DSAGEN: Synthesizing Programmable Spatial Accelerators

Jian Weng‡*, Sihao Liu‡*, Vidushi Dadu*, Zhengrong Wang*, Preyas Shah†, Tony Nowatzki*

*University of California, Los Angeles †SimpleMachines, Inc.

{jian.weng, sihao, vidushi.dadu, seanzw, tjn}@cs.ucla.edu

Abstract—Domain-specific hardware accelerators can provide orders of magnitude speedup and energy efficiency over general purpose processors. However, they require extensive manual effort in hardware design and software stack development. Automated ASIC generation (eg. HLS) can be insufficient, because the hardware becomes inflexible. An ideal accelerator generation framework would be automatable, enable deep specialization to the domain, and maintain a uniform programming interface.

Our insight is that many prior accelerator architectures can be approximated by composing a small number of hardware primitives, specifically those from spatial architectures. With careful design, a compiler can understand how to use available primitives, with modular and composable transformations, to take advantage of the features of a given program. This suggests a paradigm where accelerators can be generated by searching within such a rich accelerator design space, guided by the affinity of input programs for hardware primitives and their interactions.

We use this approach to develop the DSAGEN framework, which automates the hardware/software co-design process for reconfigurable accelerators. For several existing accelerators, our evaluation demonstrates that the compiler can achieve 89% of the performance of manually tuned versions. For automated design space exploration, we target multiple sets of workloads which prior accelerators are design for; the generated hardware has mean $1.3 \times \text{perf}^2/\text{mm}^2$ over prior programmable accelerators.

I. INTRODUCTION

As a response to the slowing of technology scaling, specialized accelerators have proliferated in many settings and across a wide variety of domains. Three basic strategies have emerged for developing and using accelerators, each with their own benefits and limitations:

- **Automated Design (eg. HLS):** High-level synthesis compiles languages like C with pragmas to custom hardware [10]. While HLS is nearly automatic, the design space is generally limited, and designs are not programmable.

- **Domain Specific (eg. [1, 6, 20, 21, 25, 27, 31, 33, 34, 36, 38, 39, 42, 44, 46, 47, 50, 60, 74, 81, 82, 84, 86, 95, 96, 98, 99])**
  This approach customizes hardware for kernels within a domain and provides a domain-specific hardware/software interface. The advantages are high performance and sufficient flexibility, at the cost of hardware/software effort, which must be repeated as workloads evolve.

- **General Purpose (eg. GPU, DSP, SIMD Extensions):**
  This approach enables a uniform programming interface that has long term stability, but it lacks the degree of specialization provided by the above.

There is a large space of applications for which the above approaches is not satisfying: where some flexibility is needed (either because of algorithm diversity/change, or the need for sharing hardware), but the cost of a domain specific hardware design and software-stack implementation cannot be justified. For such settings, an ideal specialization approach would yield the level of automation provided by HLS, enable deep enough specialization to the domain, and maintain a uniform programming interface. An ideal approach would also enable the user to make a tradeoff between specialization efficiency and generality, by intelligently tuning the flexibility of the hardware and hardware/software interface.

A possible approach is to search within a flexible architecture design-space, guided by the computation and memory patterns present in the set of desired target programs. A central challenge is in defining this design space to be both broad enough while still enabling specialization. Based on the results of much prior work (eg. [20, 45, 65, 69, 78]), we believe that decoupled spatial accelerators are a promising candidate. *Spatial* refers to designs with hardware/software interfaces that expose underlying hardware features. *Decoupled* refers to the separation of memory access and computation pipelines.

These architectures are attractive first because they are efficient – they expose low-enough level hardware details to take advantage, and also enable specialization of memory access. More importantly, it is possible to define a set of decoupled-spatial primitives which have semantics understandable by a compiler, so that they can be flexibly composed. Their modular nature also means a unified hardware/software interface with modest programmer hints can be developed.

Our goal is to develop the principles and usable framework for this problem, which we refer to as *programmable accelerator synthesis*. This involves taking as input a set of kernels (eg. written in C with minimal programmer annotations) defining the desired functionality, and serving as a proxy for generality. The outputs are 1. a synthesizable hardware artifact which is specialized to the nature of input applications, 2. a customized hardware/software interface, and 3. (logically) a compiler which can target the given design.

---

†Jian Weng and Sihao Liu are co-first authors.
‡Note these are all from a single conference, MICRO 2019.
In this work we develop DSAGEN\textsuperscript{2}, the decoupled spatial architecture generator\textsuperscript{3}, depicted in Figure 1. First, the architecture is represented as a graph – the architecture description graph (ADG) – composed of primitive components, like processing elements and switches, with flexible connectivity. The framework may be used either for normal compilation, where the ADG represents an instance of a hardware unit, or design-space exploration (DSE), where the ADG is synthesized through iterative refinement.

For either use case, the compiler first transforms each input kernel into a decoupled dataflow representation; several different versions of each kernel are created with different sets of transformations, each targeted to particular architecture features. Then the hardware mapper will distribute the program to hardware resources, and evaluate an efficiency metric (eg. performance\times power\times area) using a model. The best legal version of the compiled program is chosen. During design space exploration (DSE), the history of hardware mappings is used to modify the ADG to improve efficiency, and this repeats until convergence.

**Challenge and Approach:** The problem of programmable accelerator synthesis opens several key challenges:

**Sufficient Design Space:** The design space could hypothetically be defined as a template architecture with a few parameters. We instead take the more extreme view that there is value in 1. allowing choice of features at the ISA level, 2. allowing irregular connectivity between components. We believe both are necessary to reach closer to the specialization provided by a domain-specific architecture (DSA).

**Compilation for Modular Features:** Compiling for modular features is challenging because of the different transformations necessary for each feature. Our approach is to develop a set of transformations targeting various hardware features, and consider multiple combinations of these for mapping to hardware (eliminating those not required for the given ADG).

**Intelligent Design-space Exploration:** At each step of DSE, the framework must decide how to manipulate the ADG to improve hardware efficiency. This is challenging given the vast design space and slow nature of spatial architecture compilers. We propose a codesign algorithm which 1. leverages an application-aware performance, area, and power model for quick evaluation of the objective function, and 2. integrates a novel solution-repairing spatial-architecture scheduler into DSE to avoid redundant compilation.

**Hardware Generation:** With an arbitrary topology, generating a path to configure each hardware unit is non-trivial, and it is also important as it will determine the time to switch between program phases. We develop an architecture-aware simulated annealing-based approach.

