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between FPGA and GPU," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 127–135.

[17] Y. Chi, L. Guo, and J. Cong, "Accelerating SSSP for power-law graphs," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2022, pp. 190–200.

[18] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceeding of the 41st Annual International Symposium on Computer Architecute*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 13–24.

[19] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman *et al.*, "Serving DNNs in real time at datacenter scale with project brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.

[20] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[21] A. Sohrabizadeh, C. H. Yu, M. Gao, and J. Cong, "AutoDSE: Enabling software programmers to design efficient fpga accelerators," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 27, no. 4, feb 2022.

[22] W. Qiao, J. Oh, L. Guo, M.-C. F. Chang, and J. Cong, "Fans: Fpga-accelerated near-storage sorting," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021, pp. 106–114.

[23] M. A. Kinsy, M. Pellauer, and S. Devadas, "Heracles: Fully synthesizable parameterized MIPS-based multicore system," in *2011 21st International Conference on Field Programmable Logic and Applications*, 2011, pp. 356–362.

[24] C. E. LaForest and J. G. Steffan, "OCTAVO: An FPGA-centric processor family," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 219–228.

[25] A. Severance and G. G. Lemieux, "Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor," in *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2013, pp. 1–10.

[26] G. Schelle, J. Collins, E. Schuchman, P. Wang, X. Zou, G. Chinya, R. Plate, T. Mattner, F. Olbrich, P. Hammarlund, R. Singhal, J. Brayton, S. Steibl, and H. Wang, "Intel Nehalem processor core made FPGA synthesizable," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 3–12.

[27] J. Kingyens and J. G. Steffan, "The potential for a GPU-like overlay architecture for FPGAs," *International Journal of Reconfigurable Computing*, vol. 2011, 2011.

[28] M. Al Kadi, B. Janssen, and M. Huebner, "FGPU: An SIMT-architecture for FPGAs," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 254–263.

[29] P. Duarte, P. Tomas, and G. Falcao, "SCRATCH: An end-to-end application-aware soft-GPGPU architecture and trimming tool," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 165–177.

[30] K. Andryc, M. Merchant, and R. Tessier, "FlexGrip: A soft GP-GPU for FPGAs," in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 230–237.

[31] K. Paul, C. Dash, and M. S. Moghaddam, "reMORPH: a runtime reconfigurable architecture," in *2012 15th Euromicro Conference on Digital System Design*. IEEE, 2012, pp. 26–33.

[32] R. Ben Abdelhamid, Y. Yamaguchi, and T. Boku, "MITRACA: A next-gen heterogeneous architecture," in *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, 2019, pp. 304–311.

[33] R. B. Abdelhamid, Y. Yamaguchi, and T. Boku, "A highly-efficient and tightly-connected many-core overlay architecture," *IEEE Access*, vol. 9, pp. 65 277–65 292, 2021.

[34] S. A. Chin, K. P. Niu, M. Walker, S. Yin, A. Mertens, J. Lee, and J. H. Anderson, "Architecture exploration of standard-cell and FPGA-overlay CGRAs using the open-source CGRA-ME framework," in *Proceedings of the 2018 International Symposium on Physical Design*, ser. ISPD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 48–55.

[35] R. Ma, J.-C. Hsu, T. Tan, E. Nurvitadhi, D. Sheffield, R. Pelt, M. Langhammer, J. Sim, A. Dasu, and D. Chiou, "Specializing FGPU for persistent deep learning," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 326–333.

[36] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton *et al.*, "Chipyard: Integrated design, simulation, and implementation framework for custom SoCs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[37] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, and D. Wentzlaff, "Openpiton: An open source manycore research framework," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 217–232.

[38] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "DSAGEN: Synthesizing programmable spatial accelerators," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 268–281.

[39] C. Tan, C. Xie, A. Li, K. J. Barker, and A. Tumeo, "Aurora: Automated refinement of coarse-grained reconfigurable accelerators," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1388–1393.

[40] M. Willsey, V. T. Lee, A. Cheung, R. Bodík, and L. Ceze, "Iterative search for reconfigurable accelerator blocks with a compiler in the loop," *IEEE TCAD*, vol. 38, no. 3, pp. 407–418, 2018.

[41] T. K. Bandara, D. Wijerathne, T. Mitra, and L.-S. Peh, "REVAMP: A systematic framework for heterogeneous CGRA realization," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 918–932.

[42] J. Cong, H. Huang, C. Ma, B. Xiao, and P. Zhou, "A fully pipelined and dynamically composable architecture of CGRA," in *22th FCCM*, 2014.

