面向 Julia 语言的高层次综合工具链

Benedict Short 伦敦帝国理工学院 英国伦敦

Ian McInerney i. mcinerney17@imperial.ac.uk 伦敦帝国理工学院 英国伦敦

John Wickerson j.wickerson@imperial.ac.uk 伦敦帝国理工学院 英国伦敦

arXiv:2512.15679v1 [cs.SE] 17 Dec 2025

摘要

随着百亿亿次级计算和数据驱动方法的发展，问题规模大幅增长，底层算法的计算需求也随之增加。因此，人们开始将计算任务卸载到 GPU 和 TPU 等通用硬件加速器上，并重新关注使用 FPGA 设计面向特定问题的加速器。然而，这些面向特定问题的加速器目前仍面临“双语言问题”：算法使用一种（通常是更高层次的）语言开发，而内核则使用另一种处于完全不同抽象层次、且需要截然不同专业知识的语言实现。为解决这一问题，我们提出了一种基于 MLIR 的新型编译器工具链，通过将使用 Julia 编程语言编写的内核自动编译为 SystemVerilog，在无需任何额外指令或语言定制的情况下统一开发流程。我们的工具链同时支持动态功耗调度和静态调度，可直接集成 AXI4-Stream 协议，以连接片上和片外存储器等子系统，并生成与供应商无关的寄存器传输级代码。该原型工具链能够综合一组信号处理/数学基准程序，使其在实际 FPGA 器件上以 100MHz 运行，并达到由仅能从 C 或 C++ 等低级语言进行编译的最先进工具链生成的设计吞吐量的 59.71% 至 82.6%。总体而言，该工具链使领域专家能够像平常一样使用 Julia 编写计算内核，然后无需额外编译指示或修改即可将其重新定向到 FPGA。

ACM 计算分类体系

• **硬件 → 高层次综合与寄存器传输级综合**；**硬件描述语言与编译**；**硬件加速器**。

关键词

Julia 语言、MLIR、高层次综合、CIRCT

1 引言

摩尔定律的终结意味着，科学算法不能再依靠计算机体系结构和制造工艺的进步来提升 CPU 性能，而必须转向软硬件协同设计方法，既针对当前硬件优化软件，也可能设计针对应用进行优化的硬件 [33]。针对当前硬件优化软件的做法日益普及，美国百亿亿次级计算项目（ECP）等大型项目展现了这种做法产生大规模影响的潜力 [21, 25]。然而，针对应用优化硬件的做法通常仍局限于对延迟和吞吐量有严格要求的领域，例如实时控制系统 [42, 49], 和粒子物理学 [1, 46]。

应用专用硬件设计难以广泛采用，一个关键因素是所谓的“双语言问题”：科学算法通常先用高级语言进行原型设计和开发，但随后必须转换为低级语言（例如用于 FPGA 设计的寄存器传输级语言）[8]。这会增加开发工作量，因为通常需要同时开发和维护两种实现，而且负责这两个层次的可能是不同的团队。

解决双语言问题，意味着将领域科学家/工程师编写的高级实现直接编译为硬件加速器，这可以借助高层次综合（HLS）工具实现。传统上，现有的大多数 HLS 工具都基于 C/C++ “高级”语言，并采用为编译和优化顺序操作流而开发的编译器工具链，例如 LLVM 框架。尽管 HLS 已成功应用于视频处理、图处理和基因组学等领域，但这种传统框架并不适用于现场可编程门阵列设计，可能导致 HLS生成的设计表现欠佳，且源代码缺乏可移植性 [14]。

造成这一问题的一个关键原因是，当使用为顺序 C/C++设计的编译器工具链时，HLS编译器无法获取足够的设计高级信息。这些 HLS 编译器使用的中间表示（IR）处于算子级，包含加法、乘法、位移等基本运算。这种低层次的 IR 会丢失关于更高级并行结构的信息，并可能妨碍编译器识别任务级/模块级的并行性（例如，共享卷积等高级算子）[50]。设计人员可以通过在代码中添加编译指示，或使用能够对算子进行编码的专用库，向编译器提供这些缺失的信息；然而，这些编译指示和库通常是工具链专用的，缺乏可移植性。

