DAEGEN: A Modular Compiler for Exploring Decoupled Spatial Accelerators

Jian Weng, Sihao Liu, Vidushi Dadu, and Tony Nowatzki

Abstract—Specialized hardware accelerators, particularly those that are programmable and flexible to target multiple problems in their domain, have proven to provide orders of magnitude speedup and energy efficiency. However, their design requires extensive manual effort, due to the need for hardware-software codesign to balance the degree and forms of specialization to the domains or program behaviors of interest. This article provides the first steps towards one approach for automating much of these processes. The insight behind our work is to recognize that decoupled spatial architectures both define a rich design space with many tradeoffs for different kinds of applications, and also can be composed out of a simple set of well-defined primitives. Therefore, we propose a modular accelerator design framework, DAEGEN, a.k.a. Decoupled Access Excution Accelerator Generator. This article defines an initial compiler and architecture primitives, and we discuss key challenges.

Index Terms—Reconfigurable accelerators, design automation, hardware/software co-design, spatial architectures

1 Introduction

As a response to technology scaling challenges, specialized accelerators have proliferated in many areas (datacenters, mobile phones, edge devices, etc.). Two basic strategies have emerged for generating new accelerators, each with their own benefits and limitations:

- *Application Specific (eg. HLS):* High-level synthesis compiles languages like C with pragmas to hardware. While HLS is nearly automatic and produces designs that have a fully customized hardware and software interface, the design space is limited, and designs are not programmable.
- *Domain Specific (eg. TPU)*: This approach customizes hardware for the kernels within a domain, and provides a domain-specific software interface. The advantages are high performance and flexibility, at the expense of hardware and software design effort, an effort that must be repeated as workloads change.

There is a large space of applications for which neither approach is satisfying: where some flexibility is needed, but the cost of a domain specific design cannot be justified. For such settings, an ideal specialization approach would yield the level of automation provided by HLS, but still enable exploration of programmable architectures without redesigning hardware software interfaces.

One of the primary challenges is defining a searchable design space which is both large enough for potentially broad applicability and customizable enough to achieve the benefits of specialization. Prior work (eg. [1], [2], [3], [4], [5], [6]) suggests that spatial decoupled access execute accelerators are a good candidate. *Spatial* refers to designs with hardware/software interfaces that expose low-level communication and scheduling decisions–this can enable high parallelism at low cost. *Decoupled* refers to the separation of concerns between the hardware for memory access and computation, enabling specialization of each. Moreover, these architectures are attractive as it is possible to define a set of primitives which have semantics that can be understood by a compiler, so that they can be composed in flexible ways (refer to Section 3). A number of example designs are shown in Fig. 1. These architectures are attractive both because they are efficient, but more importantly because their primitives are *composable*.

Therefore, our goal is to develop the concept and a usable framework for developing programmable decoupled spatial accelerators.

Central to our approach is a representation of the hardware called the architecture description graph (ADG), which is composed of parameterizable primitives like processing elements (PEs) and switches. This is used in the compiler, simulator, and hardware generator without manual intervention, as outlined in Fig. 2. The first step in compilation is for each kernel to be transformed into a representation suitable for decoupled spatial architectures; several different versions of each kernel are created with different sets of transformations, each targeted to optional architecture features. Then the hardware mapper will distribute each instance of the program to hardware resources, and the legal one with the best performance. Architects can use information like hardware utilization and performance sensitivity to guide ADG exploration. During this process, no compiler backends need be implemented; the ISA is effectively determined by the ADG, and the compiler is modular to handle arbitrary sets of features in the design space. This is what makes deep exploration feasible.

In our evaluation, we show that the ADG is general enough to express variants of five spatial architectures (Softbrain [3], TIA [1], MAERI [4], SPU [2], and REVEL [7]). For these, our compiler can generate code with mean 1.32× execution time of manual versions across several workload sets, including those with control and memory irregularity. Finally, we show how our tool can be used to help uncover the best set of hardware/software features for a given workload set.

