A Full-Stack Infrastructure for Automating Spatial Architecture Research

Jian Weng, Sihao Liu, Dylan Kupsh, Tony Nowatzki
University of California, Los Angeles
{jian.weng, sihao, dkupsh, tijn}@cs.ucla.edu

Abstract—The waning benefit of transistor scaling has driven the emergence of specialized accelerator research because of their orders-of-magnitude performance and energy saving. While promising, all these accelerators require extensive engineering effort to design and develop the hardware as well as the software stack. When the target domain shifts, this process should be repeated. Automated accelerator generation tools, like HLS, do not suffice when some hardware flexibility is desired.

An ideal domain-specific accelerator design flow should automatically figure out the design tradeoffs and the hardware specialization requirements by taking advantage of the awareness of the opportunity of hardware specialization within the target domain. This raises questions on: 1. how to designate a rich design space while retaining a unified programming interface; 2. how to effectively take advantage of the awareness to search through the design space.

In this work, we discuss the possible solutions based on our experiences on developing accelerator generation framework, include DSAGEN, and OverGen. We will demonstrate that a wide range of application domains can be effectively specialized by this research infrastructure, and the generated programmable designs can achieve comparable performance against application-specific design generated by HLS on FPGA.

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

I. INTRODUCTION

Domain-specific accelerators have drawn significant attentions because of their promising speedup and energy saving over general-purpose processors while retaining flexibility.

However, the current approaches on accelerator design and programming will not suffice. HLS automatically produces an accelerator for individual application, while limiting the flexibility. Programmable domain-specific accelerators retain flexibility, but require a lengthy process to design the hardware and develop the software stack. An ideal accelerator design flow should achieve the best of both worlds — generating specialized accelerators while retaining programmability.

Based on experiences from our prior works [1,2], we suggest that the key to programmable accelerator design automation is the awareness of the program behaviors of interests (which we will refer to “idioms” in the rest of the paper) so that corresponding hardware specialization mechanisms can be developed and integrated. Figure 1 overviews the flow of our proposed framework.

The principle of having such an automated accelerator design framework is that many prior accelerators can be approximated by composing a set of simple and common hardware primitives, such that each hardware specialization feature can be independently/modularly integrated/disabled. This designates a rich hardware design space. All these primitives are strait-forward enough, so that they can be understood by a compiler for a unified programming interface. To robustly target to designs within this design space, we adopt a modular compiler transformation. On the top of this full-stack hardware/software system, we develop a design space explorer, which automatically determines the specialized feature demands and figure out the hardware/cost and performance tradeoff. The target hardware iteratively evolves under this explorer, guiding by the hardware/software affinity estimated by analytical models.

Besides accelerator design automation itself, this approach can also be used to improve the usability of FPGA by generating programmable overlay designs. The flexible generated design not only saves the total time of application generation comparing with individualized application-specific high-level synthesizing accelerator, but also avoids thorough re-compilations caused by minor changes on applications.

Paper Organization: In the rest of this paper, we first discuss the background of the target hardware in Section 2. After that, we discuss the software stack design and the compilation techniques targeting to this hardware paradigm in Section 3. In Section 4, we discuss our design space exploration flow. In Section 5, and 6, we first discuss each key aspect of the research infrastructure, and then demonstrate the usability of this infrastructure by showing the result of experiments conducted. Finally, we discuss the future directions in Section 7.

II. BACKGROUND: DECOUPLED-SPATIAL ARCHITECTURES

By studying many prior accelerators [3–7], we found that a wide range of them can be approximated by the decoupled-spatial paradigm. The decoupled-spatial paradigm is especially attractive to us because of their performance, and flexibility. decoupled indicates that program behaviors, more specifically memory and computational operations, are decoupled and specialized by different hardware components separately. spatial indicates that the computational operations are mapped to a spatial architecture. Spatial architecture typically refers to architectures with a spatial layout that
__Execution Paradigm:__ Figure 2 shows how a dot-product is mapped to the decoupled-spatial execution. The computational instructions and the memory accesses are decoupled and specialized separately. The computational instructions are mapped to spatial processing elements, and the dependences among the instructions are routed through the on-chip network. All the decoupled memory accesses with specific patterns are encoded in hardware intrinsics, and executed by the memory engine so that the data can flow through the mapped graph to execute the program.

