Unifying Spatial Accelerator Compilation with Idiomatic and Modular Transformations

Jian Weng

Sihao Liu

Dylan Kupsh

Tony Nowatzki
University of California, Los Angeles

***Abstract*—Spatial accelerators provide high performance, energy efficiency, and flexibility. Recent design frameworks enable these architectures to be quickly designed and customized to a domain. However, constructing a compiler for this immense design space is challenging, first because accelerators express programs with high-level idioms that are difficult to recognize. Second, it is unpredictable whether certain transformations are beneficial or will lead to infeasible hardware mappings. Our work develops a general spatial-accelerator compiler with two key ideas. First, we propose an approach to recognize and represent useful dataflow idioms, along with a novel idiomatic memory representation. Second, we propose the principle of modular compilation, which combines hardware-aware transformation selection and an iterative approach to handle uncertainty. Our compiler achieves 2.3$\times$ speedup, and 98.7$\times$ area-normalized speedup over high-end server CPU.**

Introduction

Moore's law has driven the advancement of general-purpose processors for decades, but the once exponential gains are waning. Reconfigurable spatial accelerators (e.g. CGRA's, reconfigurable dataflow processors, systolic arrays) are promising due to their flexibility and order-of-magnitude performance and energy efficiency. Several recent design frameworks enable spatial accelerators to be specified as a composition of simple primitives [3], [7], [14], [1], [12]. This approach allows programmers to target many domain-customized programmable accelerators with a unified software framework.

However, a significant obstacle remains: how to design a compiler that is robust to arbitrary hardware feature combinations that might be present in any given accelerator instance? The large gap between high-level language and idiomatic accelerator interfaces compounds the problem.

There are two key challenges that form the basis of our novel approach:

Idiomatic Dataflow: For efficiency, accelerators use idiomatic ISA's, which encode coarse-grain, stateful patterns of program behavior for memory, control, and parallelism (e.g. a 2D memory access pattern). A generalized compiler must be able to recognize corresponding idioms from a general high-level representation.

Our work develops a novel dataflow-based intermediate representation (IR) which can explicitly express commonly accelerated idioms. We then develop a series of compiler transformations to identify and optimize for important memory and control idioms.

Modular Compilation: Any sufficiently complex accelerator design space will contain many combinations of features that imply different preferred sets of compiler transformations – either because a certain transformation hurts the performance, or makes a compilation infeasible to hardware. There are many combinations of transformations, especially when multiple code regions compete for resources.

To address this, we develop a modular compilation approach. The basic principle is to start with the most aggressive feasible set of transformations, and iteratively relax the transformations based on the compiler-estimated performance reduction.

This work's contribution is to further develop several aspects spatial accelerator compilers:

- Recognizing a set of primitive program behaviors, *idioms*, which benefit from hardware specialization and exposure in an accelerator ISA.
- Building a structured intermediate representation to record these behaviors and ease further optimization and code generation.
- Proposing a systematic way to determine the optimal set of compilation transformations for an accelerator with modular features.

Implementation and Key Results:

Our compiler is integrated into the DSAGEN framework [14]. By extending Clang/LLVM, we build a C+pragma compilation flow for spatial architectures. According to our evaluation, our compiler robustly targets accelerators specialized for 3 different domains, achieving 2.3× speedup and 98.7× area-normalized speedup, comparing with a single core of AMD EPYC 7702P. The compiler, simulator, RTL generator, and benchmark implementations are available at https://github.com/polyarch/dsa-framework.

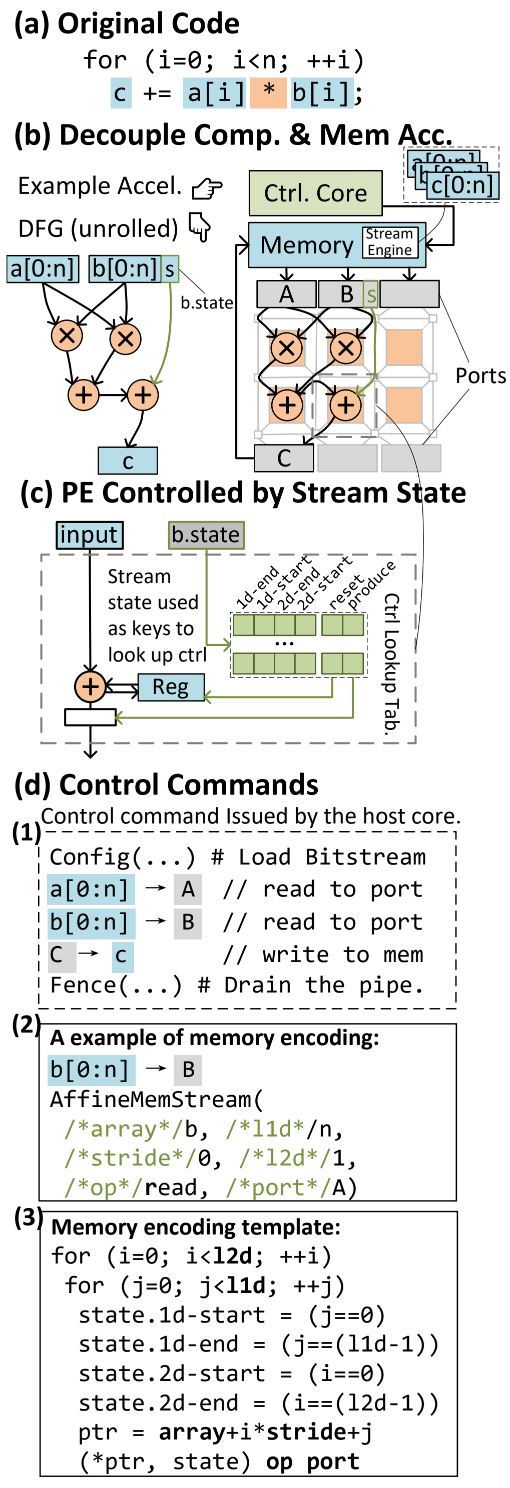
This article first discusses the spatial architecture design space, followed by an explanation of our novel compilation approaches, before compiler evaluation and related work discussion.