[43] C. Tan, T. Tambe, J. J. Zhang, B. Fang, T. Geng, G.-Y. Wei, D. Brooks, A. Tumeo, G. Gopalakrishnan, and A. Li, "ASAP: automatic synthesis of area-efficient and precision-aware CGRAs," in *Proceedings of the 36th ACM International Conference on Supercomputing*, ser. ICS '22, 2022.

[44] K. Koul, J. Melchert, K. Sreedhar, L. Truong, G. Nyengele, K. Zhang, Q. Liu, J. Setter, P.-H. Chen, Y. Mei, M. Strange, R. Daly, C. Donovick, A. Carsello, T. Kong, K. Feng, D. Huff, A. Nayak, R. Setaluri, J. Thomas, N. Bhagdikar, D. Durst, Z. Myers, N. Tsiskaridze, S. Richardson, R. Bahr, K. Fatahalian, P. Hanrahan, C. Barrett, M. Horowitz, C. Torng, F. Kjolstad, and P. Raina, "Aha: An agile approach to the design of coarse-grained reconfigurable accelerators and compilers," *ACM Trans. Embed. Comput. Syst.*, apr 2022.

[45] R. Bahr, C. Barrett, N. Bhagdikar, A. Carsello, R. Daly, C. Donovick, D. Durst, K. Fatahalian, K. Feng, P. Hanrahan, T. Hofstee, M. Horowitz, D. Huff, F. Kjolstad, T. Kong, Q. Liu, M. Mann, J. Melchert, A. Nayak, A. Niemetz, G. Nyengele, P. Raina, S. Richardson, R. Setaluri, J. Setter, K. Sreedhar, M. Strange, J. Thomas, C. Torng, L. Truong, N. Tsiskaridze, and K. Zhang, "Creating an agile hardware design flow," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[46] A. Sharifian, R. Hojabr, N. Rahimi, S. Liu, A. Guha, T. Nowatzki, and A. Shriraman, "μir -an intermediate representation for transforming and optimizing the microarchitecture of application accelerators," in *52nd MICRO*, 2019.

[47] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *44th ISCA*, 2017.

[48] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," *SIGPLAN Not.*, vol. 53, no. 2, pp. 461–475, Mar. 2018.

[49] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel paterns," in *44th ISCA*, 2017.

[50] V. Dadu, J. Weng, S. Liu, and T. Nowatzki, "Towards general purpose acceleration by exploiting common data-dependence forms," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 924–939.

[51] Z. Wang and T. Nowatzki, "Stream-based memory access specialization for general purpose processors," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 736–749.

[52] Z. Wang, C. Liu, and T. Nowatzki, "Infinity Stream: enabling transparent and automated in-memory computing," *IEEE Computer Architecture Letters*, 2022.

[53] Z. Wang, J. Weng, J. Lowe-Power, J. Gaur, and T. Nowatzki, "Stream Floating: Enabling proactive and decentralized cache optimizations," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 640–653.

[54] Z. Wang, J. Weng, S. Liu, and T. Nowatzki, "Near-Stream Computing: General and transparent near-cache acceleration," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 331–345.

[55] V. Dadu, S. Liu, and T. Nowatzki, "PolyGraph: Exposing the value of flexibility for graph processing accelerators," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 595–608.

[56] J. Weng, S. Liu, Z. Wang, V. Dadu, and T. Nowatzki, "A hybrid systolic-dataflow architecture for inductive matrix algorithms," in *HPCA*, 2019.

[57] J. Weng, S. Liu, D. Kupsh, and T. Nowatzki, "Unifying spatial accelerator compilation with idiomatic and modular transformations," *IEEE Micro*, pp. 1–12, 2022.

[58] V. Govindaraju, T. Nowatzki, and K. Sankaralingam, "Breaking SIMD shackles with an exposed flexible microarchitecture and the access execute PDG," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 341–352.

[59] T. Nowatzki, N. Ardalani, K. Sankaralingam, and J. Weng, "Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign," in *27th PACT*, 2018.

[60] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a Scala embedded language," in *49th DAC*, 2012.

[61] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016.

[62] G. Zhang, K. Zhao, B. Wu, Y. Sun, L. Sun, and F. Liang, "A RISC-V based hardware accelerator designed for Yolo object detection system," in *2019 IEEE International Conference of Intelligent Applied Systems on Engineering (ICIASE)*. IEEE, 2019, pp. 9–11.

[63] H. Cook, W. Terpstra, and Y. Lee, "Diplomatic design patterns: A TileLink case study," in *1st Workshop on Computer Architecture Research with RISC-V*, 2017.