The main contribution of this work is to demonstrate that decoupled spatial architectures have composable primitives which can form a rich design space, and that it is possible and useful to explore hardware/software interfaces within such a framework.

Organization. We first describe our spatial architecture design space and hardware generation (Section 2), and then outline the compilation techniques (Section 3). We evaluate DAEGEN by its breadth of architecture scope as well as comparison to manual optimization (Sections 4, 5).

2 Decoupled-Spatial Architectures

2.1 Design Space

At the heart of our programmable accelerator framework is a graph-based representation of the decoupled spatial hardware components primitives, described in Fig. 3 (processing elements, switches, memories, delay elements, and synchronization elements). Changing parameters of PEs, (eg. which set of functions they can perform), or parameters of memories (eg. degree of banking), enable a tradeoff between flexibility and specialization.

Fig. 1 shows examples of architecture description graphs (ADGs) composed from these primitives. Each node has its own attributes which correspond to the parameters shown in Fig. 3. For example, the "S" components in Fig. 1 are switches. Each would be annotated with their bitwidth, ability to support dynamic scheduling, and their routing connectivity (which inputs may flow to which outputs). These designs demonstrate the wide range of efficiency versus generality which is possible to express. For example, the ratio of datapath flexibility versus switch overhead. CCA [5] has the fewest switches, but has only limited flexibility. The Softbrain(d) [3] is

• *The authors are with University of California, Los Angeles, Los Angeles, CA 90095. E-mail: jian.weng@ucla.edu, {sihao, vidushi.dadu, tjn}@cs.ucla.edu.*

Manuscript received 17 Oct. 2019; revised 13 Nov. 2019; accepted 17 Nov. 2019. Date of publication 25 Nov. 2019; date of current version 13 Jan. 2020.
(Corresponding author: Jian Weng.)
Digital Object Identifier no. 10.1109/LCA.2019.2955456

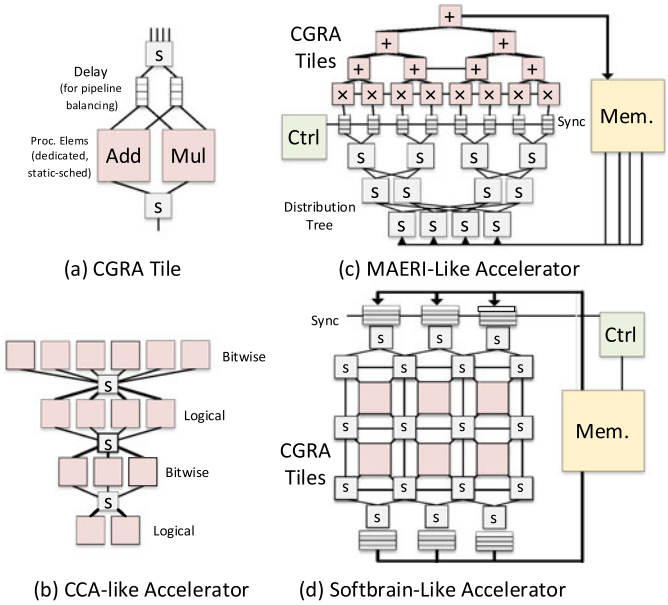

Fig. 1. Decoupled spatial architectures These are represented as an architecture description graph (ADG) in DAEGEN (primitives in Fig. 3).

the most flexible but with the highest overhead. The MAERI(c) [4] is somewhere in the middle, customized for DNNs.

The high-level execution model for these architectures is that programs execute in a series of phases, where in each phase the memory and compute units are configured with the appropriate parameters. We assume a Von Neumann control core to execute the control finite state machine (FSM), but our approach is not specific to this decision.

One large benefit of the ADG abstraction is the ability to specify arbitrarily application-specific accelerators, in that the datapath and set of memories can exactly match the needs of the original program or closely match a set of related programs. Later, we discuss how the compiler framework will attempt to take advantage of whatever resources and topology is available.