Spatial Accelerator Design Space

A Decoupled-Spatial Execution Model

Our target accelerator paradigm is "decoupled-spatial". *Decoupled* indicates how computation and memory operations are executed on different specialized hardware components. *Spatial* indicates how the low-level details of the execution (e.g. instruction placement, operand routing) are exposed in the ISA.

In this model, programs are represented as a decoupled dataflow graph (DFG). Figure 1(a) and (b) show an example DFG with "stream" nodes (e.g. `a[0:n]`) to represent coarse-grain memory patterns, while other nodes represent computation. Figure 1(b) also shows an example of the mapping to an accelerator, composed of scratchpad memory, a compute substrate, and port-based interface. We next elaborate on each aspect of the execution model.



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

```
Config(...) # Load Bitstream
a[0:n] → A  // read to port
b[0:n] → B  // read to port
C → c       // write to mem
Fence(...) # Drain the pipe.
```

A example of memory encoding:

```
b[0:n] → B
AffineMemStream(
 /*array*/b, /*l1d*/n,
 /*stride*/0, /*l2d*/1,
 /*op*/read, /*port*/A)
```

Memory encoding template:

```
for (i=0; i<l2d; ++i)
 for (j=0; j<l1d; ++j)
  state.1d-start = (j==0)
  state.1d-end = (j==(l1d-1))
  state.2d-start = (i==0)
  state.2d-end = (i==(l2d-1))
  ptr = array+i*stride+j
  (*ptr, state) op port
```

Figure 1: Mapping dot product onto decoupled-spatial paradigm, highlighting idiomatic hardware/ISA features.

Idiomatic Memory Representation Common memory access patterns are represented as first-class primitives in the ISA for *lean and high-throughput* memory request generation pipelines (called stream engines).

Figure 1(b) shows that memory accesses are represented in coarse-grain streams and executed on the stream engine. The streams communicate with computational instructions through "ports", named FIFO's that enable coordinated dataflow firing of computation inputs. This coordination is required for supporting statically-scheduled components. Streams for different ports are defined to be concurrent (unless there is an intervening fence, explained later).

Besides data, the encoded streams can also communicate their loop *state* to compute instructions, which indicates the beginning/end of loop dimensions, and can be useful for control flow (explain later).

Compute Substrate: Spatial Dataflow Instructions are mapped to processing elements (PEs), and the dependences are routed by a network of switches (small circles in Figure 1(b)). Logically, instructions will be fired when their inputs have arrived.

Besides computation, each PE can optionally have a control lookup table. This table is indexed either by least significant bits of the instruction output or additional input, and determines whether to reuse one of the inputs, discard an output, or reset an accumulator. In Figure 1(b)&(c), the stream state is used as the control lookup key for both resetting the accumulator on loop termination, and discarding intermediate values before then.

Control Core: Program phases are coordinated by the control core. Figure 1(d.1) shows the control commands issued by the control core, including loading configuration bits, memory streams encoded in ISA, and a fence. Fence instructions synchronize with the accelerator at the boundary of program phases. Streams after the fence are stalled until all prior streams are completed. Figure 1(d.2&3) concretely shows how memory streams are encoded and executed.

Control commands are executed within a typical Von Neumann program so that program aspects which are not accelerator-friendly can be handled by the lightweight control core.

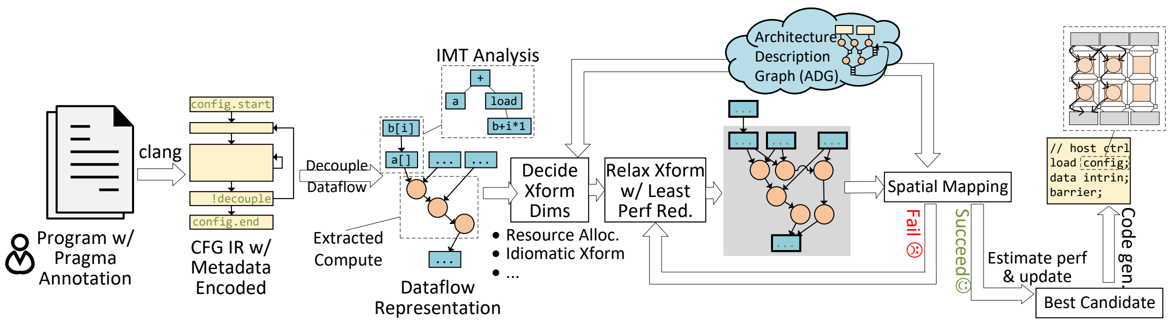


Figure 2: An overview of the compiler.

Design Space

There are three key aspects to the design space, which would be determined either through manual design or by a design-space exploration (DSE) algorithm [14]: the parameters, topology, and capabilities.

Parameters Parameters are the tunable aspects of the hardware, like scratchpad size or PE register count. These could impact the performance/area tradeoffs, and can affect whether an optimized program can legally map to hardware.

Topology This defines the set of components in the accelerator (PEs, memories, ports, network switches) and their connectivity. The topology is a key factor in determining both the maximum performance and the generality; for example, a mesh-based network would be more flexible than an application specific network.