**Key Results:** According to our evaluation, hardware primitives are expressive enough to approximate many state-of-the-art decoupled spatial accelerators. Our compiler is able to generate code with mean $1.12\times$ execution time of the manual version across several different domains, including those with control and memory irregularity. We also demonstrate the capability of our hardware software co-design algorithm, which is able to achieve a balance between performance, and area and power cost. Our contributions are:

- Recognizing a set of hardware primitives that compose a rich design space for spatial accelerators.
- Automated flow for software-hardware codesign, with:
  - Modular compilation for composable h/w primitives.
  - Solution-repairing spatial-scheduling techniques.
  - Configuration generation for irregular spatial topologies.

**Paper Organization:** We first discuss the background of decoupled spatial architectures in Section II. Then we discuss our formulation of their hardware primitives in Section III. The compiler is discussed in Section IV, DSE in Section V, and hardware generation in Section VI. Methodology and evaluation is in Section VII and VIII. Finally, we discuss the related work in Section IX.

**II. DECOUPLED SPATIAL ARCHITECTURE BACKGROUND**

Decoupled spatial architectures (eg. [17, 20, 45, 65, 92]) have shown capability to attain high performance with low power/area overhead while retaining programmability. A decoupled spatial architecture is defined by two characteristics: 1. decoupled in the sense that there are customized pipelines

---

\textsuperscript{2}Open-source repository: https://github.com/PolyArch/dsa-framework

\textsuperscript{3}In some sense, DSAGEN can generate domain specific architectures.
for handling memory access and computation\(^4\), and 2. spatial in the sense that aspects of low-level hardware execution, like the network and scheduling of operations, is exposed through the hardware/software interface.

Mapping applications to decoupled-spatial hardware involves: 1. decoupling data access from computation operations, and mapping memory access to corresponding units; 2. mapping computation onto processing elements, and communication to the on-chip network.

As a concrete example consider Figure 2(a), which is a vector dot product. In Figure 2(b), the program is represented as a decoupled dataflow graph [65], where memory accesses are represented as coarse-grain streams, and computations are dataflow graphs (in the example, the computation is “unrolled” by two iterations). This decoupled dataflow form eliminates the implicit memory ordering constraints of the original program – disambiguated memory streams are simpler to map to decoupled memories.

The spatial architecture shown in Figure 2(c) is composed of memories, processing elements, switches, and ports (synchronization elements). We will later refine these components into modular primitives. Figure 2(c) also shows the mapping of the program to the hardware substrate, where operations are mapped onto the PEs, and all of the instruction dependences are mapped onto the on-chip network (as it is shown in Figure 2(c)). Each memory engine generates memory requests – here array \( a \) and \( b \) are mapped to separate memories, and processing elements operate on data as it arrives.

Comparing with the conventional Von-Neumann model, the distributed nature of the PEs enables high concurrency without the overheads multi-threading. Wide memory-access can feed many PEs. Instruction dispatch overhead is amortized by spatial fabric configuration.

### III. Decoupled Spatial Design Space

Here we overview our graph-based approach to hardware description for our design space, and describe a set of modular hardware primitives and their parameters. We then give a few examples of architectures expressible within the design space.

\(^4\)Specifically, the request initiator and response receiver are decoupled, also known as explicit-decoupled data orchestration (EDDO) [73].

---

**Fig. 2:** Example decoupled-stream program and h/w mapping.

---

**Fig. 3:** Modular Spatial Architecture Components.

---

**A. Decoupled Spatial Primitives**

Our approach is to develop a set of architecture primitives, which are simple enough to be composable, are parameterizable enough to yield substantial benefits in customization, and which have well-defined execution models that can be understood by a compiler. Once these primitives are determined, the overall architecture can be described as a graph – the architecture description graph (ADG).

Figure 3 describes the set of proposed spatial architecture primitives. The basic elements are processing elements (PEs) to perform computations, memories which provide abstractions for shared memory, switches and connections for forming the network, and a control unit to synchronize different phases of a program. Most components can specify a power-of-two datapath bitwidth. What follows is a detailed description of the design space.

**Execution Model Parameters:** One of the most important factors for determining the tradeoff between generality and efficiency is the “execution model” of the component: how does the unit decide what and when to perform an action. We allow parameterizability in two important dimensions:

**Dynamic vs Static Scheduling:** PEs and switches support either static or dynamic scheduling of instruction execution and routing. In static scheduling, the order of all operations and data arrivals is determined by the compiler, whereas in dynamic scheduling the operation is chosen dynamically based on data arrival. Dynamic scheduling requires more power/area to implement logic for checking operand readiness, which is
proportional to the instruction window of the PE. Moreover, it
needs flow-control on the network to balance the different rate
of incoming operands. Static scheduling loses this flexibility
but gains lower power/area overhead.

**Dedicated vs Shared:** Dedicated elements only support one
instruction or routing decision (e.g. PEs in a systolic array
or some coarse grain reconfigurable architectures [20, 26,
65]), whereas shared elements temporally multiplex
different static instructions or routing decisions (e.g. like some other
CGRAs [53, 54, 56, 62, 69, 85]). Shared elements are parameter-
ized by the number of instructions supported. Dedicated
PEs have higher throughput by avoiding contention, and have
lower power/area overhead because of the smaller size of
the instruction buffer. Shared PEs enable more instruction
concurrency at the cost of area/power.

**Processing Elements (PEs):** In addition the above, PEs may
specify a set of instructions which are to be supported.
Functional units (FUs) which support the required functions
will be selected during hardware generation. This includes
the use of *decomposable* FUs – FUs that can be decomposed into
smaller power-of-two functions (e.g. 64-bit adder into two 32-
bit adders). Dynamically scheduled PEs support stream-join
control [20], which enables them to conditionally re-use their
inputs or abstain from computation. This is useful to support
join operations in sparse linear algebra and database ops.

**Switches:** Central to this approach is the flexibility of the
switch, which can connect inputs and outputs with differing
bitwidths. A routing connectivity matrix describes which
inputs can connect to which outputs, down to the granularity of
bytes. A switch may optionally be *decomposable* down to a
certain bitwidth, which means that it can route power-of-two
finer-grain datatypes independently [20]. Switch complexity
is a key determinant in the tradeoff between complexity and
efficiency. Complex networks may be formed out of multiple
switches, and switches may choose not to flop their output so
that a compound routing stage may execute in a single cycle.

**Connections:** Direct communication between hardware
elements is specified with connections, and naturally these form
the edges of the ADG.