Breadth of Design Space. A key distinguishing feature of our decoupled spatial design space is that we enable composition of computation and network elements which have different execution models. As a simplification, there are two major execution model dimensions: 1. Processing and network elements can be either *statically or dynamically scheduled*. 2. They may also be *dedicated* to a single instruction (eg. akin to a systolic array), or be *temporally shared* amongst multiple instructions (eg. like a traditional coarse grain reconfigurable architecture (CGRA)).

One final important aspect of the design is the support for irregular memory and control. One option we consider here is for the memories to enable indirect-atomic access at high bandwidth (eg. `a[f(b[i])]++`), similar to SPU [2];

Principles of Composition. We overview the basic principles and considerations for composition. First, *statically scheduled elements* have less hardware overhead, but require a synchronization point to guarantee that all inputs are available at a known time. The *synchronization element*, which buffers a configurable set of inputs and releases when all are ready, aids the compiler to provide this guarantee.

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 [8]. The delay element allows a configurable delay to aid the compiler to compensate to enable the use of dedicated elements efficiently.

Central to composability support is 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. Switches can connect PEs regardless of whether they use static/dynamic scheduling, or if they are dedicated or shared. The compiler will then enforce that values do not flow from static to dynamic PEs (without going through sync. element) or from dedicated to temporal PEs (due to overwhelming temporal PE).

2.2 ISA and Hardware Generation

DAEGEN generates an ISA and hardware design,[1] by selecting an encoding of instructions, and also generating a configuration path.

For speed and efficiency, the hardware generator reuses the datapath for configuration. All elements are routed a configuration enable signal to start configuration mode. The configuration information encoding is composed of a module id and control information, so that configuration may be pipelined. According to different types of modules, the control information may include opcode, explicit timing delay (static-dedicated PE), routing information (switch), etc.

In order to be able to reconfigure all modules in network, one or more configuration paths (depending on the bandwidth) should be constructed to reach each nodes in the ADG. DAEGEN will generate config path(s) to cover all nodes in ADG, with the goal of minimizing the maximum path length (which dictates configuration time). By treating this problem as *non-cycle graph traversal problem with multiple start nodes*, DAEGEN framework will first generate config paths by connecting all nodes randomly and then move nodes from the longest path to the shortest to correct for imbalance in the path length. The overall ISA is determined by the set of components, their parameters, and the configuration path through them.

Finally, generated hardware requires a method to program and coordinate memories, as well as configure and synchronize the spatial architecture over multiple phases. We leverage the stream-dataflow ISA [3] for this, so this part of the ISA is fixed.

3 Modular Decoupled Spatial Compilation

We developed a C+pragma interface to enable simple programming, and implemented within Clang. The pragmas both aid in the compilation of decoupled spatial architectures, and yet are agnostic to specific hardware features. The compiler has two basic responsibilities: 1. extracting and mapping the computations to the spatial accelerator, and 2. decoupling the involved data access and encoding them.

Programming Interface. To achieve the above, the code transformation pass should first be aware of: 1. the instructions to be offloaded to the spatial fabric and how they occupy/share the resources on it, and 2. the data access instructions that can be decoupled without violating the original semantics. Therefore we support the following pragmas, with an example in Listing 1.

`#pragma dae offload`: This pragma defines a code region whose computations will be offloaded to the spatial fabric.

`#pragma dae decouple`: This pragma informs the compiler that all memory dependences are enforced through data-dependences (ie. no unknown aliasing). This enables the compiler to decouple and hoist memory operations to the loop level annotated with this pragma.