Capabilities Capabilities define optional idiomatic functionality exposed in the ISA. Instantiating expensive features only when necessary is helpful to limit the hardware cost. We discuss three important capabilities:

Linear vs Indirect Memory Access: The memory stream engine can be specialized for either linear (e.g. `a[i*n+j]`) or indirect (e.g `a[b[i]]`) memory access. Supporting parallel indirect memory access requires significant power/area to manage bank conflicts and reorder requests [4].

Dynamic vs Static Scheduling: In static scheduling, the order and cycle-level timing of PE's and routers is defined statically by the compiler, whereas in dynamic scheduling the execution is triggered by data arrival. Dynamic scheduling requires more power/area, and static scheduling saves power/area at the expense of flexibility.

Dedicated vs Shared Execution: Dedicated components only support one instruction (or route), whereas shared components are temporally multiplexed. For a given number of PE's (i.e. maximum throughput), dedicated components have lower power/area overhead, whereas shared components enable mapping larger and more complex program regions.

Architecture Description Graph (ADG) The ADG combines topology, capabilities, and parameters into a unified, graph representation that defines the accelerator ISA. It includes nodes for PE's, switches, memories/address generators, and ports for synchronization. The ADG is not only a hardware specification for RTL generation, but also a capability abstraction to the compiler.

Compiler Overview

Figure 2 shows an overview of the compiler. The programming interface is pragma-annotated C for aiding alias analysis. The annotated code is analyzed and transformed into a decoupled dataflow intermediate representation (IR): The computational instructions are represented in a dependence graph, and the memory operations are analyzed and represented in an idiomatic memory tree (IMT) form. The compiler will iteratively explore the combination of transformations for the best software/hardware affinity. Finally, the compiler performs code generation on the selected transformations and removes accelerator-mapped instructions from the control program.

We next briefly highlight key aspects of the compiler.

Pragma Annotation We rely on two pragmas to aid compilation, as shown with a simple example:

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

`#pragma dsa config` This defines the set of concurrent program regions (loop nests and compound statements) on the spatial architecture.

`#pragma dsa decouple` This indicates all memory accesses under this loop level are "restricted", i.e. no address intersection among arrays with different pointers. This completely avoids enforcing load-store ordering explicitly, expensive for dataflow architectures.

Dataflow Decoupling This pass decouples computation and memory access within the scope of `config` pragma and builds a decoupled dataflow IR for idiomatic analysis and transformation. Using slicing [8], operands that are transitively dependent on address operands of memory instructions are considered part of address generation, and remaining instructions are computation. The pointer expressions of the memory instructions will be fed to idiomatic memory analysis for aiding stream ISA encoding, while the computation is represented in dataflow form for spatial accelerator mapping. Private arrays with a known size are considered to map to scratchpad.

Transformation Space Exploration To determine the optimal set of transformations – termed a transformation point – our compiler first determines all the tunable dimensions of the transformation space based on the IR and ADG, including the possible unrolling degrees and the profitable idioms. Then our compiler tries mapping different transformation points to hardware through *spatial mapping*, which will determine its feasibility and predicted performance. We iterate over the transformation points in order of decreasing expected performance, so that we can break early and avoid unnecessary and expensive spatial scheduling iterations.

Spatial Mapping Spatial mapping is responsible for mapping instructions and memory streams onto hardware units, routing dependences onto the network, and matching the timing of operand arrival for statically-scheduled components (either by path lengthening or using delay FIFO's [10]). We adopt a stochastic search algorithm [10], [9].

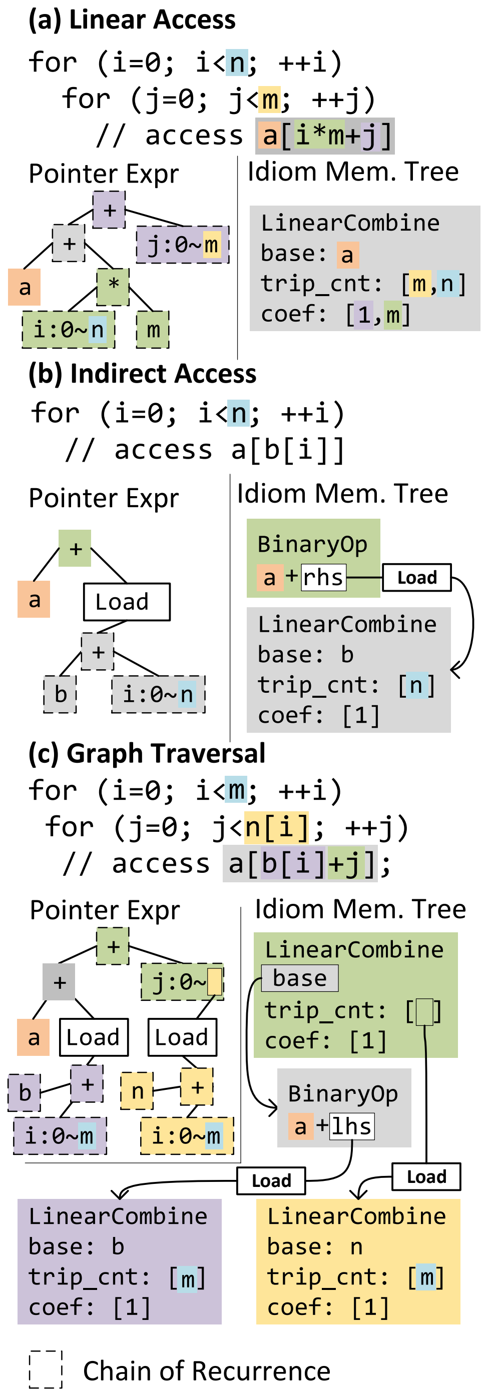


Figure 3: An example of analyzing memory access pattern and control command generation.

Idiomatic Transformation

The compiler for an idiomatic ISA must find the best-suited idiom that is applicable to a program region. Here, we explain a set of broadly applicable idiom transformations for memory access and computation.

