The exponential growth of computational power has transformed every aspect of society. However, performance scaling for general-purpose processors has plateaued; while transistor counts have kept growing, the fraction of a chip that can be kept active is limited by power constraints. These trends have lead to a Cambrian explosion in the design and use of *specialized accelerators* which are customized to particular applications and use domain-specific programming models. However, the deployment of accelerators into the computational stack is limited by tools designed for traditional, general-purpose processors.

Hardware description languages (HDLs) operate with circuit-level abstractions insufficient for rapid and productive design, and most accelerators expose primitive, low-level programming abstractions. These specialized programming models are only accessible to experts. I will design the next generation of programming systems that make it dramatically easier to design and use customized hardware. I will **address these challenges by developing novel techniques built upon foundational ideas** in programming languages and compilers and **ground them in the challenges of building real, end-to-end hardware-software systems**. My work seeks to address the fundamental tension between high-level abstractions—which offer composability and productivity—and low-level abstractions—required for precise control and integration—through the following principles:

1. **Reason about target constraints**: High-level programming models lose performance by hiding constraints about target abstractions. Can we expose these constraints while retaining our abstractions?
2. **Build reusable infrastructure**: Efficiently mapping high-level languages to low-level abstractions requires deep domain expertise. Can we centralize this expertise by building reusable infrastructures?
3. **Enrich low-level abstractions**: Low-level languages offer efficiency but use behaviors that leak across abstraction boundaries. Can we design *efficient* and *correct* interfaces to encapsulate these behaviors?

My PhD work has explored these principles by building **programming systems for hardware design**. I have identified problems with existing, high-level programming models for hardware design and remedied them [1, 2]. I have built a **widely-used compiler infrastructure** for transforming high-level languages to efficient hardware designs [3,4,5], and designed a new hardware description language that enable efficient, modular composition of circuits [6, 7]. In recent collaborations, I have explored the problem of **building new programming models for custom hardware** [8, 9].

Reasoning about Target Constraints

High-level synthesis (HLS) tools raise the level of abstraction in hardware design by compiling languages such as C and C++ into circuit descriptions. Such tools abstract the physical constraints of hardware generation, such as physical memories only allowing a single operation every cycle, and allow users to control hardware generation using code annotations. However, this leads to an *unpredictable* programming model: applying an "unroll" annotation to a loop will allocate more resources in the circuit, but there will be no performance benefit because all memory accesses are serialized.

My work on Dahlia [1] identified the problem: HLS tools **abstract away important target constraints**; specifically, they abstract away the limitations of low-level memories. Sub-structural type systems—which can express constraints such as "a variable may only be used once"—seem like a good fit to reason about such memory constraints. However, existing type systems could not reason about *temporal reuse* of hardware resources; while a memory may only allow a single operation in a given cycle, it can be used again in the next cycle. I built a new type system to reason about temporal reuse and created a programming model where well-typed programs have *predictable* resource-performance trade-offs: a program that uses more resource has better performance. Dahlia was highlighted at the **MIT PL review**: *"[Dahlia] demonstrates the potential of programming languages methods in [high-performance computing] spaces to* ***not only reason about correctness but also performance****"*.

This insight—that memory constraints dramatically affect the HLS programming model—has enabled the design of new abstractions. In order to parallelize computations, HLS compilers need to statically predict the memory access patterns; while sufficient for dense computations, such as linear algebra, this limitation means that important computations, such as graph processing, are hard to support. In our recent work [2], we have redesigned the memory abstractions used by HLS tools. Instead of simple, software-like arrays, we use a new *memory interface language* to *declaratively* specify memory interfaces: latency requirements, physical partitioning for parallelism, managing conflicting accesses. The tool then automatically co-optimizes and generates the memory and compute implementations.

Finally, reasoning about target constraints has also proven useful for **programming energy efficient processors**. Such processors eschew the complex circuitry needed to automatically extract parallelism from programs; instead users must explicitly specify parallelization and data reuse using low-level, domain-specific instructions such as *vector shuffles*. With Facebook Reality Labs and collaborators, I designed a compiler [8] that transforms programs using equality saturation (a technique for efficiently rewriting programs) to automatically expose opportunities for data reuse and generates low-level implementations that match performance of expert-written code.