1. In Chisel RTL, though we note that generated RTL validation is incomplete.

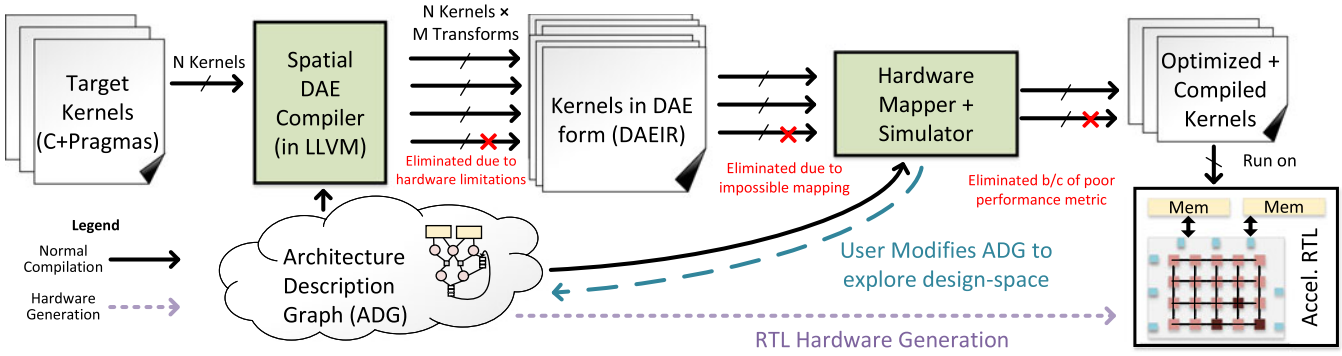

Fig. 2. Overview of DAEGEN architecture design framework.

`#pragma dae config`: This pragma defines the program scope where reconfiguration occurs – ie. regions within this scope are concurrent on the spatial fabric.

Listing 1. An Example of C+Pragma Annotation

```
1 #pragma dae config
  {
3  #pragma dae decouple
   for (i = 0; i < n; ++i) {
5   v = 0;
    # pragma dae offload
7   for (j = 0; j < n; ++j)
      v += a[i * n + j] * b[j];
9   #pragma dae offload
    for (j = 0; j < n; ++j)
11    a[i * n + j] -= v * b[j];
  ...
```

Decoupling Memory and Compute. To extract data accesses, we inspect each code block marked with `offload` pragma, and extract memory access patterns by finding the program slice from the address of each memory access. The sliced memory operations will be analyzed by LLVM's SCEV module. We use this infrastructure to gather the stride and trip count information of each loop level over the memory pattern. Address computations are hoisted to the appropriate loop level, which is similar to the approach in Clairvoyance [9], and we encode each pattern as a "stream" intrinsic [3]. Remaining operations are fed to a spatial mapper, and eliminated in the LLVM IR.

Spatial Hardware Mapper. The responsibility of the spatial hardware mapper is to map instructions to spatial PEs, dependences to the network, memory access to memories, and manage with the timing of data arrival (if necessary). We use a simulated annealing algorithm to map the instructions onto the spatial fabric, and a heuristic-based search to route the network (an improved version of [8]).

Modularized Code Generation. Our compiler takes a modular approach to compilation, where certain features are used if they exist and can be legally used within the defined architecture. There is always a slower fallback if the feature does not exist.

One example of this is region selection. The compiler first tries to offload as many regions as possible, but if the spatial compiler fails to map every region onto the fabric because of insufficient instruction slots, the compiler will first drop those regions with relatively low execution frequency and re-invoke the spatial compiler. We use LLVM's static `BlockFrequencyInfo` module to estimate frequency. We use a similar technique to increase the unrolling degree of frequent blocks to attain better utilization of the spatial fabric.

Second, the compiler can convert control flow to predicate-based dataflow (referred to as stream-join in [2]). If the underlying hardware does not support this idiom, the mapper will inform the compiler so that it can fall back to leaving these code on the host.

Third, the compiler can detect and encode indirect memory accesses like `a[b[i]]`. If the underlying hardware cannot support these intrinsics, our compiler may invoke the fall-back module to generate scalar data to feed the offloaded code region.