Decoupled Memory Access

Expressing memory in terms of predefined coarse grain idioms — i.e. streams — has many benefits, including better-utilizing memory bandwidth and reducing control instructions and core-accelerator communication. To aid mapping/optimization on streams, we develop the idiomatic memory tree (IMT).

Idiomatic Memory Tree Prior works on program idiom analysis under loops, like chain of recurrence (CR)[1][6], mainly focus on analyzing the expressions of loop variables and invariants, and have limited support on memory-dependent expressions.

To store and analyze the idiomatic memory behaviors in a structured way, our insight is that complicated program behaviors can be composed by a set of simple primitive idioms. To explain, Figure 3(c) shows a complicated graph traversal example. This program behavior is composed of an affine pattern in the inner loop and an indirect pattern in the outer loop. Next, we introduce IMT nodes that capture primitive behaviors.

LinearCombine Figure 3(a) shows an example of linear memory accesses. Dashed boxes annotate the sub-expressions that can be analyzed by CR. By walking through these expression nodes, we can extract the coefficient of each loop variable and represent the pointer expression, `a[i*n+j]`, in a linear combination format:

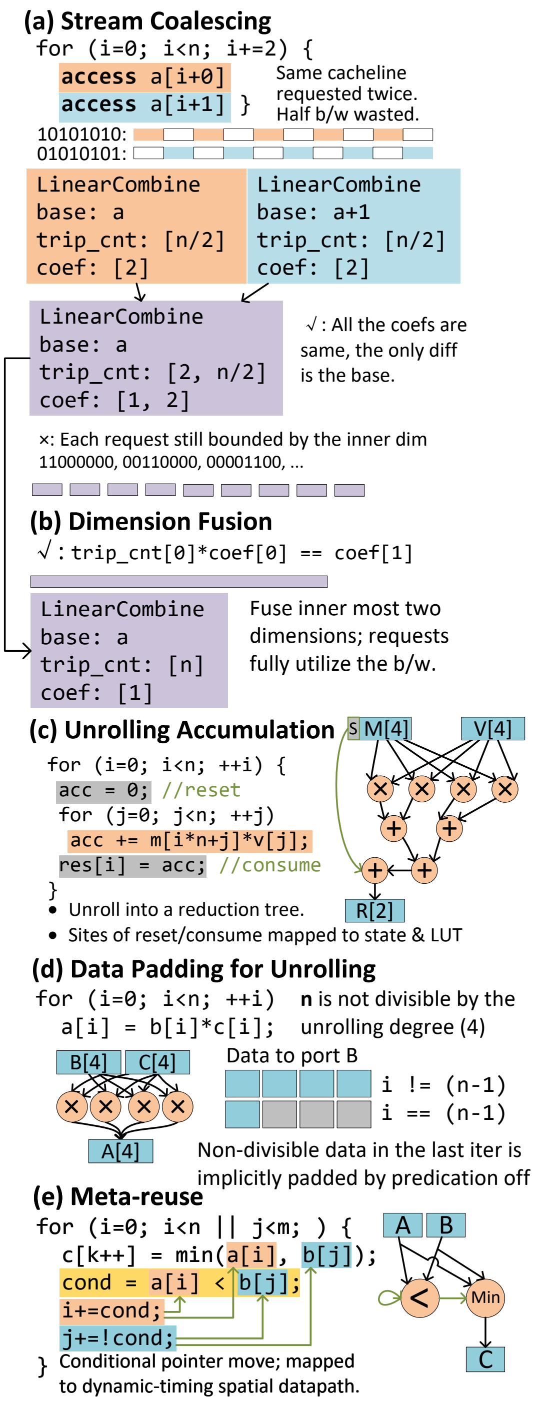
$$\sum_k \mathrm{i}_k \times \mathrm{coef}_k + \mathrm{base}$$

BinaryOp Figure 3(b) shows a pointer expression (`a[b[i]]`) without loop variable directly involved in the operands, which cannot be analyzed by CR. Thus, we use a `BinaryOp` node to wrap this node and recursively analyze both operands.

Load Continuing with the example shown in Figure 3(b), a memory load is involved on the rhs operand, which indicates an indirect memory operation. We wrap the memory load with a `Load` node, and recursively analyze the pointer expression of this memory load. In this case, the load pointer can be handled by CR, and yields a `LinearCombine` node.

The IMT is useful for generic optimizations (explained next) and to choose the right set of idioms for code generation.

[1]Our implementation uses LLVM's ScalarEvolution for CR.


(a) Stream Coalescing
Same cacheline requested twice. Half b/w wasted.
10101010:
01010101:
LinearCombine base: a trip_cnt: [n/2] coef: [2]
LinearCombine base: a+1 trip_cnt: [n/2] coef: [2]
LinearCombine base: a trip_cnt: [2, n/2] coef: [1, 2]
√: All the coefs are same, the only diff is the base.
×: Each request still bounded by the inner dim 11000000, 00110000, 00001100, ...
(b) Dimension Fusion
√: trip_cnt[0]*coef[0] == coef[1]
LinearCombine base: a trip_cnt: [n] coef: [1]
Fuse inner most two dimensions; requests fully utilize the b/w.
(c) Unrolling Accumulation
S
M[4]
V[4]
R[2]
Unroll into a reduction tree.
Sites of reset/consume mapped to state & LUT
(d) Data Padding for Unrolling
n is not divisible by the unrolling degree (4)
B[4]
C[4]
A[4]
Data to port B
i != (n-1)
i == (n-1)
Non-divisible data in the last iter is implicitly padded by predication off
(e) Meta-reuse
A
B
<
Min
C
Conditional pointer move; mapped to dynamic-timing spatial datapath.

```
for (i=0; i<n; i+=2) {
  access a[i+0]
  access a[i+1] }
```

```
for (i=0; i<n; ++i) {
  acc = 0; //reset
  for (j=0; j<n; ++j)
    acc += m[i*n+j]*v[j];
  res[i] = acc; //consume
}
```