**Delay Elements:** Delay elements are essentially FIFOs used
for pipeline-balancing, parameterized by their depth. A deeper
delay element implies more area but helps the compiler meet
timing requirements [64]. Static-scheduled delay elements
offer a fixed delay, while dynamic scheduled delay elements
act as a buffer which is drained opportunistically.

**Synchronization Elements:** These units are the interface be-
tween dynamically scheduled elements (e.g. memory, dynamic
PEs) and static elements (static PEs). Their purpose is to
synchronize multiple inputs to a computation, to enable static
reasoning about the timing of all dependent events. They are
implemented as FIFO buffers, which may be configured to
fire (i.e. to be read and popped) simultaneously based on the
presence of data. They are coordinated by a programmable
ready-logic, which can be configured statically to allow dif-
ferent synchronization elements to be fired together.

**Memories:** The basic execution model of a memory is that
it arbitrates access from concurrent coarse grained memory
patterns, which we refer to as streams [15, 65]. The relative
ordering of streams can be synchronized with barriers.

Memories are parameterized by their capacity, width, and
number of concurrent streams. We presently support two fixed
candidate controllers, linear and indirect. The linear controller
is similar to the address generator in REVEL [92], which
is able to generate inductive 2d memory streams (eg. which
can enable triangular access patterns). The indirect generator
is similar to the controller in SPU [20], which can generate
indirect memory access (e.g. a[b[i]]). We also optionally
support atomic update operations by embedding compute units
within each bank (e.g. to support a[b[i]]+=1).

**Control:** Each of the above components accepts a control
input, which configures it for some coarse amount of work
(access patterns for memory and computation graphs for PEs and
switches). The control unit distributes work to other components,
and thus synchronizes all other units for each phase of the algorithm. In this work, we assume the control
unit is a programmable core with stream-dataflow [65] ISA to
encode commands for other units.

### B. Principles of Composition

One benefit of the ADG abstraction is the ability to cus-
tomize the datapath and memory features (and parameters)
for a set of related programs. Later, we discuss how the
compiler framework will attempt to take advantage of available
resources and topology.

We overview the basic principles and considerations for
composition. First, *statically scheduled PEs and switches* have
less hardware overhead than their dynamic counterparts, but
require that all inputs are available at a known time. The

---

**Fig. 4:** Example Architecture Description Graphs (ADGs).
synchronization element can provide such a guarantee by buffering and releasing data in a coordinated fashion.

Analogously, dedicated elements have less hardware overhead, as they do not store or arbitrate between multiple instructions. However, if the timing of input operands is not matched, there is no other work which can be performed, and the pipeline will become imbalanced. Indeed, the throughput loss will be proportional to this imbalance [64]. The delay element allows a configurable delay to aid the compiler to compensate to enable the use of dedicated elements efficiently.

Switches can connect PEs regardless of whether they use static/dynamic scheduling, or if they are dedicated or shared. The hardware generator will output the appropriate switch depending on the properties of the connectors. The compiler will then enforce that values do not flow from static to dynamic PEs (without going through sync. elements) or from dedicated to temporal PEs (due to overwhelming a temporal PE).

C. Design Space Capabilities & Limitations

Ability to Express Existing Architectures: Figure 4 shows example architecture description graphs (ADGs), composed from these primitives, for several existing architectures. These graphs demonstrate topological generality, which controls the ratio of datapath flexibility versus switch overhead: CCA(b) [16] has the fewest switches, but has only limited flexibility. Softbrain(d) [65] is the most flexible but has with highest overhead.

Beyond the topology, many existing accelerators can be approximated, primarily those that would be considered CGRAs. The bounds on the expressiveness are better explained by the limitations of the current ADG. There are several categories of limitations with DSAGEN’s design space, which include features which may be hypothetically added as well as fundamental limitations.

Potential Features:

- **Coalescing**: We could implement memory coalescing; irregular access is currently supported through banking.
- **Flexible Buses**: Another example is arbitrary buses in the topology. As of now, within the architecture network, buses are only between memories and synchronization elements, and on the outputs of processing elements.
- **Alternate Control Cores**: For designs that do not require programmability, we could replace the control core with much simpler FSMs or even a simple fixed stream RAM.
- **Heterogeneous Cores**: The ADG models a single instance of a decoupled-stream architecture (ie. one control core). We could add support for designs that connect multiple unique instances, and perhaps a custom inter-core network.

Fundamental Feature Limitations:

- **Memory Consistency**: First, DSAGEN does not provide support for maintaining strict sequential memory access semantics (only barrier-wise synchronization of memory access). Therefore, the compiler and/or programmer is responsible for maintaining correctness of memory ordering.
- **Speculation**: DSAGEN does not support speculative execution or general memory disambiguation.

Because of the above, designs like DySER [26], BERET [28], TRIPS [9], WaveScalar [88] and Tartan [55] would not be expressible.

Examples: To make the limitations concrete, we discuss a few examples of how DSAGEN could approximate several accelerators which we do not explore further in this work.

- **TABLE** [49] uses a hierarchical mesh of static-scheduled temporal PEs, each with their own scratchpad. We could approximate TABLE if we decouple the scratchpad control from the PE datapath control.
- **Plasticine** [70] first has scalar/vector FIFOs, serving a similar purpose to sync. elements: the datapath is composed of static-scheduled/dedicated PEs; its pattern memory unit (PMU) is a combination of datapath plus banked scratchpad; its PCU has no memory and a larger datapath. Nested fine-grain parallelism is supported by allowing dataflow graphs to communicate. As noted, we do not yet support memory coalescing for scratchpad.
- **Time-scheduled Plasticine** [97] can be similarly approximated but with temporal PEs.
- **Classic CGRAs** [2, 17, 24, 53, 54, 56, 62, 85], which have static-scheduled, shared-PEs, can be approximated, but we will necessarily employ decoupled memories.

Domain-specific Designs: Domain-specific designs can be approximated, provided their FUs operate on primitive datatypes (DSAGEN only supports power-of-two bitwidths). For example, DianNao [12] can be instantiated with two scratchpads and static-scheduled, dedicated PEs with a binary-tree interconnect. On the other hand, Q100 [98] is harder to approximate, as it uses non-primitive datatypes.

Clarification on our Goals: Though the ADG can approximate some existing architectures, we do not seek to create a general compiler that rivals prior domain-specific compilers, as this is a grand challenge problem. In the remainder of our work, our goal is to create a compiler and design space explorer that works as well as possible starting from a domain-agnostic program representation.