4 Evaluation Methodology

Target Accelerators. We chose five accelerators to stress DAEGEN on different hardware execution models: Softbrain [3] and MAERI [4] are both static-dedicated, but MAERI has a tree-based interconnect (all others have a mesh). SPU [2] has dedicated-dynamic network, Triggered-Insts [1] has a temporal-dynamic network, and REVEL [7] uses both static-dedicated and temporal-dynamic PEs/switches. For simplicity, we assume all designs have floating point units and a 16-KB scratchpad with 512-bit bandwidth, so these are not exactly identical to the original architectures. Table 2 shows the parameters of the target accelerators. We construct a unified cycle-level simulator for performance results, and assume 1.25 GHz frequency.

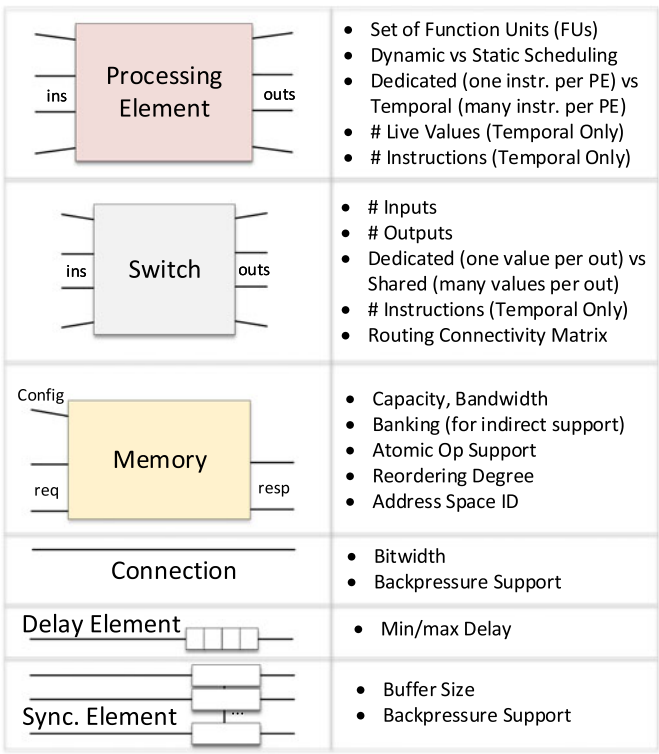

Fig. 3. Decoupled spatial architecture components.

TABLE 1
Workload Specification

Benchmarks	MachSuite					Sparse		Dsp				PolyBench				
Workloads	md	crs/ellpack	mm	stencil-2d	stencil-3d	histogram	join	qr	chol	fft	mm	2mm	3mm	atax	bicg	mvt
Data Size	128×16	494×1666	64^3	$130^2 \times 3^3$	$32^2 \times 16 \times 2$	$2^{10} \times 2^{16}$	768×2	32^2	32^2	2^{10}	32^3	32^3	32^3	32^2	32^2	32^2

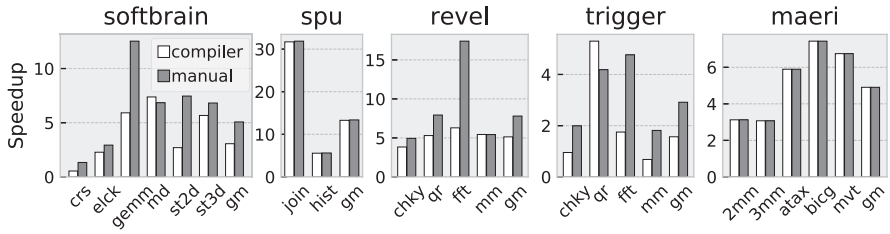

Fig. 4. Compiler versus manual-tuned performance.