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

```
for (i=0; i<n || j<m; ) {
  c[k++] = min(a[i], b[j]);
  cond = a[i] < b[j];
  i+=cond;
  j+=!cond;
}
```

Figure 4: Generic program idioms

Fusing Stream Idioms Because memory requests are issued at line granularity, stride-access wastes bandwidth. We address this through two transformations:

Stream Coalescing Figure 4(a) shows if we generate streams for `a[i]` and `a[i+1]` separately, these two streams will request each cacheline twice. Streams which have the same coefficients, and where the base differs by one, are coalesced by appending a new dimension (see purple box, Figure 4(a)).

Dimension Fusion Continuing with Figure 4(b), after appending a new dimension, two 1-d streams become a 2-d stream with two continuous dimensions. Therefore, our compiler will fuse subsequent stream dimensions when their outer dimension's coefficient equals the coefficient times the trip count of the inner loop.

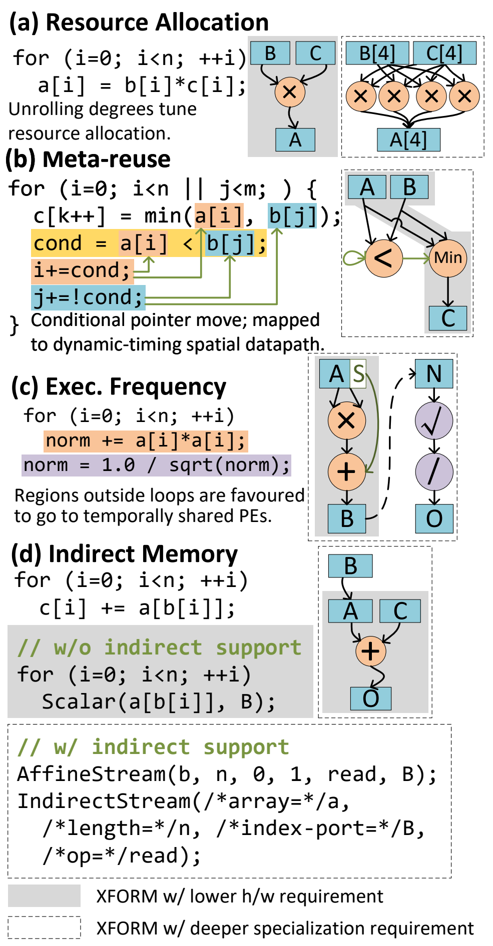
In the code generation phase, our compiler matches the analyzed and optimized IMT on idiomatic ISA to fill in parameters. A `LinearCombine` node indicates affine memory access, so it encodes as many as possible linear dimensions supported on the hardware. A `BinaryOp` node with a `Load` involved indicates an indirect memory access, so the compiler extracts the

Computational Idioms

Accumulator Accumulation manifests as a loop carried dependence that involves itself — see Figure 4(c). To specialize for this idiom, the intermediate accumulated results can be stored implicitly in the instruction (later allocated to a PE register), and the data output/accumulator-reset is controlled by a stream *state* metadata of an operand stream.

Data Padding As shown in Figure 4(d), the trip count of the innermost dimension can be indivisible by the unrolling degree, which normally requires loop peeling for the final iterations. Our compiler specializes for idiom by generating memory streams with different padding flags according to their consumers. For elementwise operations, stream data is padded to fill with invalid data to predicate off the unused datapath. For accumulator operations, stream data is padded with zeros to avoid adding extra control/muxing.

Meta-reuse Figure 4(e) describes an algorithm for merging two lists, by repeatedly increasing the iterator of the smaller value. To specialize for this, input elements elements are popped/reused according to the result of comparison. This shortens the dependence chain on control, but causes data-dependent consumption rate on the spatial architecture; thus, dynamic-scheduled hardware elements are required.


(a) Resource Allocation
Unrolling degrees tune resource allocation.
B
C
A
B[4]
C[4]
A[4]
(b) Meta-reuse
Conditional pointer move; mapped to dynamic-timing spatial datapath.
A
B
Min
C
(c) Exec. Frequency
Regions outside loops are favoured to go to temporally shared PEs.
A
S
B
N
O
(d) Indirect Memory
B
A
C
O
XFORM w/ lower h/w requirement
XFORM w/ deeper specialization requirement

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

```
for (i=0; i<n || j<m; ) {
  c[k++] = min(a[i], b[j]);
  cond = a[i] < b[j];
  i+=cond;
  j+=!cond;
}
```

```
for (i=0; i<n; ++i)
  norm += a[i]*a[i];
norm = 1.0 / sqrt(norm);
```

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

```
// w/o indirect support
for (i=0; i<n; ++i)
  Scalar(a[b[i]], B);
```

```
// w/ indirect support
AffineStream(b, n, 0, 1, read, B);
IndirectStream(/*array=*/a,
  /*length=*/n, /*index-port=*/B,
  /*op=*/read);
```

Figure 5: Modular transformations with fallbacks.

Modular Compilation

To stay robust across accelerators, we use modular transformations with fallbacks for each idiom and develop an exploration algorithm to quickly converge to the optimal set of transformations.

Modular Transformations with Fallbacks

Fallback transformations (shadowed regions in Figure 5) either uses less resources or non-idiomatic hardware features. Here, we give four examples:

High Allocation→Low allocation Executing more loop iterations simultaneously is faster but requires more resources. Thus, a knob to resource allocation is the unrolling degree, as shown in Figure 5(a).