[64] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The IBM system/360 model 91: machine philosophy and instruction-handling," *IBM J. Res. Dev.*, vol. 11, no. 1, pp. 8–24, 1967.

[65] L. Guo, Y. Chi, J. Wang, J. Lau, W. Qiao, E. Ustun, Z. Zhang, and J. Cong, "AutoBridge: Coupling coarse-grained floorplanning and pipelining for high-frequency HLS design on multi-die FPGAs," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 81–92.

[66] Q. M. Nguyen and D. Sanchez, "Fifer: Practical acceleration of irregular applications on reconfigurable architectures," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21, 2021.

[67] V. Dadu and T. Nowatzki, "Taskstream: Accelerating task-parallel workloads by recovering program structure," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22, 2022.

[68] V. Dadu, S. Liu, and T. Nowatzki, "Systematically understanding graph accelerator dimensions and the value of hardware flexibility," *IEEE Micro*, vol. 42, no. 4, pp. 87–96, 2022.

[69] C. Tan, T. Geng, C. Xie, N. B. Agostini, J. Li, A. Li, K. Barker, and A. Tumeo, "DynPaC: coarse-grained, dynamic, and partially reconfigurable array for streaming applications," in *2021 IEEE 39th International Conference on Computer Design (ICCD)*, 2021.

[70] E. Mirsky, A. DeHon *et al.*, "MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources." in *FCCM*, vol. 96, 1996, pp. 17–19.

[71] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, “ADRES: an architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix,” in *FPL*, 2003.

[72] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer, “Triggered Instructions: a control paradigm for spatially-programmed architectures,” in *40th ISCA*, 2013.

[73] V. Dadu, J. Weng, S. Liu, and T. Nowatzki, “Towards general-purpose acceleration: Finding structure in irregularity,” *IEEE Micro*, vol. 40, no. 3, pp. 37–46, 2020.

[74] G. Gobieski, S. Ghosh, M. Heule, T. Mowry, T. Nowatzki, N. Beckmann, and B. Lucia, “RipTide: a programmable, energy-minimal dataflow compiler and architecture,” in *MICRO-55: 55th Annual IEEE/ACM International Symposium on Microarchitecture*, 2022.

[75] M. Karunaratne, D. Wijerathne, T. Mitra, and L.-S. Peh, “4D-CGRA: introducing branch dimension to spatio-temporal application mapping on CGRAs,” in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019.

[76] J. Cong, H. Huang, C. Liu, and Y. Zou, “A reuse-aware prefetching scheme for scratchpad memory,” in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2011, pp. 960–965.

[77] B. Reagen, R. Adolf, Y. S. Shao, G. Wei, and D. Brooks, “MachSuite: benchmarks for accelerator design and customized architectures,” in *IISWC*, Oct 2014.

[78] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, “Programming heterogeneous systems from an image processing DSL,” *ACM Transactions on Architecture and Code Optimization*, vol. 14, no. 3, p. 1–25, Aug 2017.

[79] K. Sankaralingam, T. Nowatzki, V. Gangadhar, P. Shah, M. Davies, W. Galliher, Z. Guo, J. Khare, D. Vijay, P. Palamuttam, M. Punde, A. Tan, V. Thiruvengadam, R. Wang, and S. Xu, “The Mozart reuse exposed dataflow processor for AI and beyond: Industrial product,” in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA ’22, 2022, p. 978–992.

[80] J. Weng, A. Jain, J. Wang, L. Wang, Y. Wang, and T. Nowatzki, “UNIT: Unifying tensorized instruction compilation,” in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 77–89.

[81] X. Li and D. L. Maskell, “Time-multiplexed FPGA overlay architectures: A survey,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 24, no. 5, jul 2019.

[82] “MicroBlaze processor reference guide.”

[83] “Nios II processor reference guide.”

[84] P. Yiannacouras, J. Rose, and J. G. Steffan, “The microarchitecture of FPGA-based soft processors,” in *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES ’05. New York, NY, USA: Association for Computing Machinery, 2005, p. 202–212.

[85] H. Y. Cheah, S. A. Fahmy, and D. L. Maskell, “iDEA: a DSP block based FPGA soft processor,” in *2012 International Conference on Field-Programmable Technology*, 2012, pp. 151–158.

[86] H. Y. Cheah, F. Brosser, S. A. Fahmy, and D. L. Maskell, “The IDEA DSP block-based soft processor for FPGAs,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 3, Sep. 2014.

[87] Y. Bin and K. Ryoo, “Openrisc core-based soc platform design and verification,” in *ITC-CSCC: International Technical Conference on Circuits Systems, Computers and Communications*, 2007, pp. 276–277.