Benchmarks. We selected 6 workloads from Machsuite [10], 2 micro benchmarks from SPU's workloads, 5 benchmarks from PolyBench [11], and 4 DSP workloads for TIA and REVEL. Table 1 shows the data size of each workload and the hardware features they may use. We compile these original C programs with GCC-8 `-O3`, and run on Intel Xeon Silver 4116 @2.10 GHz as the baseline.

Area Analysis. We synthesized DAEGEN's generated RTL with Synopsys DC with UMC 28nm UHD library (HVT, ff, 0.99v).

5 DAEGEN Evaluation

Our evaluation answers two main questions about DAEGEN. First, how well can the compiler target diverse accelerators with its modular approach as compared to hand-tuned code? Second, what are the opportunities for modular-architecture codesign?

Modular Compilation Efficiency. Fig. 4 shows performance of each accelerator with respect to baseline CPU. Our compiler is able to achieve in average 75 percent of the performance of manually tuned kernels. The primary reason for the difference is that manually tuned kernels use fewer instructions to configure memory, and do not overburden the core (a problem for short loops only).

Each of the designs above makes a tradeoff in hardware complexity versus generality. Fig. 5 shows the power and area per operation in each accelerator–the fact that they are so different comes both from their topology (tree versus mesh) and the features that they employ. MAERI is the most efficient–it has the highest computation to memory ratio, use static routing, and has lower-degree switches – but it comes at the price of not being effective for irregular computations.

Modular Codesign Opportunities. To better demonstrate the generality versus efficiency tradeoffs for modular hardware/software features, we evaluate different combinations of features on top of a consistent base architecture: 4x4 mesh of dedicated static PEs, 64-bit network, and 512-bit wide scratchpad. We consider adding three key features:

- *"shared"* designs replace four dedicated PEs with shared PEs to better balance resource utilization across inner/outer loops.
- *"dynamic"* scheduling confers the ability to handle control-dependent data-reuse (aka. stream join [2]).
- *"indirect"* designs support vectorized indirect load/update.

Fig. 6 shows how each feature affects the performance. Here, "0 0 0" means no features are selected, and "1 1 1" means all features are selected. PolyBench does not require optional features, as they are regular, so performance/mm^2 only degrades with more features. 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.

TABLE 2
Hardware Parameters (d:Dynamic Sched, s:Temporal Shared, mix:Some of Both)

Architecture	#PEs	#Switch	Z#Add.	#Mult.	#Div.
Softbrain	4×4	5×5	8	8	0
SPU	5×4	6×5 (d)	12	8	0
Trigger	3×3	4×4 (d&s)	4	4	1
REVEL	4×4	5×5 (mix)	8	8	1
MAERI	31	31: tree	16	15	0

6 Related Work

Custom fit processors [12] is a framework to build application-specialized VLIW designs only. A related area is network synthesis (eg. SUNMAP [13]), but this only addresses network design without considering computation.

The most related body of work are tools for CGRA exploration, of which a few include Plasticine [6], CGRA-ME [14], and ADRES explorer [15]. To our knowledge, none of the above 1. have a design space including multiple execution models (eg. dynamic and static scheduling), and 2. allow for irregular topologies to potentially specialize to the datapath of a particular program or set of programs.

DAEGEN's C pragmas bear similarity to those in OpenMP: `dae offload` is similar to `omp task` and `dae config` is like `omp taskgroup`.

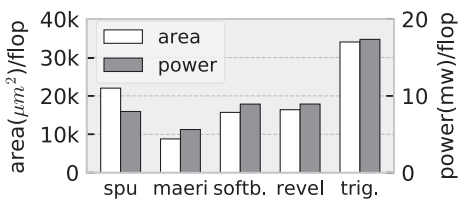

Fig. 5. Area and power normalized by peak FLOPs.