Meta-reuse→Host Execution Meta-reuse control requires dynamic PEs/switches which are more expensive. A fallback is to execute related instructions on the control core: the comparison, the green arrows, and the memory streams `a[i]` and `b[i]`.

Temporal PEs→Dedicated PEs/Host Execution Figure 5(c) shows that `norm` computation is out of the loop body and involves two expensive operations. Offloading instructions with lower execution frequency to dedicated PEs leads to low resource utilization, so these instructions are favored to go to temporally shared PEs. If shared PEs are not available, the compiler first falls back to dedicated PEs if enough are available, then falls back to the control core.

Indirect Memory→Scalar Memory A memory operation pointer expression involves another memory operation indicates an indirect memory access. Accelerators without indirect-memory specialization require a fallback transformation: The compiler will perform indirect accesses as a series of scalar accesses (single element stream); this requires an order-of-magnitude more core/accelerator communication. Figure 5(d) shows the different generated stream commands depending on whether indirect memory is available.

Transformation Space Exploration

As aforementioned, idioms and modular features are dimensions of a transformation space. When it comes to multiple loops and program behaviors of interest, the transformation space grows exponentially. Meanwhile, invoking the spatial scheduling algorithm is expensive, making it difficult to try every possibility. Therefore, we adopt a somewhat-greedy search algorithm. The basic steps are:

1) The compiler first determines all the explorable dimensions for each concurrently mapped program region, according to relevant transformations and the hardware capability in the ADG.
2) For each region and dimension, the compiler "relaxes" it (e.g. reduce the unrolling degree of one of the loops, or disable indirect memory encoding), and then estimate the performance reduction caused by the relaxation based on the expected ILP and memory bandwidth of any streams.
3) The transformation with the least performance reduction is applied, and the transformed IR is fed to the spatial scheduler for hardware mapping. If it fails, eliminate the transformation point and go to step 2; else, the the feasible transformation point is returned.

Because this algorithm enumerates transformation points in a roughly decreasing order of the ideal performance, the first successful mapping is likely to have the best performance. In addition, we also adopt a region-balance strategy to prune this space — resource allocations where a low execution frequency code region has a higher resource allocation than a higher-frequency code region will be skipped.

Methodology

Software Stack: The compiler is implemented by extending the Clang frontend for pragma parsing, and LLVM for ISA extension and IR transformation.

Benchmarks We select 9 from MachSuite, 9 from Xilinx Vitis, and 5 DSP workloads, each with their own prevalent program idioms. The data type, size, and computation intensity of each is shown in Table 1.

Hardware Setup We choose the *AMD EPYC 7702P* as our CPU baseline. All the benchmarks run on this are compiled by `gcc -O3`.

Accelerators are generated with DSAGEN [14]. The accelerator controller is a single-issue RISCV core with extended ISA. Both the AMD CPU and the accelerator have the same L1/L2 cache size (32KB, 512KB) and bandwidth (64B/cycle). The AMD CPU has nearly 24GB/s DRAM bandwidth, and the accelerator has 20GB/s memory bandwidth.

We start with a general accelerator with full specialized features and the spatial architecture is 5×5 (16 dedicated multiplier PE's, 8 dedicated adder PE's, one temporally shared PE with full arithmetic capability) mesh-topology.

We then use DSAGEN to auto-generate accelerator targets for each benchmark suite with three different degrees of specialization.

- *Capability Specialization (Cap)* indicates the architecture adopts all the specialized features required and a generic mesh topology. Both floating point and integer functional units are included.

Workloads	crs/ellpack	gemm	nw	stcl-2d	stcl-3d	viterbi	merge	radix
Size	496×4	64^3	128^2	34^2	34^3	140×64	2048	2048
DType	f64	i64	i64	i64	i64	f64	i64	i64
Op/DRAM	0.16	4	0.13	1.99	1.14	0.1	0.5	0.06
Feat.	Ind. Mem	Basic	Basic	Basic	Basic	Basic	Dyn. Timing	Ind. Mem

Specialization		Cap.	+FU	+Topo
Area (mm2)	FU	0.15	0.11	0.08
	SW	0.03	0.02	0.02
	Port	0.03	0.03	0.02
	Spad	0.12	0.12	0.12
	Total	0.36	0.31	0.27

(a) MachSuite

Workloads	acc	acc-sqr	acc-wei	grey	blur	cha-ext	conb	drvt	vecmax
Size					128^2×4				
DType					i16				
Op/DRAM	0.16	0.33	0.66	0.4	0.5	n/a	0.5	0.5	0.16
Feat.					Basic				

Specialization		Cap.	+FU	+Topo
Area (mm2)	FU	0.15	0.05	0.05
	SW	0.02	0.02	0.02
	Port	0.03	0.02	0.02
	Spad	0.04	0.04	0.00
	Total	0.28	0.18	0.13

(b) Vitis

Workloads	chol	fft	mm	qr	solver
Size	48^2	2048	32^3	48^2	48^2
DType	f64	f32x2	f64	f64	f64
Op/DRAM	3.07	6.8	1.33	4.08	0.24
Feat.	Shared PE	Basic	Basic	Shared PE	Basic

Specialization		Cap.	+FU	+Topo
Area (mm2)	FU	0.15	0.08	0.07
	SW	0.03	0.02	0.02
	Port	0.02	0.02	0.02
	Spad	0.04	0.04	0.04
	Total	0.29	0.20	0.19

(c) DSP

Table 1: Benchmark specification and generated hardware characteristics.

- *FU Specialization (+FU)* indicates the unused functional units will be trimmed off.
- *Topology Specialization (+Topo.)* indicates the topology (the connectivity of hardware components) is specialized to the applications.