[88] J. Gaisler and M. Isomäki, “Leon3 gr-xc3s-1500 template design,” *Copyright Gaisler Research*, pp. 1–153, 2006.

[89] R. Dimond, O. Mencer, and W. Luk, “CUSTARD - a customisable threaded FPGA soft processor and tools,” in *International Conference on Field Programmable Logic and Applications, 2005.*, 2005, pp. 1–6.

[90] R. Moussali, N. Ghanem, and M. A. R. Saghir, “Supporting multithreading in configurable soft processor cores,” in *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES ’07. New York, NY, USA: Association for Computing Machinery, 2007, p. 155–159.

[91] C. Kumar H B, P. Ravi, G. Modi, and N. Kapre, “120-Core MicroAptiv MIPS overlay for the Terasic DE5-NET FPGA board,” in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 141–146.

[92] C. E. Laforest and J. H. Anderson, “Microarchitectural comparison of the MXP and Octavo soft-processor FPGA overlays,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 10, no. 3, May 2017.

[93] R. Rashid, J. G. Steffan, and V. Betz, “Comparing performance, productivity and scalability of the TILT overlay processor to OpenCL HLS,” in *2014 International Conference on Field-Programmable Technology (FPT)*, 2014, pp. 20–27.

[94] R. Balasubramanian, V. Gangadhar, Z. Guo, C.-H. Ho, C. Joseph, J. Menon, M. P. Drumond, R. Paul, S. Prasad, P. Valathol, and K. Sankaralingam, “Enabling GP-GPU low-level hardware explorations with MIAOW: An open-source RTL implementation of a GP-GPU,” *ACM Trans. Archit. Code Optim.*, vol. 12, no. 2, Jun. 2015.

[95] S. Shukla, N. W. Bergmann, and J. Becker, “QUKU: A coarse grained paradigm for FPGAs,” in *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.

[96] D. Capalija and T. S. Abdelrahman, "Towards synthesis-free JIT compilation to commodity FPGAs," in *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2011, pp. 202–205.

[97] J. Benson, R. Cofell, C. Frericks, C.-H. Ho, V. Govindaraju, T. Nowatzki, and K. Sankaralingam, "Design, integration and implementation of the DySER hardware accelerator into OpenSPARC," in *IEEE International Symposium on High-Performance Comp Architecture*, 2012, pp. 1–12.

[98] C.-H. Hoy, V. Govindarajuz, T. Nowatzki, R. Nagaraju, Z. Marzec, P. Agarwal, C. Frericks, R. Cofell, and K. Sankaralingam, "Performance evaluation of a DySER FPGA prototype system spanning the compiler, microarchitecture, and hardware implementation," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2015, pp. 203–214.

[99] A. Brant and G. G. Lemieux, "ZUMA: An open FPGA overlay architecture," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, 2012, pp. 93–96.

[100] F. Garzia, W. Hussain, and J. Nurmi, "CREMA: A coarse-grain reconfigurable array with mapping adaptiveness," in *2009 International Conference on Field Programmable Logic and Applications*. IEEE, 2009, pp. 708–712.

[101] W. Hussain, T. Ahonen, and J. Nurmi, "Effects of scaling a coarse-grain reconfigurable array on power and energy consumption," in *2012 International Symposium on System on Chip (SoC)*. IEEE, 2012, pp. 1–5.

[102] C. Liu, H.-C. Ng, and H. K.-H. So, "Automatic nested loop acceleration on FPGAs using soft CGRA overlay," *arXiv preprint arXiv:1509.00042*, 2015.

[103] A. K. Jain, D. L. Maskell, and S. A. Fahmy, "Are coarse-grained overlays ready for general purpose application acceleration on FPGAs?" in *2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*. IEEE, 2016, pp. 586–593.

[104] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson, "CGRA-ME: a unified framework for CGRA modelling and exploration," in *28th ASAP*, July 2017.

[105] K. Niu and J. H. Anderson, "Compact area and performance modelling for CGRA architecture evaluation," in *FPT*, Dec 2018.

[106] S. A. Chin and J. H. Anderson, "An architecture-agnostic integer linear programming approach to CGRA mapping," in *55th DAC*, 2018.

[107] M. J. Walker and J. H. Anderson, "Generic connectivity-based CGRA mapping via integer linear programming," in *27th FCCM*, 2019.

[108] J. Melchert, K. Feng, C. Donovick, R. Daly, C. W. Barrett, M. Horowitz, P. Hanrahan, and P. Raina, "Automated design space exploration of CGRA processing element architectures using frequent subgraph analysis," *CoRR*, 2021.