IV. MODULAR DECOUPLED SPATIAL COMPIlATION

A key challenge in building a DSE framework is compiling a single, domain-neutral program representation to a variety of hardware units, each with a unique combination of parameters and ISA features. Here we describe the compiler’s responsibilities, methods, and the modular compilation approach.

A. Compiler Overview

Compilation involves the following basic steps:

1) **Region Choice**: Decide which program regions to offload.
2) **Region Concurrency**: Which of these run concurrently.
3) **Analyze Memory**: Determine which accesses can be decoupled without violating program semantics.
4) **Translate to Dataflow**: Transform regions to dataflow IR.
5) **Apply Loop Transformations**: Apply generic transformations to each region for spatial/temporal locality.

6) **Apply Modular Transformations**: Apply transformations specific to the selected hardware features in the ADG.

7) **Schedule Computations**: Map resources of concurrent program regions to hardware resources.

8) **Code Generation**: Generate control code and spatial hardware configuration.

Ideally, all steps in the compilation would be fully automated, however, this is difficult considering the limitations of compiler analysis and programming languages. Instead, we rely on programmer help for some aspects, which we believe could also be accomplished by higher-level domain-specific language compilers (eg. TVM [11] or Tensorflow).

First, we assume that the programmer or framework can perform higher-level loop transformations to extract locality (eg. loop blocking). Second, we rely on additional information about memory aliasing in order to decouple memory accesses. Finally, we rely on help for choosing which program regions to consider offloading; this final aspect can be easily automated, whereas the first two are more difficult.

In the following subsections, we describe the programming interface, the basics of the compilation flow, generic transformations, and our approach of modular compilation with examples.

**B. Programming Interface**

A typical approach for programming an accelerator would be to use a domain-specific language (DSL). While this is appropriate for a fixed hardware and domain, our goal is to enable the maximum possible freedom to include different programming idioms. A general-purpose high-level programming interface serves this purpose better, and we choose to use C.

On the other hand, the semantics of such languages are not generally very rich, and we require some higher-level information like regions to be offloaded, and memory alias freedom. Therefore, this programming interface should expose such information without violating the original semantics. For this reason, we rely on some additional information about memory aliasing in order to decouple memory accesses.

In the following subsections, we describe the programming interface, the basics of the compilation flow, generic transformations, and our approach of modular compilation with examples.

**C. Compiler Transformation**

The basic compilation flow involves four steps: decoupling memory and computation, applying modular transformations, spatial scheduling, and code generation.

**Decoupling the Memory and Compute**: To decouple computation and memory operations, our compiler inspects code blocks marked with the offload pragma, and slices the memory operations (typically Load and Store in LLVM). Address computation will be analyzed by LLVM’s SCEV module, so that this information can later be used to hoist and encode these memory operations in stream intrinsics [65]. After slicing and decoupling address computations, the remaining operations are transformed into a dataflow representation.

**Data Dependence Transformation**: Mapping control flows onto the spatial accelerator requires transforming control dependences to data dependence, and we use a variant of the transformation to program dependence graphs [22]. Figure 6(a,b) shows that the original code and the control flow graph with two branches, and Figure 6(c) shows the transformed data dependence graph – both branches will be executed, and a selector will select the proper value according to the result of the comparison.

**Modular Compilation**: The compiler optimizes for the given ADG’s hardware features. Before performing any hardware-dependent transformations, the compiler will first inspect if the underlying hardware has the corresponding feature to support it. If not, we ensure that there is always a fallback that does not use this feature to guarantee the success of compilation.

For example, if in the program there is an indirect memory access (i.e. a[b[i]]), ideally, we want to encode this idiom in indirect stream intrinsics. However, if the underlying hardware is not capable, the analysis and transformation pass for this idiom will be skipped/disabled. In final code generation, the compiler will fall back to generating scalar operations for this memory access. In Section IV-E, we will discuss the technical details of these transformations.

**Spatial Scheduling**: There are three responsibilities of spatial scheduling [64, 66]: 1. map instructions and memory streams...
Unmap one or more mapped instructions (or streams);  
for each unmapped instruction (or stream); do  
  for each compatible PEs (or memory); do  
    Route this instruction’s operands and dependences to the network using Dijkstra’s algorithm;  
    Recompute the timing (min/max time of each instr.);  
    Compute the objective based on timing and mapping;  
  end  
  Commit to the PE which yields the highest objective.;  
end  
Stop if the objective converges or maximum iters. reached;  

Algorithm 1: Structure of scheduling algorithm iteration.

onto hardware units; 2. route dependences onto the network; 3. match the timing of operand arrival (for static components).

The first two responsibilities must be extended to support mixing PEs with different execution models. For example, data-dependent control flow can only be mapped to PEs with dynamic scheduling (or be transformed to predication), and instructions with low-rate computations (e.g., from an outer-loop) should favor shared PEs. Moreover, as discussed in Section III-B, the scheduler must enforce constraints when ADG components with different execution models communicate.

We adopt a stochastic search based algorithm, similar in spirit to prior FPGA [51] and CGRA schedulers [52, 64]: Each scheduling iteration attempts to improve the objective by remapping some instruction or stream (see Algorithm 1).

To avoid local minima during the search, the routing and PE resources are allowed to be overutilized, and the routing-algorithm and objective minimization together minimize overutilization. The objective is formulated as a weighted function which prioritizes minimizing the following: 1. overutilization of PEs and network, 2. maximum initiation interval of dedicated PEs, 3. latency of any recurrence paths. The algorithm completes when there is no overutilization and when the objective has converged (stable for several iterations).

Code Generation: The compiler goes through each candidate of each code transformation, and chooses one with the highest estimated performance (see Section V for more details on the estimation model) for code generation. The code generator removes the operations offloaded to the spatial architecture, encodes the decoupled data access_communication in controller intrinsics, and injects memory fences to enforce the semantics.

D. Generic Optimizations

Enforcing dependences by stalling the whole spatial architecture significantly harms the performance. We find two idioms are quite useful to avoid this.

Producer-Consumer: Consider the example shown in Figure 7(a), where a value v produced by the first offloaded region is consumed by the second. When it comes to this idiom, the compiler will generate control code that directly forwards the produced value to the consumer. This not only avoids the synchronization overhead introduced by waiting for the producer phase to be done, but also enables pipelining the producer and consumer regions.