Building Reusable Infrastructures

Tools like HLS and Dahlia raise the level of abstraction but compiling them to efficient circuits requires deep expertise and technical investment. Furthermore, for each new tool, this effort must be repeated. Modular compiler infrastructures, such as LLVM, use *intermediate languages* to **centralize the work of analyzing, optimizing, and compiling programs** to multiple targets. Building such a modular infrastructure to compile high-level languages to efficient circuits brings its own set of unique challenges: (1) circuits must be optimized for physical objectives, such as power and resource usage, in addition to latency and throughput, (2) different targets, like ASICs and FPGAs, require different optimization trade-offs, and (3) optimizations must exploit the extra structure available in designs generated from high-level programs.

I designed the Calyx infrastructure [3], **a modular, reusable infrastructure for accelerator generators**. Calyx uses a novel intermediate language that intermixes software-like control operators (loops, conditionals, sequencing), with HDL-like structural abstractions such as circuit instantiation and wire connections. This representation enables Calyx's pass-based compiler optimize to designs using control-flow information present in circuits generated from high-level languages. Calyx has been **adopted by the LLVM CIRCT project** [10] as a core dialect, enabling compilation of PyTorch and C++ programs to hardware. It has been used to **design several frontends and accelerator generators**: Halide [11], TVM [12], AMC [2], systolic arrays [13], pangenomics computations [14], and polynomial approximations [15]. Finally, Calyx is a successful open-source project with more than a dozen contributors, **300 stars on GitHub**, and is the primary subject of a **submitted NSF Pathways to Enable Open-Source (POSE) grant** that I helped write.

Reusable infrastructures like Calyx **magnify the impact of each research project**: tools built using Calyx are immediately accessible to the rest of the ecosystem. For example, we designed a debugging infrastructure for Calyx [5] (**distinguished artifact award**) which provides software-like debugging abstractions—breakpoints, source-level stepping—to any frontend that compiles to Calyx. Calyx users in the LLVM CIRCT ecosystem were able to immediately use this tool to quickly debug their designs. Similarly, in ongoing work [4], we extended Calyx to support reasoning about *latency-sensitive* interfaces, which are less compositional but more efficient than Calyx's existing *latency-insensitive* interfaces. By extending Calyx with these new abstractions, we transparently improved the performance of existing frontends and enabled new optimizations that were impossible before.

Tools like Calyx enable automatic generation of accelerators but do not address the programmability challenge. With collaborators at MIT, I have been working on the Exo infrastructure [16] which uses *user-level scheduling* [17] to map high-level programs to accelerators. We have designed new mechanisms [9]

to abstract and bundle schedules into reusable application- and accelerator-specific libraries which make programming new accelerators easier.

Enriching Low-level Abstractions

Hardware description languages (HDLs) offer low-level control: designers have precise control over how hardware is instantiated and its timing. However, this efficiency comes at a cost; structural and temporal constraints are not well encapsulated. For example, a multiplier might have a latency of four cycles but be able to process new inputs every cycle due to internal pipelining. The signature of this multiplier, however, will look identical to a multiplier that takes 32 cycles to process inputs and isn't pipelined. Users must instead read out-of-source documentation to understand the *latency-sensitive* behavior of such modules or use *latency-insensitive* interfaces [18] which use extra signals to abstract away timing behaviors. If such modules are composed incorrectly, or their timing behavior is slightly altered, the design will silently break with hard-to-debug data corruption issues.

I designed Filament [6], a new HDL, to enable efficient and correct composition of *latency-sensitive* designs. Filament *enriches* HDL signatures with timing information that tracks when signals are available and how modules are internally pipelined. Next, Filament's type system, inspired by *separation logic*, checks the design at compile-time and **guarantees that there are no pipelining bugs**. If there are potential problems, instead of hours of painful debugging, the users are provided with actionable error messages. Filament has seen rapid impact. The **Jane Street hardware acceleration team** is exploring the feasibility of integrating Filament by porting existing designs written in the HardCaml [19] to Filament. We are also working with the **Google XLS** [20] team to build a new integrated system that combines the power of Filament's type system with high-level hardware generation. Both teams have stated the concision and generality of Filament's type system as a primary motivation for this integration.

When building complex hardware designs, users parametrize their implementation: the bitwidth of the computation, the number of circuits to generate, the parallelism expose in the interface. Parametrization induces exponentially large design spaces making traditional, test-based methodologies insufficient at establishing correctness. My ongoing work [7] adds *symbolic reasoning* to Filament's type system allowing it to **prove absence of pipelining bugs in entire design spaces**. This extension also allows Filament to provide stable interfaces to designs generated from high-level programming models; by abstracting over their timing behaviors, Filament can ensure that no matter what module is generated, the composition will be correct.