为克服信息不足的问题，人们提出了两种方法：从高于 C/C++ 的语言入手，以及提高编译器工具链所用中间表示的抽象层次。将高级语言编译为硬件一直是数十年来活跃的研究领域，先前研究提出了针对 Haskell [3, 20], 、Python [2, 13, 23, 27, 29], 、Rust [5] 和 Ruby [37] 的工具链。这些研究在与各自语言生态系统的集成方式上各不相同，有些研究在基础语言之上实现了定制的领域专用语言，例如用于 Python 的 HeteroCL [29] 和 PyHDL[23] ，或用于 Ruby 的 HDLRuby [37] 。其他工具链则专注于语言或框架的精简子集，例如将 TensorFlow 计算图编译到现场可编程门阵列的 HPIPE [24], 或 MATLAB HDL Coder。

其他 HLS 工具链提出改变用于生成硬件描述语言的中间表示的抽象层级。抽象层级最高的方案直接解析语言，并从内部代码和数据流图（CDFG）结构生成硬件描述语言，例如

[13] 用于 Python， [9] 用于 Julia。近年来，工具链开始采用现有的较高层级中间表示，例如 SPIR-V [27]和多级中间表示（MLIR） [2, 47, 50]。

MLIR 目前已成为构建 HLS 工具的热门选择。通过以下方式，MLIR 能够生成硬件描述语言：使用 CIRCT 框架编写 SystemVerilog [35, 47],为现有高层次综合工具链生成 C/C++ 代码 [4, 50], 或输出 LLVM 中间表示，供 Vitis 高层次综合等供应商专属工具链使用 [34, 44]。先前研究通过多种前端生成 MLIR，包括遍历 Clang 语法树 [39, 51], 使用 flang Fortran 前端 [44], 以及通过 PyTorch-MLIR 从 PyTorch 生成 [47, 51]。MLIR 的设计还允许工具链在标准 MLIR 方言之上实现自己的中间表示“方言”，以表示自定义操作/功能或协助优化。自定义方言的例子包括用于表示模板计算的 Stencil-HMLS [45] ，以及用于添加调度信息的 HIR [38] 。

1.1 我们的贡献

在这项工作中，我们提出了一套采用宽松许可协议的开源工具链。它通过 CIRCT 框架使用 MLIR，将 Julia 语言编译为 SystemVerilog。Julia 语言专为科学与数学而设计，并原生支持常见的数学运算和概念（例如线性代数）。[1] 我们的工具链名为 *JuliaHLS*，结合了上述两种方法——从 Julia 语言直接转换为 MLIR。借助这一方案，我们的工具链能够进行高级分析与优化，例如线性代数运算符融合或分块；若没有脆弱的“提升遍”或程序员标注的编译器指令，这些优化此前都难以实现。我们还利用 Julia 编译器基础设施的独特功能来构建这一工具，包括 AbstractInterpreter 接口和方法表覆盖。AbstractInterpreter 接口使我们能够创建一种新的 MLIRInterpreter 编译流程，在标准 Julia 编译器旁运行，同时复用其中的许多部分（例如 AST 降级、类型分析、优化遍等），而不依赖 Julia 编译器栈的内部实现。

JuliaHLS能够生成高性能设计，在 100MHz 下通过时序分析，并实现最高达 Dynamatic v2.0 吞吐量的 82.6%。此外，在我们的测试中，该工具链能够正确编译 Dynamatic 无法编译的若干程序，并生成能够以更高时钟频率运行的设计。

我们首先在第 2 节介绍 Julia 和相关高层次综合工具链的必要背景信息。随后在第 3 节讨论 JuliaHLS 的设计理念，并在第 4 节介绍其实现。第 5 节展示了若干案例研究，接着在第 6 节和第 7 节讨论结果与未来工作。

2 背景

2.1 Julia 语言

Julia [8] 是一种基于 LLVM 框架构建的开源动态类型语言 [30], 可使用高级、富有表现力且可组合的语言编写代码，然后通过即时编译（JIT）创建高性能软件。其语法与数学符号十分相似，也类似于 MATLAB 语言，因此在科学计算和工程领域广受欢迎。

Biggs 等人最早倡导将 Julia 用作 HLS 工具的源语言。[9], 他们介绍了该语言的若干优势特性，包括：

高级信息 Julia 将线性代数等数学概念作为一等特性集成到语言中。这使编译器能够识别更高级别的操作以及矩阵等高级对象，从而为优化、并行化和资源共享提供更多机会。这样便无需通过提升遍来重建高级信息，例如 Polygeist [26, 39] 等工具所做的那样。