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

(b) Repetitive Update

As shown in Figure 7(b), the array c[j] is undergoing a repetitive in-place update. The compiler first inspects the size of the data updated each time (in this case, this is m). Then the compiler compares this number with the capability (the size) of the on-chip synchronization buffers. If this data size fits in the buffer, the compiler routes data directly between producer and consumer (on the datapath) to avoid the unnecessary memory traffic and memory fences. Otherwise, the compiler will rewrite the update loop level by tiling it so that data size updated each time can fit in the capability of the synchronization buffers.

E. Modular Code Transformation

Here we discuss three key modular code transformations.

Resource Allocation: A simple example of a hardware feature which the compiler should be robust to is its size (in computation and memory bandwidth). This can be accomplished by choosing the degree of vectorization to match hardware capability. It may be unknown how much to vectorize each concurrent program region, as it depends on whether an efficient schedule exists on the ADG for that degree. Thus the degree of vectorization becomes a modular feature which the compiler explores.

Control-Dependent Memory Access: Control-dependent memory access is common in kernels which perform “joins”, for example merge sort, database join, and sparse tensor operations. Figure 8(a,b) gives an example program (sparse inner-product multiply) and its control-flow graph. A naive mapping of the program to a spatial architecture would preserve the data dependence from the control decision back to the pointer increment (backwards branch in Figure 8(b)). This introduces a long recurrence chain which limits the performance.

This can be avoided by decoupling the memory access and reusing inputs based on the control flow, as shown in the resulting decoupled dataflow in Figure 8(c) (this automates the stream-join transformation [20]). This transformation is only valid if the hardware supports dynamic scheduling, as the data-consumption is data-dependent. However, it is hard for the compiler to know whether this transformation will be successful, because until mapping the program to hardware, it is uncertain whether there exists enough PEs and switches which support dynamic scheduling in the right topology, which are not yet being consumed by other resources. Thus this is a feature the compiler explores whether to use.
The ADG at each iteration is not completely di-

v1[0:n1]

Computation Inst. ×

(c) Data Dependence Graph

iterative approach to codesign is as follows:

of hardware features based on iterative graph search. The basic
formations to each program along with a fine-grain selection
input programs and hardware, selecting the best set of trans-
performance of a spatial architecture can only be determined
of the objective function. This is due to the fact that the
force which is explored by the compiler.

However, even on hardware instances which support these
idioms, its difficult to tell how much memory bandwidth
will be available for any given stream until after scheduling
the program. Therefore, such transformations are a modular
feature which is explored by the compiler.

V. AUTOMATED DESIGN SPACE EXPLORATION

DSAGEN can perform an automated codesign between the
input programs and hardware, selecting the best set of trans-
formations to each program along with a fine-grain selection
of hardware features based on iterative graph search. The basic
iterative approach to codesign is as follows:

1) Start with some initial default ADG.
2) At each step:
   a) Create a modified ADG where a random number of
      components are added or removed (with random con-
      nectivity), without exceeding the power and area budget.
   b) Schedule all N input kernels to the spatial architecture,
      with M different versions of each kernel, corresponding
to different unrolling (ie. vectorization) factors.
   c) Estimate the performance of every version of each
      kernel, based on a performance model.
   d) Select the best performing version of each kernel, and
      estimate the objective function.
   e) If the objective improves, continue with the new ADG.
3) Repeat until the objective (eg. perf²/mm²) converges

The following subsections describe how we improve the
time-per-iteration through a novel spatial-scheduling repair
technique, as well as the performance and power/area models.

A. Fast DSE with Repairing Scheduler

Challenge: Length of DSE Step: The most time consuming
aspect of each iteration is evaluating the performance aspect
of the objective function. This is due to the fact that the
performance of a spatial architecture can only be determined
once the program is mapped to the hardware; the mapping
affects the memory and resource contention, as well as the
latency of any critical dependencies. Unfortunately, spatial
scheduling is known to be a lengthy algorithm, with practical
heuristics taking on the order of at least minutes for larger
problems [52,64,66,71,75], and sometimes even longer. What
exacerbates this problem further is that the compiler must
consider different sets of transformations due to its modular
approach, multiplying the overhead by some factor.

Insight: The ADG at each iteration is not completely dif-

ferent: perhaps some edges are added to increase network
connectivity, processing elements may be added to increase
performance, and a small subset of the above may be deleted
to improve area/power overheads. For many incrementally
changed ADGs, the previous schedule will still be valid. Even
when a hardware component which is used by a program
region is deleted, the remainder of the schedule is still valid.

Repairing Spatial Scheduler: Therefore, we propose a DSE-
approach with a repairing spatial scheduler. After each ADG

Indirect Memory Access: Indirect memory access and atomic
update is quite common in many critical workloads that have
irregular memory patterns, for example histogramming, graph
processing, and sparse matrix operations. If the underlying
hardware has support (ie. with indirect memory controller),
the compiler should vectorize these operations.

However, even on hardware instances which support these
idioms, its difficult to tell how much memory bandwidth
will be available for any given stream until after scheduling
the program. Therefore, such transformations are a modular
feature which is explored by the compiler.

V. AUTOMATED DESIGN SPACE EXPLORATION

DSAGEN can perform an automated codesign between the
input programs and hardware, selecting the best set of trans-
formations to each program along with a fine-grain selection
of hardware features based on iterative graph search. The basic
iterative approach to codesign is as follows:

1) Start with some initial default ADG.
2) At each step:
   a) Create a modified ADG where a random number of
      components are added or removed (with random con-
      nectivity), without exceeding the power and area budget.
   b) Schedule all N input kernels to the spatial architecture,
      with M different versions of each kernel, corresponding
to different unrolling (ie. vectorization) factors.
   c) Estimate the performance of every version of each
      kernel, based on a performance model.
   d) Select the best performing version of each kernel, and
      estimate the objective function.
   e) If the objective improves, continue with the new ADG.
3) Repeat until the objective (eg. perf²/mm²) converges

The following subsections describe how we improve the
time-per-iteration through a novel spatial-scheduling repair
technique, as well as the performance and power/area models.

A. Fast DSE with Repairing Scheduler

Challenge: Length of DSE Step: The most time consuming
aspect of each iteration is evaluating the performance aspect
of the objective function. This is due to the fact that the
performance of a spatial architecture can only be determined
once the program is mapped to the hardware; the mapping
affects the memory and resource contention, as well as the
latency of any critical dependencies. Unfortunately, spatial
scheduling is known to be a lengthy algorithm, with practical
heuristics taking on the order of at least minutes for larger
problems [52,64,66,71,75], and sometimes even longer. What
exacerbates this problem further is that the compiler must
consider different sets of transformations due to its modular
approach, multiplying the overhead by some factor.