Future Work

The golden age of computer architecture [21] offers exciting opportunities for democratizing the design and use of application-specific hardware. Building upon the principles explored in my PhD, I will design new programming systems across the hardware-software gap to explore these problems.

Rethinking the HDL abstraction. Rust, inspired by work on Cyclone [22], pioneered a new model of systems programming where users can rely on the type system to eliminate a large class of memory-safety and concurrency issues. I will redesign the register transfer level (RTL) abstraction of HDLs to provide strong compile-time guarantees that enable users to build optimized hardware designs, publish and use libraries, and rapidly explore design spaces with hardware generators. Filament demonstrates that this is possible for particular classes of circuits; I plan to extend it to a general-purpose HDL that can reason about arbitrary hardware and use it to build **a new, compositional hardware design infrastructure**. There are several interesting challenges and opportunities: How can Filament reason about latency-insensitive interfaces [23] and provide strong guarantees such as deadlock freedom [24]? How can we **reason about target constraints of physical implementations**—clock domains, multi-cycle paths, and wire loads [25]—which are needed to design real systems-on-chip (SoCs)? Can Filament-style reasoning be instantiated as a gradual type system for traditional HDLs and work in tandem with existing formal verification tools like SystemVerilog Asser-

tions to provide scalable and compositional guarantees [26]? Solving these problems will require cross-stack collaborations with researchers working on formal methods, chip design, and physical implementations.

Integrating hardware design and synthesis. Hardware synthesis tools transform circuit descriptions, written in HDLs, into physical layouts, which can then be fabricated as chips or simulated on field-programmable gate arrays (FPGAs). However, there is an abstraction gap here: a functionally correct design might turn out to be impossible to realize within the required resource and frequency constraints. Like HLS tools, this is because HDLs **do not reason about the target constraints** of synthesis tools. For example, a particularly long wire may limit the frequency, but the user has no way of knowing in advance; the length of the wire is decided during the synthesis process, leading to long iteration cycles. I will design new programming models to integrate hardware design with synthesis. For example, common routing problems, like long wires, can be resolved by adding additional registers, but a synthesis tool cannot automatically make this change because it affects the timing behavior of the circuit. By **enriching the HDL-synthesis interface** to expose design parametrization [7], the tool can *automatically* resolve some of these problems and, by using a Filament-like system, guarantee that the changes are correct. This is a timely problem: renewed interest in open-source flows for chip design [27] offers the opportunity to redesign the interface between HDLs and synthesis tools and dramatically reduce the time and effort needed to go from HDL programs to real chips.

Co-designing languages, compilers, and architectures. Accessible software abstractions are key to the adoption of new accelerators [28]. However, new accelerators offer inflexible programming models while users want high-level abstractions. Bridging this impossible gap is the compiler's job: map computations to the accelerator's limited resources while exploiting spatial and temporal parallelism. I will co-design languages, compilers, and architectures to more appropriately balance this task.

1. **Reason about target constraints.** Programming models for accelerators should expose the constraints of target. For example, using coarse-grained reconfigurable arrays (CGRAs) [29] requires careful, temporal reuse of physical resources. I will design new domain-specific languages (DSLs) that allow users to express temporal reuse explicitly, thereby simplifying the task of the compiler, and providing more user-level control. Such DSLs can then be enriched with Dahlia-like type systems [1] to help guide the user towards better performance trade-offs.
2. **Build reusable infrastructures.** Shared infrastructures can allow us to rapidly design programming models for diverse accelerators. Such an infrastructure much factor out commonalities while being *extensible* enough to support accelerator-specific reasoning. Vectorized reconfigurable dataflow architectures [30] offer a natural, architecture-level split: each vector lane can be reconfigured for a particular computation. I will design a *multi-tier* compiler with a high-level vectorizing compiler to map computations to vector lanes while a lower-level, domain-specific compiler decides how to reconfigure each lane.
3. **Enrich low-level abstractions.** Programming interfaces for accelerators are impoverished and require deep knowledge about the invariants of the architecture. I will design new *interface languages* to specify structural and temporal constraints of the architectures. By connecting such languages to type and logic systems, the interface will provide compositional correctness guarantees and enable design of a diverse array of programming models for each architecture.

Productive design and use of accelerators will usher the next era of computational scaling, achieved by new programming systems that fundamentally rethink existing abstractions. I am excited to apply ideas from programming languages, compilers, and computer architecture to build tools that accelerate the modern computing stack.

References

[1] **Rachit Nigam**, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. Predictable accelerator design with time-sensitive affine types. In *PLDI*, 2020. doi:10.1145/3385412.3385974.