The bottom of Table 1 shows the area breakdown of these accelerators. All these numbers are synthesized by Synopsys DC @28nm.

Simulation We develop a cycle-level simulator for performance estimation, by integrating a spatial architecture simulator to a gem5 single-issue core.

Evaluation

We evaluate the compiler's performance and robustness. The key takeaways are:

- Our compiler achieves 2.2×, 3.3×, and 1.3× speedup on the three workload suites, MachSuite, Vitis, and DSP, respectively.
- The generated binaries reduce dynamic RISCV instructions by mean 99.8% on the control core.
- Our compiler allows graceful performance degradation when compiling to accelerators with different degrees of specialization.

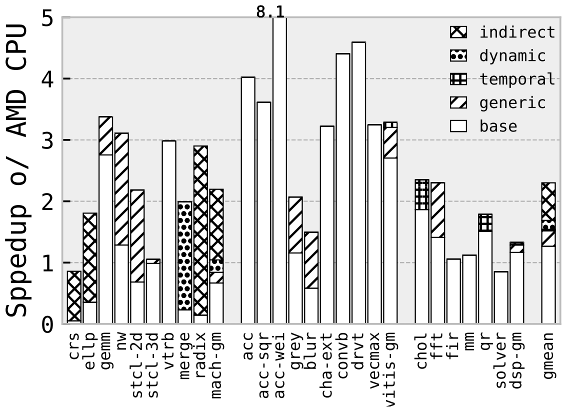
Idiomatic Transformation: Figure 6a shows the performance of each workload on the general initial accelerator when incrementally enabling optimizations. It achieves mean 2.3× speedup and 98.7× area-normalized speedup over the CPU.

Base is the code generation without any idiomatic optimization, which just transforms the program into decoupled dataflow and maps the decoupled aspects to specialized units, e.g. linear memory access to stream engine.

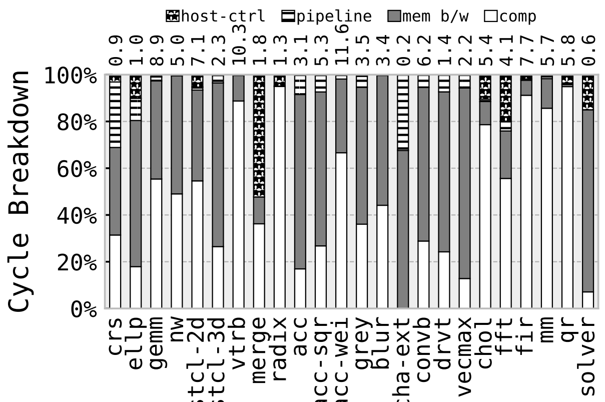
Generic refers to the stream coalescing and dimension fusion optimization. `gemm`, `nw`, `stcl-2d`, `stcl-3d`, `grey`, `blur`, and `fft` all have adjacent scalar access in the innermost loop body, thus benefiting from stream coalescing and dimension fusion. The performance of these 7 workloads is improved by mean 1.8× compared with *base* optimizations.

Temporal means offloading instructions to temporally shared processing elements. `cholesky` and `qr` both have code regions with more than one instruction outside the loop nest. Offloading these to shared PE's enables higher ILP and avoids control core serialization. Thus, their speedup is improved by mean 1.2×.

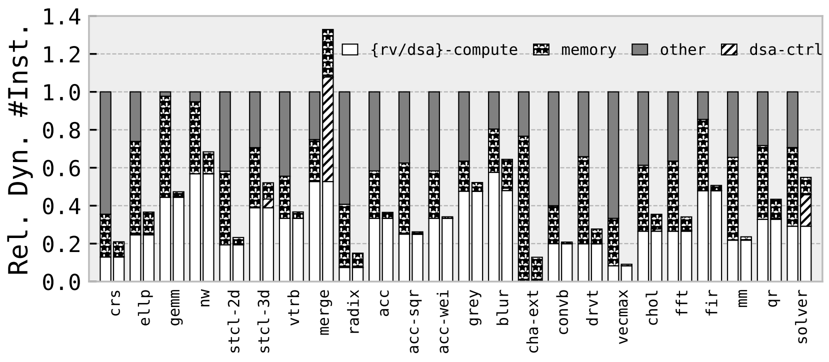
Dynamic supports the capability of conditionally popping data. Only `merge` benefits from dynamic capabilities; falling back to control instructions on the single issue control-core would



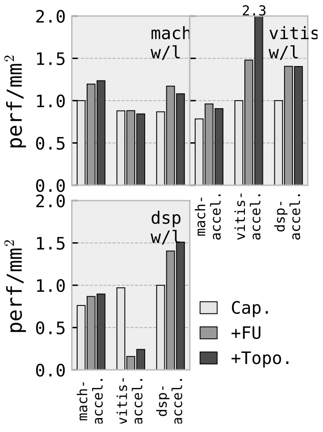
(a) Performance over CPU by each compiler transformation



(b) CPI stack of each workload; numbers on the top are IPC.



(c) Dynamic instruction reduction enabled by decoupled-spatial compilation.



(d) Perf/mm^2 normalized to capability specialized architecture.

Figure 6: Performance and specialization study

cost 8.6× speedup.

Indirect enables parallel indirect memory access, and `radix`, `crs`, and `ellpack` all benefit by 11.5× speedup over feeding indirect access data one-by-one to the accelerator.

Speedup Implication: To study the source of speed up, we demonstrate the dynamic instruction reduction and the cycle breakdown. By compiling the original C codes to RISCV and simulating on gem5, we count the dynamic instructions. Figure 6c shows that 66% of the dynamic instructions are removed, and 99.8% of RISCV instructions are eliminated.