__Design Space:__ The execution paradigm discussed above can be supported by composing a set of simple and common hardware primitives, including processing elements, switches, memory address generator, scratch memory, coordination FIFOs, and a controller. All these components have different parameters, which enable tradeoffs between the degree of specialization and the hardware cost.

__Spatial Execution Model:__ The most important dimension that enables the tradeoff between generality and hardware cost is the “execution model” — when and what to perform operation, which can be together determined by the processing elements and the on-chip network.

In a static instruction scheduling [8], all the timing of data arrival should be reasoned by the compiler. In a dynamic instruction scheduling [9,10], it requires additional power/area to implement the logic for checking the operand readiness, and the flow-control on the network to dynamically balance the data traffic.

Computational and routing resources can be assigned to their dedicate instructions and data or be temporally shared. Shared resources enable higher instruction concurrency at the cost of power and area. Dedicated resources enable higher execution throughput by avoiding contention and have lower power/area overhead because of their smaller size of the instruction buffer and simpler control logic.

Ports: Coordination FIFOs are injected between dynamically scheduled elements (e.g. memory and dynamic-timing PEs) and statically schedule elements (e.g. static PEs). The purpose is to synchronize multiple inputs to enable static reasoning about the timing of all the dependent data availability.

Memories: The basic execution model of memories in decoupled-spatial execution is that memory accesses with specific patterns are encoded in coarse-grain hardware intrinsics, which we refer to as streams. The stream engine (address generator) arbitrates the concurrent stream execution.

Memories can be parameterized by their capacity, bandwidth, the number of stream entries, and the capability of capturing double buffering [11]. We currently support two memory idioms in the stream engine, linear and indirect.

linear typically refers to streams with address pointers like a[i*n+j] where i and j are loop variables of canonical loops (loop variables start with zero, and increased by one). The indirect memory address generator is similar to the memory controller in SPU [10], which is specialized for pointer expressions like a[b[i]], and we also optionally support atomic update operations by embedding a computing unit in each bank (e.g. to support a[b[i]] += 1).

Control: All the aspects mentioned above need a control input to configure coarse grain amount of work, including loading bitstream to the spatial architecture, command the memory engine for encoded memory streams, and synchronize a phase of the application. Specific to our
implementation, we use a light-weighted general-purpose RISCV core, with extended ISA as the host controller.

**Architecture Description Graph:** By composing all the components with different parameters mentioned above, we can build decoupled-spatial architectures with different tradeoffs. The composition can be naturally represented as a graph.

By representing the architecture in a graph, we no longer rely on a templated architecture, so that the flexibility of topology can also be explored for deeper degree of specialization.

We call this graph Architecture Description Graph (ADG). This graph not only represents each design point, but also serves as a hardware abstraction for the compiler.

### III. Software Stack

As mentioned above, our framework aims at generating domain-specific accelerator, but the “domain” itself is agnostic. Therefore, a general-purpose programming language (C) is preferred. This also leaves the possibility of making this programming interface as a backend for domain-specific languages. To avoid excessive compiler work over a general-purpose language, we require programmers to provide moderate hints for code transformation, by extending three pragmas:

- **#pragma dsa offload:** This pragma annotates a loop body, if this loop body is not an innermost loop, all the loops below will be fully unrolled, which indicates the annotated loop body will be offloaded to the decoupled-spatial execution. We also provide a clause `unroll(x)` to tune the resource occupation of each offloaded region. If not specified, the compiler will automatically determine the unrolling degree based on the execution frequency.

- **#pragma dsa stream:** This pragma annotates a loop body, which indicates all the memory access below are restricted. `restricted` means all the memory address expressions derived from different array pointers will not intersect.