[2] Andrew Butt, Matt Hoffman, **Rachit Nigam**, and Zhiru Zhang. Customizing memory structures for high-level synthesis. *In preparation*, 2023.

[3] **Rachit Nigam**, Samuel Thomas, Zhijing Li, and Adrian Sampson. A compiler infrastructure for accelerator generators. In *ASPLOS*, 2021. doi:10.1145/3445814.3446712.

[4] Caleb Kim, Pai Li, Anshuman Mohan, Andrew Butt, Adrian Sampson, and **Rachit Nigam**. Unifying static and dynamic intermediate languages for accelerator generators. *Under review (APSLOS 2024)*.

[5] Griffin Berlstein, **Rachit Nigam**, Christophe Gyurgyik, and Adrian Sampson. Stepwise debugging for hardware accelerators. In *ASPLOS*, 2023. doi:10.1145/3575693.3575717.

[6] **Rachit Nigam**, Pedro Henrique Azevedo de Amorim, and Adrian Sampson. Modular hardware design with timeline types. In *PLDI*, 2023. doi:10.1145/3591234.

[7] **Rachit Nigam**, Ethan Gabizon, Edmund Lam, and Adrian Sampson. Correct and compositional hardware generators. *Under review (ASPLOS 2024)*.

[8] Alexa VanHattum, **Rachit Nigam**, Vincent T. Lee, James Bornholt, and Adrian Sampson. Vectorization for digital signal processors via equality saturation. In *ASPLOS*, 2021. doi:10.1145/3445814.3446707.

[9] Yuka Ikarashi, Samir Droubi, Kevin Qian, **Rachit Nigam**, Alex Reinking, Gilbert Louis Bernstein, and Jonathan Ragan-Kelley. Productive abstractions for user-scheduable languages. *Under review (PLDI 2024)*.

[10] The CIRCT Authors. CIRCT: Circuit IR compilers and tools, 2023. URL: https://circt.llvm.org.

[11] Sergi Granell Escalfet. Accelerating Halide on an FPGA. Master's thesis, Universitat Politècnica de Catalunya, 2023.

[12] Tianqi Chen et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI*, 2018.

[13] Hsiang-Tsung Kung. Why systolic architectures? *IEEE Computer*, 1982.

[14] The Pollen Authors. Pangenomics graph queries in Calyx, 2023. URL: https://github.com/cucapra/pollen.

[15] Benjamin Carleton and Adrian Sampson. Polynomial approximations in Calyx, 2023. URL: https://github.com/cucapra/calyx-nums.

[16] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. Exocompilation for productive programming of hardware accelerators. In *PLDI*, 2022.

[17] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*, 2013.

[18] Kevin E. Murray and Vaughn Betz. Quantifying the cost and benefit of latency insensitive communication on FPGAs. In *FPGA*, 2014.

[19] Jane Street. HardCaml: Register transfer level hardware design in OCaml, 2022. URL: https://github.com/janestreet/hardcaml.

[20] Google. XLS: Accelerated hardware development, 2023. URL: https://google.github.io/xls/.

[21] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Communications of the ACM*, 2019.

[22] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX ATC*, 2002.

[23] Luca P Carloni, Kenneth L McMillan, and Alberto L Sangiovanni-Vincentelli. Theory of latency-insensitive design. *ICCAD*, 2001.

[24] Muralidaran Vijayaraghavan and Arvind. Bounded dataflow networks and latency-insensitive circuits. IEEE Press, 2009.

[25] Ron Ho, Kenneth W Mai, and Mark A Horowitz. The future of wires. *Proceedings of the IEEE*, 2001.

[26] Peitan Pan, Shunning Jiang, Yanghui Ou, and Christopher Batten. Towards gradually typed hardware description languages. *Languages, Tools, and Techniques for Accelerator Design*, 2023.

[27] The OpenROAD Project. OpenROAD: An open-source RTL-to-GDS flow, 2023. URL: https://github.com/The-OpenROAD-Project/OpenROAD.

[28] Sara Hooker. The hardware lottery. *Communications of the ACM*, November 2021.

[29] Graham Gobieski, Souradip Ghosh, Marijn Heule, Todd Mowry, Tony Nowatzki, Nathan Beckmann, and Brandon Lucia. A programmable, energy-minimal dataflow compiler and architecture. In *MICRO*, 2022.

[30] Alexander Rucker, Matthew Vilim, Tian Zhao, Yaqi Zhang, Raghu Prabhakar, and Kunle Olukotun. Capstan: A vector RDA for sparsity. In *MICRO*, 2021.