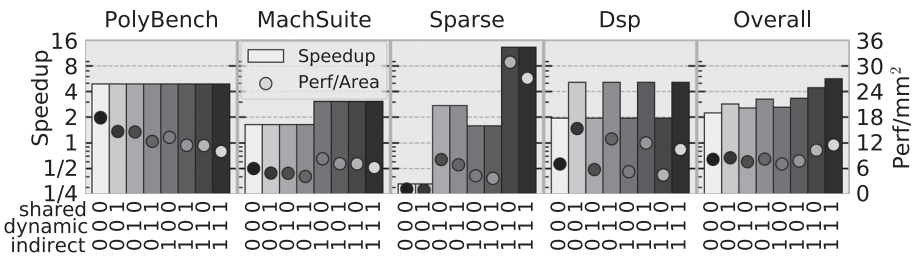

Fig. 6. Modular architecture design space exploration.

7 Conclusion

This work demonstrates that there exists a broad decoupled spatial accelerator design space, capable of high performance and efficiency on regular and irregular workloads. This design space can be expressed as a composition of simple architecture primitives. This enables exploration because 1. these primitives are composable and can be understood by a compiler, and 2. have parameters which fundamentally alter the tradeoffs for workloads with different properties. We show that compilation from high-level languages is made possible with simple pragmas and a modular approach with per-feature fallbacks.

There are many challenges left, including broadening the design space to include tradeoffs in multicore architectures, using higher-level program representations to eliminate the need for pragmas and enable deeper loop transformations in the search, and enabling tractable automatic search of a vast programmable accelerator design space.

Acknowledgments

This work was supported by an NSF CAREER award CCF-1751400.

References

[1] A. Parashar et al., "Triggered instructions: A control paradigm for spatially-programmed architectures," in *Proc. 40th Annu. Int. Symp. Comput. Architecture*, 2013, pp. 142–153.

[2] V. Dadu and T. Nowatzki, "Towards general purpose acceleration by exploiting common data-dependence forms," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2019, pp. 924–939.

[3] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Architecture*, 2017, pp. 416–429.

[4] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects," *ACM SIGPLAN Notices*, vol. 53, pp. 461–475, Feb. 2018.

[5] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner, "Application-specific processing on a general-purpose core via transparent instruction set customization," in *Proc. 37th Int. Symp. Microarchitecture*, 2004, pp. 30–40.

[6] R. Prabhakar et al., "Plasticine: A reconfigurable architecture for parallel paterns," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Architecture*, 2017, pp. 389–402.

[7] J. Weng, V. Dadu, and T. Nowatzki, "Exploiting fine-grain ordered parallelism in dense matrix algorithms," 2019, *arXiv arXiv: 1905.06238*.

[8] T. Nowatzki et al., "Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign," in *Proc. 27th Int. Conf. Parallel Architectures Compilation Techn.*, 2018, Art. no. 36.

[9] K. Tran et al., "Clairvoyance: Look-ahead compile-time scheduling," in *Proc. IEEE/ACM Int. Symp. Code Gen. Optim.*, Feb. 2017, pp. 171–184.

[10] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "MachSuite: Benchmarks for accelerator design and customized architectures," in *Proc. IEEE Int. Symp. Workload Characterization*, 2014, pp. 110–119.

[11] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," 2012. [Online]. Available: http://www. cs. ucla. edu/pouchet/software/polybench

[12] J. A. Fisher, P. Faraboschi, and G. Desoli, "Custom-fit processors: Letting applications define architectures," in *Proc. 29th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 1996, pp. 324–335.

[13] S. Murali and G. De Micheli, "SUNMAP: A tool for automatic topology selection and generation for NoCs," in *Proc. 41st Design Autom. Conf.*, 2004, pp. 914–919.

[14] S. A. Chin et al., "CGRA-ME: A unified framework for CGRA modelling and exploration," in *Proc. IEEE 28th Int. Conf. Appl.-Specific Syst. Architectures Processors*, 2017, pp. 184–189.

[15] F. Bouwens et al., "Architectural exploration of the ADRES coarse-grained reconfigurable array," in *Proc. Int. Workshop Appl. Reconfigurable Comput.*, 2007, pp. 1–13.