- **#pragma dsa config:** This pragma annotates a compound statement, which indicates all the `offload` regions will be concurrent on the spatial architecture.

Below, an example of pragma annotation is shown:

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

**Listing 1:** An FIR example annotated with pragmas.

**Decouple and Analysis:** The instructions within the scope of loop body annotated by `offload` will be sliced. The computational instructions will be represented in a dependence graph and fed to a spatial mapper, and the memory pointer expressions will be fed to a memory analyzer for further optimization memory stream command code generation.

**Spatial Mapping:** The decoupled computational instructions are fed to a spatial mapper for mapping the computations on PE’s, and routing the dependences through the network, and matching the timing of data arrival when static-schedule is adopted. We adopt a stochastic search algorithm [12].

**Idiomatic and Modular Transformation:** Our compiler [13] can recognize the idiomatic program behaviors, and try to map them to the hardware specialization. Figure 3 shows generic optimizations that software can benefit from even with

---

**Figure 3:** Generic optimizations.
the very basic design, including coalescing scalar memory accesses, fusing high-dimension continuous memory streams, and generate implicit predication by leveraging the stream state.

To maintain the robustness of our compiler across any arbitrary set of hardware features, we modularize each feature-oriented transformation. Figure 4 shows transformations that require specialized features. There will always be a fallback transformation that escapes this feature to guarantee a successful compilation at the cost of performance. For example, as it is shown in Figure 4(d), indirect memory accesses, e.g. $a[b[i]]$, requires specialized address generator. If it is not available, it will fall back to scalar memory access.

This is useful for not only targeting arbitrary hardware, but also for design space transformation, which will be discussed in the next section.

Algorithms Optimization: Because of the excessive design space, the DSE can be time-consuming. The key to accelerating the algorithm is to accelerate the application recompilation.

Our design space explorer first retains all the possible versions of IR of each application with different set of transformation enabled. These versions are fed to the compiler to estimate the hardware/software affinity (explain in next paragraph) and guide the further hardware evolution. By leveraging the spatial mapping results from prior iterations, the time of re-compilation can be significantly improved. In our published work, we mainly use two techniques:

Mapping Repair: As it is shown in Figure 7, once a mapping is invalidated by hardware modification, we do not start over the remapping process. Only the portion affected by the hardware modification is re-mapped by finding a new
spot of PE and rerouting the dependences.

**Mapping Preserving:** If the hardware modification is as simple as it can be fixed by another simple empirical hardware modification, we just heuristically apply this empirical modification. Figure 6 shows two examples of empirical modifications. In general, this optimization strategy gradually replaces general-purpose components by more specialized components.

**Objective Function:** We use perf/resource as our objective function. On a synthesized ASIC, the resource is chip area, and on an FPGA, the resource is the percentage occupied over the resource available on board. Therefore, the objective function can be separated into two aspects, the performance and the hardware cost. It is impractical to rerun all the compiled applications and re-synthesize the candidate hardware every iteration. Therefore, we use analytical models.

To estimate the performance, we first need to understand that the main source of acceleration achieved by the decoupled-spatial paradigm is the instruction-level parallelism achieved by the spatial data path. Therefore, we estimate the instruction parallelism by

\[
\text{ILP} = \#(\text{Instructions Offloaded}) \times (\text{Data Availability})
\]

We can get the number of instruction offloaded by simply counting the number of instructions fed to the spatial mapper. To get the data availability ratio, our compiler first analyzes each data source and data reusability of each operand, including the reusability captured by scratchpad, cache hierarchy, and ports. By counting the number of bytes required from each data source to sustain the computation, and dividing it by the reusability, we get the data availability ratio.

Besides the performance, another aspect is the memory hardware cost. For both FPGA and ASIC design, we synthesize hardware components alone with different parameters and collect the resource consumption. After collecting enough data, we use these synthesized results to train a regression model to infer the resource occupation of each component. Finally, to estimate the hardware cost of a candidate design, we sum up each single component resource occupation in the ADG.