Insight: The ADG at each iteration is not completely dif-

ferent: perhaps some edges are added to increase network
connectivity, processing elements may be added to increase
performance, and a small subset of the above may be deleted
to improve area/power overheads. For many incrementally
changed ADGs, the previous schedule will still be valid. Even
when a hardware component which is used by a program
region is deleted, the remainder of the schedule is still valid.

Repairing Spatial Scheduler: Therefore, we propose a DSE-
approach with a repairing spatial scheduler. After each ADG

for (i0=0, i1=0; i0<n0 && i1<n1; ) {
  if (k0[i0]<k1[i1]) {
    ++i0; // pop k0, v0
  } else if (k0[i0]==k1[i1]) {
    ++i1; // pop k1, v1
  } else {
    // implicitly k0[i0] == k1[i1]
    acc += v0[i0]*v1[i1]; // do compute
    ++i0; ++i1; // pop both
  }

  All the branches are dominated by the result
  of the comparison between k0[i0] and k1[i1].
}

(a) Original C Code (Sparse Inner Product)
modification, the set of schedules being explored are updated to reflect the new hardware. Specifically, any aspect of the input program which used a deleted ADG component is also deleted from the schedule. Then schedule repair is performed (during step 2b), which attempts to both repair the incomplete schedule, as well as try to take advantage of any added hardware features. Scheduling repair is natural with the iterative-stochastic spatial scheduler described in Section IV, as this algorithm is anyways iterative – repair is implemented as starting with an initially possibly-unfinished schedule.

Figure 9 shows an example of a DSE step, where the input programs (dense and sparse vector multiply) are being explored. Figure 9(b) shows the ADG modification step; in this case the ADG modifier randomly chooses to delete the lower left PE; this invalidates the position and timing of the accumulate for the sparse multiply. The scheduler would then perform schedule repair on its dataflow, moving its accumulate to another available PE.

B. Performance Modeling Approach

We estimate the performance of a transformed code by estimating the IPC: IPC = #Insts × Activity Ratio.

The activity ratio is limited either by bandwidth from memory, or dependences within or between program regions. Therefore, the performance model computes 1. the memory bandwidth required to achieve fully pipelined execution, and 2. if there are any dependences (e.g. loop-carried dependence), the impact of those dependences on activity ratio. The memory bandwidth activity ratio is computed as the minimum ratio of bandwidth-requested / bandwidth-supplied for each memory. The dependence activity ratio is computed as the number of concurrent computation instances in the pipeline which can hide each dependence / dependence latency. Concurrent instances can be determined by analyzing the length of data stream, and the dependence latency is available in the spatial schedule. Finally, we need the execution frequency in order to normalize the importance of each region, for which we leverage LLVM’s BlockFregencyInfo.

C. Power/Area Modeling Approach

The iterative exploration approach requires a quick and accurate evaluation of the power/area of the proposed hardware. Traditional synthesis tools are too time-consuming to achieve a practical DSE. Therefore, we use an analytical regression model for power/area estimation. A dataset of all hardware modules with a sampling of possible parameters (number of I/O links, data width, register file size etc.) was synthesized to build the analytical model. For PEs, the power/area overhead includes its constituent function units. As an optimization, we develop functional units which support multiple functions (e.g. a 32-bit adder which can also perform subtract, and which can also be decomposed into two 16-bit adders).

D. Limitations

We assume that the change of the on-chip network topology and the parameters of each component will not significantly affect the clock frequency of the implemented hardware, because of the difficulty of estimating the synthesis timing and violation. Therefore, we fix each switch to flop its output so that each switch becomes a stage in the datapath pipeline. This makes it much less likely for the network to significantly harm the critical path.

Other Fixed Features During DSE: Similarly, some other features are fixed during DSE, even though the ADG could express them. This includes memories, for which we currently assume one memory interface (fixed) and one scratchpad (parameters of which are explored). We also do not change any parameters of the control core.

VI. HARDWARE GENERATION

The hardware generator not only produces RTL, but also formalizes the software/hardware interfaces.

Bitstream Encoding: Each component of the spatial architecture has local registers to store the bitstream that encodes the programmable information: A switch’s bitstream encodes the routing information. A PE’s bitstream encodes instructions opcodes, execution timing (for static PEs only), and instruction tags (for shared PEs only). A synchronization element’s bitstream encodes the cycles of delay. The spatial architecture is configured by loading the bitstream into these registers.

Configuration: We add one extra bit to the on-chip network to indicate a configuration message, which will be routed according to a static path determined by the hardware generator. There can exist multiple configuration paths, and each unit must be on some path. The configuration data also includes the ID of its destination, so the component can identify relevant configuration data to keep and not-relevant data to forward.

Config. Path Generation: Our framework supports arbitrary topologies, so we must construct a set of configuration paths which minimizes configuration time. We define the problem as finding one or more paths that covers all the nodes in the ADG which minimizes the length of the longest path. In our approach, we first use a spanning-tree like algorithm to get multiple initial paths. Then we iteratively apply a heuristic: cut a node from the longest path and connect to any nearby shorter paths. This continues until the maximum length converges.

Limitations: The hardware generator has limited capability to change the encoding format of each module. A future optimization would be to reduce the configuration bits for specifying a register based on the number of registers. Another potential optimization is to choose the most efficient implementation at gate-level for the same functionality to meet latency/throughput/frequency tradeoffs (like replacing a carry-ripple adder with carry-lookahead adder).

The capability of the generator to reuse hardware circuits for implementing different functionality is also limited. For example, it can reuse the hardware of a 64-bit adder to achieve 8-bit SIMD addition, but it is not currently able to reuse the alignment circuit of the floating-point divider to complete a shifting operation. Further optimizations at circuit and functional unit composition level is future work.
VII. METHODOLOGY

ISA & Compiler: We use a RISC-V-based control core. We extend Clang to implement required pragmas, which are conveyed as metadata to our LLVM-based compiler, and use the RISC-V GCC assembler backend.

Target Accelerators: We chose five accelerators to instantiate (approximately), to stress different hardware features:

- Softbrain [65] is instantiated using a mesh of static-scheduled/dedicated PEs and switches and a single non-banked scratchpad memory.
- MAERI [45] is approximated similarly to Softbrain, but with its novel tree-based topology.
- Triggered Instructions [69] is approximated with a mesh of dynamic-scheduled/temporal PEs. Our designs assume a group of PEs shares access to a decoupled scratchpad memory.
- SPU [20] is similar but has dynamic-scheduled/dedicated PEs, and banked scratchpad.
- REVEL [92] composes static-scheduled and dynamic-scheduled PEs in one mesh, and allows communication through synchronization elements.