[109] F. Tombs, A. Mellat, and N. Kapre, "Mocarabe: High-performance time-multiplexed overlays for FPGAs," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021, pp. 115–123.

[110] R. Nigam, S. Atapattu, S. Thomas, Z. Li, T. Bauer, Y. Ye, A. Koti, A. Sampson, and Z. Zhang, "Predictable accelerator design with time-sensitive affine types," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 393–407.

[111] L. Vega, J. McMahan, A. Sampson, D. Grossman, and L. Ceze, "Reticle: A virtual machine for programming modern FPGAs," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 756–771.

[112] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun, "Spatial: A language and compiler for application accelerators," in *PLDI*, 2018.

[113] E. Schkufza, M. Wei, and C. J. Rossbach, "Just-in-time compilation for verilog: A new technique for improving the FPGA programming experience," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 271–286.

[114] A. Becker, S. Sirowy, and F. Vahid, "Just-in-time compilation for FPGA processor cores," in *2011 Electronic System Level Synthesis Conference (ESLsyn)*, 2011, pp. 1–6.

[115] D. Park, Y. Xiao, N. Magnezi, and A. DeHon, "Case for fast FPGA compilation using partial reconfiguration," in *28th International Conference on Field Programmable Logic and Applications, FPL 2018, Dublin, Ireland, August 27-31, 2018*. IEEE Computer Society, 2018, pp. 235–238.

[116] Y. Xiao, E. Micallef, A. Butt, M. Hofmann, M. Alston, M. Goldsmith, A. Merczynski-Hait, and A. DeHon, "PLD: fast FPGA compilation to make reconfigurable acceleration compatible with modern incremental refinement software development," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds. ACM, 2022, pp. 933–945.

[117] L. Guo, P. Maidee, Y. Zhou, C. Lavin, J. Wang, Y. Chi, W. Qiao, A. Kaviani, Z. Zhang, and J. Cong, "RapidStream: parallel physical implementation of FPGA HLS designs," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2022, pp. 1–12.

[118] L. Guo, Y. Chi, J. Wang, J. Lau, W. Qiao, E. Ustun, Z. Zhang, and J. Cong, "Autobridge: Coupling coarse-grained floorplanning and pipelining for high-frequency HLS design on multi-die FPGAs," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 81–92.

[119] Y. Chi, L. Guo, J. Lau, Y.-k. Choi, J. Wang, and J. Cong, "Extending high-level synthesis for task-parallel programs," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2021, pp. 204–213.

[120] H. Zheng, K. Wang, and A. Louri, "Adapt-NoC: A flexible network-on-chip design for heterogeneous manycore architectures," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 723–735.

[121] J. Cong, M. Gill, Y. Hao, G. Reinman, and B. Yuan, "On-chip interconnection network for accelerator-rich architectures," in *Proceedings of the 52nd Annual Design Automation Conference*, ser. DAC '15. New York, NY, USA: Association for Computing Machinery, 2015.

[122] J. Yin, P. Zhou, S. S. Sapatnekar, and A. Zhai, "Energy-efficient time-division multiplexed hybrid-switched NoC for heterogeneous multicore systems," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 293–303.

[123] T. A. Khan, D. Zhang, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci, "Ripple: Profile-guided instruction cache replacement for data center applications," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 734–747.

[124] J. Zuckerman, D. Giri, J. Kwon, P. Mantovani, and L. P. Carloni, "Cohmeleon: Learning-based orchestration of accelerator coherence in heterogeneous socs," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 350–365.

[125] C. Giannoula, N. Vijaykumar, N. Papadopoulou, V. Karakostas, I. Fernandez, J. Gómez-Luna, L. Orosa, N. Koziris, G. Goumas, and O. Mutlu, "Syncron: Efficient synchronization support for near-data-processing architectures," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 263–276.

[126] M. D. Hill and V. J. Reddi, "Accelerator-level parallelism," *Communications of the ACM*, vol. 64, no. 12, pp. 36–38, 2021.

[127] M. Huzaifa, R. Desai, S. Grayson, X. Jiang, Y. Jing, J. Lee, F. Lu, Y. Pang, J. Ravichandran, F. Sinclair, B. Tian, H. Yuan, J. Zhang, and S. V. Adve, "ILLIXR: enabling end-to-end extended reality research," in *IEEE International Symposium on Workload Characterization, IISWC 2021, Storrs, CT, USA, November 7-9, 2021*. IEEE, 2021, pp. 24–38.

[128] I. Karageorgos, K. Sriram, J. Veselý, M. Wu, M. Powell, D. Borton, R. Manohar, and A. Bhattacharjee, "Hardware-software co-design for brain-computer interfaces," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 391–404.