**V. Implementation**

Our implemented research infrastructures include three aspects:

- **Functional Simulator:** To rapidly check our newly proposed hardware specialization, we extend a GEM5 RISCV simulator with our decoupled-spatial ISA and integrate a spatial simulator to it.
- **Compiler:** We extend a clang frontend to parse the extended pragmas and encode them in metadata, and a LLVM pass for decoupled-spatial transformation by taking advantage of this additional information. We extend the RISCV-GNU-GCC to integrate our decoupled-spatial ISA.
- **Spatial Mapper:** We implemented a stochastic and heuristic-based spatial mapper.
- **Hardware RTL:** We implemented a Scala-embedded DSL to describe the underlying hardware and generate the RTL in Chisel. The generated Chisel implementation can be integrated to ChipYard ROCC for a full-system implementation.
- **Workloads:** All the applications can be either programmed by writing high-level language and annotating pragmas or directly writing embedded assembly code.

All our research infrastructure discussed above are available at https://github/polyarch/dsa-framework. We held tutorials on MICRO-2020, and will hold on MICRO-2022.

**VI. Usability Demonstration**

In this section, we demonstrate the usability of each aspect of the research infrastructure discussed in the last section.

**Compiler:** To demonstrate the effectiveness of our compilation approach and the effectiveness of the transformations, we first pick 23 workloads from MachSuite [14], Xilinx Vitis, and our prior DSP workload implementation [15]. We handcrafted a general-purpose decoupled-spatial accelerator, and then incrementally enable each compiler optimization to see the effects shown in Figure 8a. The speedup is normalized over an AMD EPYC 7702 CPU with gcc -O3 enabled. Base optimization only decouples the memory operations and the computational instructions, and most applications already benefit from this. Generic optimizations coalesce adjacent
scalar memory operations into one stream, so applications like fft, blur, and gray scale benefit. Temporal, dynamic, and indirect optimizations are hardware feature oriented optimization for specific program idioms. QR, and Cholesky have program regions with different execution frequency benefit from temporally shared PEs. Merge sort consumes data in a data-dependent rate, so it benefits from dynamic timing. CRS, Ellpack, and radix sort all have indirect memory access, so they benefit from indirect optimization.

**Functional Simulator:** To demonstrate how the simulator helps to find the performance bottleneck, we extract the cycle breakdown of each application with all the optimization enabled, which is shown in Figure 8b. The number above each bar is the instruction parallelism on the spatial architecture.

**Design Space Exploration & H/w Generation:** To demonstrate the effectiveness of the DSE as well as the quality of the generated hardware, we stress our research infrastructure over 19 workloads. We ruled out 4 workloads, nw, viterbi, merge sort, and radix sort, from MachSuite, because of their poor scalability on multicore system. The performance speedup over state-of-the-art HLS, AutoDSE, is shown in Figure 9.

The general-OG bar is a four-core handcrafted general-purpose accelerator, which already achieves near performance against AutoDSE. With suite-wised DSE, our generated overlay can outperform AutoDSE by 1.4×. With application-specific DSE, our generated overlay can outperform AutoDSE by 1.65×.

**Summary:** We can effectively conduct architecture design space exploration experiments on this infrastructure.

**VII. Future Work**

We are currently actively improving both the design space and the software stack. Here, we discuss two direction.

**Core-wise Heterogeneity:** Core-wise heterogeneity is a widely adopted technique on SoCs and desktop CPUs. Currently, our DSE only supports homogeneous system — each core is identical to others, which limits further degree of specialization. For example, by giving a universal set of applications, it is highly desired for the DSE to intelligently separate them into several subsets with different degree of specialization and generate multiple heterogeneous cores.

**Spatial Virtualization:** It has long been claimed that reconfigurable domain-specific accelerators are good alternations for unifying the ocean of ASICs on a SoC. However, unlike ASICs, all the resources are dedicated to the applications for which they are designed, hardware and software support for sharing resources on spatial architectures across applications are still missing. We suggest virtualization techniques should be introduced to isolate resources across applications, and scheduling, context switching should also be considered.

**References**