In our experiments, we assume that accelerators are integrated to a high-bandwidth L2 cache (75 GB/s).

Simulation: For performance, we implemented a cycle-level simulator for all ADG components. This is integrated with gem5 [7] to use its RISC-V core [80] as the control core.

Benchmarks: We selected several workloads from multiple domains: 6 workloads from MachSuite [79], 5 benchmarks from PolyBench [77], 2 microbenchmarks from SPU [20] workloads, and 4 DSP workloads targeted by REVEL [92]. Table I shows the data size of each workload.

In our compiler, we added pragmas in Section IV. We also implemented manually mapped accelerator code in assembly. The original C codes are used as baselines, which are compiled by GCC-8 with -O3, on Intel Xeon Silver 4116 @2.10GHZ.

Power and Area Analysis: We implemented a parameterized CGRA generator with Chisel [5] backend to generate accelerator RTL. We synthesized the generated RTL using Synopsys DC with UMC 28nm UHD library (SVT, ff, 0.99v), with target 1ns clock period (1GHz). For floating-point units, we used Matlab HDL coder and Synopsys SMC. Using the results of synthesis, we constructed an analytical regression model for quick power/area estimation within the design-space explorer.

VIII. EVALUATION

We evaluate DSAGEN’s compiler, design space explorer and hardware generator. The major takeaways are:

- The compiler achieves 89% of the performance of manually tuned versions, and each modularized compilation feature can be independently enabled/disabled.
- According to our estimation, the design space explorer is able to save 42% power and area over the initial hardware.
- The automated DSE generates hardware with mean 1.3× perf/mm² comparing with prior programmable accelerators across multiple sets of workloads.

A. Modular Compilation

Performance: Figure 10 shows that the compiler achieves 89% of the performance of manually tuned versions. The performance degradation is mainly due to a lack of peephole optimizations in the compiler: the manual versions exploit features of the low-level ISA to reduce the number of control instructions. An outlier is fft, which is 2× slower in both REVEL and Triggered Instructions. In the last of several iterations of fft, the stride of data access becomes so small that the compiled version may generate too many requests to the same line, which underutilizes the scratchpad bandwidth. The manual version peels down these underutilized iterations, and combines these requests to avoid this pattern.

Modularity: To demonstrate the robustness of the modularity, we evaluate the performance on a baseline architecture with different sets of features turned on/off. The baseline architecture is a 4x4 mesh of dedicated static PEs, 64-bit network, and 512-bit wide scratchpad. Three key features can be individually enabled or disabled:

- “shared” designs replace four dedicated PEs with shared PEs to balance resource utilization across inner/outer loops.
- “dynamic” scheduling confers the ability to handle control-dependant data-reuse (aka. stream join [20]).
- “indirect” designs support vectorized indirect load/update.

Figure 12 shows how each feature affects the performance. Here, 0/1 means the corresponding feature is disabled/enabled. PolyBench workloads are all simple dense linear kernels with mostly perfect loops, so adding or removing features does not change the performance. However, DSP workloads heavily benefit from shared PEs for their outer-loop computations, and Sparse workloads benefit from indirect access and dynamic scheduling due to frequent data-dependence. Across all workloads, the best design includes all features.

B. Design Space Exploration

The goal of our design space exploration is to demonstrate the ability to automatically tune the fine-grain hardware/software features and architecture topology. We evaluate three different sets of workloads:

- MachSuite: This set represents a variety of workloads with different needs and some irregularity. This allows us to compare DSAGEN’s design (DSAGEN_{MachSuite}) against a hand-designed accelerator for these kinds of workloads: Softbrain [65].

<table>
<thead>
<tr>
<th>Workloads</th>
<th>MachSuite</th>
<th>Sparse</th>
<th>Dsp</th>
<th>PolyBench</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Size</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>md</td>
<td>128 x 16</td>
<td>464 x 4</td>
<td>643</td>
<td></td>
</tr>
<tr>
<td>crs/ellpack</td>
<td>32 x 16 x 2</td>
<td>2 x 10 x 2</td>
<td>2 x 10 x 2</td>
<td></td>
</tr>
<tr>
<td>mm</td>
<td>130 x 3</td>
<td>768 x 2</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>stencil-2d</td>
<td>32 x 2</td>
<td>32</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>stencil-3d</td>
<td>32 x 16</td>
<td>32</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>histogram</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>join</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>qr</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>chol</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>fft</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>mm</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>2mm</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>3mm</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>atax</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>bicc</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>mvt</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
</tbody>
</table>

TABLE I: Workload Specification
Dense neural networks: We evaluate convolution, pooling, and classifier kernels, which have regular access and control. This allows us to compare the generated design (DSAGEN\textsuperscript{DenseNN}) against not only Softbrain, but also a domain-specific accelerator: DianNao [12].

Sparse convolutional neural network: This is a single workload: outer-product multiply and resparsification. It has regular computation but data-dependent memory access. We compare DSAGEN\textsuperscript{SparseCNN} against SCNN [70] (a fixed accelerator) and SPU [20] (a programmable accelerator for sparse workloads).

We perform three DSE runs starting from the same initial hardware, a 5×4 mesh with full capability, including control flow, FU decomposability, and an indirect memory controller. The design space explorer estimates the performance, power, and area using the model discussed in Section V, and the objective function is \( perf^2/mm^2 \). The explorer runs up to 200 scheduling iterations to initialize or repair the mapping after changing the hardware. The algorithm will exit after 750 iterations without objective improvement.

Figure 14 shows how the area (left bar), power (right bar), and overall objective (color intensity) evolve during design space exploration. The first two iterations initialize the exploration: after the datapaths are mapped to the initial hardware in the first iteration, the redundant features, including known unneeded functional units and address generation capability are removed. Because of the objective function, achieving better performance has higher priority than saving resources. Therefore, in these three DSE runs, the estimated performance is enhanced, and then the explorer trims redundant resources. It is hard to map sparse CNN’s datapaths onto the initial hardware within a few iterations, so the explorer in the early iterations adds some redundant compute and routing resources to ease the difficulties of mapping. For MachSuite, the memory bandwidth is the bottleneck, so the explorer adds more initially. Subsequently the explorer focuses on enhancing the reuseability of on-chip network across multiple workloads, and minimizing the synchronization element depth.