The instructions are in four categories: computation, memory, other, and dsa control. The "memory" and "other" category constitute the biggest savings, because of the stream encoded memory. For simplicity, we compare here against a non-vectorized RISCV baseline, but idiomatic memory streams can coalesce multiple memory operations into one request; "other" instructions mainly include pointer expression and loop control: these can be expressed by encoded streams to reduce instruction count by orders-of-magnitude.

Moreover, Figure 6b shows that besides computation, memory bandwidth is the second largest portion of the execution time, which indicates that speedups are mainly from high ILP enabled by the decoupled-spatial execution.

An outlier is merge sort, which not only has 1.3× dynamic instructions, but is also bounded by control instructions. Our current idioms can only capture the inner loop of merge sort. A loop nest with affine outer loop and data-dependent inner loop is not supported, and is a candidate for broadening the idioms.

Robustness over Specialization: We demonstrate the compiler's robustness on accelerators for three domains with different degrees of specialization.

Figure 6d shows the relative perf/mm^2 normalized by the capability-specialized accelerator. Our compiler can robustly target architectures with different feature combinations while exploiting the hardware/software affinity. Performance is retained on the designs specialized for the target domain. Topology-specialized have reduced area at the expense of performance on non-targeted domains.

Related Work

Idiomatic ISAs Prior idiomatic ISA constructs include streams [11] and streams+vectorization [5]. Prior idiomatic spatial architecture ISA's include database [13] and sparse processing primitives [4].

Accelerator Compilers Prior works developed general-purpose compilers for accelerators. DySER's compiler [8] developed a dataflow representation (AEPDG) which separates memory and computation. SARA [16] parallelizes sequential programs across spatial accelerator tiles. ParallelXL's compiler [2] targets general-purpose dynamic task scheduling. However, these compilers are for an accelerator with fixed capabilities.

Spatial Architecture Design FAST [15] incorporates loop reorganization into the DSE for ML-accelerators. CGRA-ME [3] and SNAFU [7] are CGRA generation frameworks that are flexible across topologies and resource allocation. REVAMP [1] and AURORA [12] have automated DSE for spatial architecture parameters (but not topology or capabilities).

High-Level Synthesis Vivado HLS also adopts a C+pragma programming interface, bit its pragmas are more complex. The key difference is that HLS generates a fixed hardware design for a single application.

Our modular compilation approach enables robustness across a significantly larger design space of programmable architectures, including different topologies and parameters.

Conclusion

This work identifies and takes on two critical problems of the accelerator era: How to compile for increasingly complex accelerator ISA's, and how to create a generalized compiler across a wide design space of accelerator capabilities and resource allocations. Idiomatic datfalow IR and transformations, combined with their modular application, will be key to supporting robust compilation in future accelerator design frameworks. We envision that this capability can enable automated exploration of accelerator generality versus specialization for highly-heterogeneous accelerator-rich architectures.

REFERENCES

1. Thilini Kaushalya Bandara, Dhananjaya Wijerathne, Tulika Mitra, and Li-Shiuan Peh. REVAMP: a systematic framework for heterogeneous cgra realization. In *ASPLOS*, 2022.
2. Tao Chen, Shreesha Srinath, Christopher Batten, and G. Edward Suh. An architectural framework for accelerating dynamic parallel algorithms on reconfigurable hardware. In *MICRO*, 2018.
3. S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson. CGRA-ME: a unified framework for cgra modelling and exploration. In *ASAP*, 2017.
4. Vidushi Dadu and Tony Nowatzki. Towards general purpose acceleration by exploiting common data-dependence forms. In *52nd MICRO*, 2019.
5. Joao Mario Domingos, Nuno Neves, Nuno Roma, and Pedro Tomás. Unlimited vector extension with data streaming support. In *ISCA*, 2021.
6. Robert Engelen. Symbolic evaluation of chains of recurrences for loop optimization. 03 2000.
7. Graham Gobieski, Ahmet Oguz Atli, Kenneth Mai, Brandon Lucia, and Nathan Beckmann. SNAFU: an ultra-low-power, energy-minimal cgra-generation framework and architecture. In *ISCA*, 2021.
8. Venkatraman Govindaraju, Tony Nowatzki, and Karthikeyan Sankaralingam. Breaking simd shackles with an exposed flexible microarchitecture and the access execute pdg. In *PACT*, 2013.
9. B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. *IET-CDT*, 2003.
10. Tony Nowatzki, Newsha Ardalani, Karthikeyan Sankaralingam, and Jian Weng. Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign. In *27th PACT*, 2018.
11. Fabian Schuiki, Florian Zaruba, Torsten Hoefler, and Luca Benini. Stream semantic registers: A lightweight

risc-v isa extension achieving full compute utilization in single-issue cores. *IEEE Trans. Comput.*, 2021.

12. Cheng Tan, Chenhao Xie, Ang Li, Kevin J Barker, and Antonino Tumeo. Aurora: Automated refinement of coarse-grained reconfigurable accelerators. In *DATE*, 2021.
13. Matthew Vilim, Alexander Rucker, Yaqi Zhang, Sophia Liu, and Kunle Olukotun. Gorgon: Accelerating machine learning from relational data. In *ISCA*, 2020.
14. Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. DSAGEN: synthesizing programmable spatial accelerators. In *ISCA 2020*, 2020.
15. Dan Zhang, Safeen Huda, Ebrahim Songhori, Kartik Prabhu, Quoc Le, Anna Goldie, and Azalia Mirhoseini. A full-stack search technique for domain optimized deep learning accelerators. In *ASPLOS*, 2022.
16. Yaqi Zhang, Nathan Zhang, Tian Zhao, Matt Vilim, Muhammad Shahbaz, and Kunle Olukotun. Sara: Scaling a reconfigurable dataflow accelerator. In *ISCA*, 2021.