Overall, our design space explorer saves mean 42% of the area and achieve mean 12× objective improvement over the initial hardware across the three selected sets of workloads.
We validate our power/area regression model by comparing the numbers against synthesis. The results are shown in Figure 15. The bold label is the corresponding hardware, which is either a DSE generated hardware or existing reconfigurable accelerator. “Est.”, “Synth”, and “Scaled” stand for “estimated by the regression model”, “obtained by synthesis”, and “obtained from prior paper by technology scaling” respectively.

For the generated hardware, the estimate is 4-7% smaller than the synthesis area/power. While the model was tuned by synthesizing each component alone, extra structures are required to meet timing for the whole fabric. Our estimated model shows a somewhat large discrepancy between estimated and scaled area/power of Softbrain and SPU, which is partly due to microarchitecture and technology/scaling differences. Further, some overhead may be due to having to provide more general protocols for modularity.

To validate the performance model, we simulate the generated hardware with the compiled programs after DSE. The model has mean performance error of 7%, with maximum error of 30%. The maximum error occurred in stencil-3d, because our model does not yet capture the performance impact of excessive control instructions.

Quality of the Generated Hardware: Figure 15 shows comparison of DSAGEN designs with corresponding less-specialized programmable accelerators (Softbrain and SPU). According to our regression model’s estimation, DSAGEN\textsubscript{DenseNNet} and DSAGEN\textsubscript{Sparse CNN} saves 64% and 18% area comparing against SPU and Softbrain for respective workloads. While DSAGEN\textsubscript{MachSuite} introduces 1.2× area overhead comparing with Softbrain, it also provides 1.2× speedup (favorable given the objective function).

We also compare against scaled-domain-specific accelerators, DianNao and SCNN, for reference; this is not particularly accurate due to technology differences. DSAGEN\textsubscript{DenseNNet} has overhead of 2.4× area and 2.6× power over scaled DianNao. DSAGEN\textsubscript{SparseNN} is 1.3× area and power over SCNN. We believe the overhead is mainly from reconfigurability. While these accelerators use a specific network (eg. tree in DianNao), DSAGEN’s irregular network does not converge perfectly to these specific, perhaps optimal, topologies. There is still future work to be done to improve the design space exploration.

Schedule Repair: We compare two different strategies, traditional scheduling (map entire dataflow every iteration) and our schedule repair approach. During DSE, after each iteration of the hardware update, both perform up to 200 scheduling iterations. The result is shown in Figure 11 for the MachSuite workloads. At the early stages, both strategies have a very close objective, because there are abundant resources on the hardware and scheduling is simple. Remapping the whole schedule can still succeed within 200 iterations. When the hardware resources become tight, the traditional scheduler cannot succeed on these more efficient designs, because it has to re-discover the entire mapping. Overall, schedule repair leads to a 1.3× better objective for DSE.

Configuration Path: Improving the configuration time can aid performance of short program regions. Configuration time is dominated by the longest configuration path. We evaluate the path generator by giving it multiple mesh spatial architectures (2 × 2 to 5 × 5 PEs) under the constraint of having 3, 6, and 9 configuration paths, and the result is shown in Figure 13. The dashed lines are the ideal lengths (for a network with n nodes, p paths, the longest path cannot be shorter than ⌈ n / p ⌋), and the solid lines are the actual lengths. The path generator only introduces mean 1.4× overhead versus the ideal.

IX. RELATED WORK

DSE for General Purpose Processors: Custom fit processors [23] is a framework to build application-specialized VLIW designs. Somewhat related proposals target customized VLIW or superscalar pipelines through some codesign process [4, 18, 19, 32, 35, 57]. Another related work is for general purpose processors called Liberty [89], which uses a microarchitecture specification to generate simulators and perform DSE. Similar frameworks include Expression [29], UPFAST [67] and LISA [72]. None of these support spatial architectures.

Network Synthesis: Network synthesis techniques enable customized network topologies based on workload properties. One example is SUNMAP [59], which performs network topology synthesis, and similar techniques have been developed for irregular network topologies [76, 90]. DRNoC [43] and Connect [68] are network generators tailored for FPGAs. DRNoC is particularly relevant, as it generates a network based on the application’s task graph. These NoC generators only addresses network design without considering computation. Other works map applications onto potentially irregular NoCs [37, 58], but do not perform codesign search.

Accelerator Design Frameworks: CGRA-ME [13] is a design framework for static-scheduled CGRAs. It uses a C programming strategy, and includes fast power and area models [63]. The framework has a generic spatial scheduler for any topology based on integer linear programming [14, 91]; this is too slow for DSE. Several works explore the design space for CGRAs, including ADRES [8] and the KressArray Explorer [30]. Kim et al. develop a design space exploration framework tailored for DSP applications [40]. EGRA [3]
is another template-based CGRA which supports compound functional units (we do too through composition). RADISH is a CGRA generator which uses genetic algorithms to search for compound PEs based on a corpus of applications [93]. Suh et al. propose a CGRA with heterogeneous FUs, amenable to DSE [87].

The Spatial [41] compiler uses DSE to map parallel programs to FPGAs and the Plasticine [78] CGRA, with an optimizer called HyperMapper [61]. It is fundamentally orthogonal as it is targeting the compilation problem and not DSE of the architecture itself.

μIR [83] is an IR and framework for designing application-specific accelerators that exposes microarchitecture features as first-order primitives.

Key Differences: None of the above 1. have a design space including multiple execution models (eg. dynamic-static scheduling, dedicated-temporal PEs), and 2. perform topology search to specialize the hardware datapath to a set of programs.

X. CONCLUSION

To broaden the potential of acceleration, this work develops an approach and framework, DSAGEN, for programmable accelerator synthesis. In this paradigm, an accelerator can be developed by composing simple spatial architecture primitives, and also be generated through automated codegen. Codegen works because the compiler can understand how best to use the simple primitives that are composed in an architecture description graph. Modular compiler transformations can robustly target accelerators with different ISA features, parameterizations, and topologies. Further, only traditional languages are required with relatively little programmer intervention.

More broadly, the field of computer architecture has historically grappled with what should be the layers of abstraction from hardware to software to enable efficient designs. A fixed ISA has been both the typical assumption and a persistent burden. This work suggests that the ISA does not need to be the hardware/software abstraction which designers rely on, at least for the domain of accelerators. Instead, a modular accelerator description can serve that purpose, and enable much greater flexibility to explore deeply specialized designs.

REFERENCES