编译器基础设施 Julia 可扩展的编译器基础设施（例如 AbstractInterpreter）允许新的目标后端直接集成到该语言的标准发行版中，从而复用现有基础设施。此外，已有多个成功项目从 Julia 面向加速器平台，包括图形处理器 [6, 7], TPU [19] 和 Graphcore IPU [22]。

生态系统 Julia 包生态系统包含许多“纯 Julia 包”。这些包并非只是在 Fortran 或 C语言库之上提供一个薄封装，而是真正使用 Julia 实现算法。这使得人们能够在现有算法中模拟自定义数值格式，也有望仅通过使用现有包中的算法来创建加速器核心，而无需重新实现这些算法。

多重分派与类型系统 Julia 使用动态类型系统，也允许为变量和方法参数指定类型约束，并通过多重分派来调用函数：编译器会检查所有方法参数的类型，并选择最适合这些方法参数的方法。这种机制鼓励使用抽象类型和泛型运算符编写泛型代码，随后通过类型推断过程在编译时对代码进行特化，生成适用于特定数据类型的实现。抽象类型的使用使方法能够处理自定义数据类型，例如任意浮点或定点数格式（分别通过 ArbFloat 和 FixedPointNumbers 软件包实现），从而可以使用 CPU 代码而非模拟器/仿真器，在现场可编程门阵列上对算法进行功能仿真。

2.2 高层次综合工具链

目前已开发出许多商用和研究用的高层次综合工具链，表 1 对相关工具链进行了高级别比较，下面还将简要介绍这些工具链。

AMDVitis 是一款商业 HLS 工具，专门为 AMD/Xilinx FPGA 生成经过优化的硬件。它的设计入口语言是 C/C++, 不过近期版本也支持直接传入 LLVM 7 中间表示。许多研究工具链会通过生成

[1]为进行双盲评审，已移除指向代码仓库的链接

HLS 工具	Open 源代码	供应商无关	调度 动态	调度 静态	可用设计	可复用的	语言 源代码	语言 IR	语言 目标
本工作	✓	✓	✓	✓	✓	✓	Julia	MLIR	SystemVerilog
Biggs 等人 [9]	✓	✓	✓	✗	✓/✗[3]	✗	Julia	Julia/点图	VHDL
Lounes 等人 [36]	✗	✗	✗	✓	✓	✓	Julia	MLIR	HLS C语言++
MATLAB 硬件描述语言编码器	✗	✓	✗	✓	✗	✗	MATLAB	未知	Verilog/VHDL/SystemVerilog
AMD Vitis HLS	✗	✗	✗	✓	✓	✓/✗[2]	C语言/C++	LLVM	比特流
Dynamatic [26]	✓	✓	✓	✗	✓[4]	✓/✗[2]	C语言/C++	v1：LLVM v2：MLIR[5]	VHDL
ScaleHLS [50]	✓	✗	✗	✓	✓	✓	C语言/C++[5] PyTorch	MLIR	HLS C++
HIDA [51]	✓	✗	✓	✗	✓	✓/✗[2]	C语言/C++[5] PyTorch	MLIR	HLS C语言++
PandA Bambu 高层次综合 [17]	✓	✓	✗	✓	✓	✗	C语言/C语言++	“Bambu 中间表示”	Verilog/VHDL
LegUp 高层次综合 [10]	✓/✗[1]	✓/✗	✗	✓	✓	✓	C语言/C++	LLVM	Verilog

[1] 最初为开源工具，但现在已闭源，[2] 部分可复用，[3]仅支持非常有限的语言子集，[4] 工具链会作出额外假设，[5]C语言/C++ 使用 Clang 解析为 LLVM 中间表示，然后通过 Polygeist 提升为 MLIR [39]**表 1：高层次综合工具链及其特性对比**

C/C++ （例如 HIDA [51] 和 ScaleHLS [50]），或直接传入 LLVM 中间表示（例如 Fortran [44]），来使用 Vitis 作为后端。

MATLAB HDL Coder 是 MathWorks 推出的一款商业工具，可将 MATLAB 代码和 Simulink模型编译为 IP 核，并使用 Verilog、SystemVerilog 或 VHDL。它支持多种现场可编程门阵列技术，包括 AMD、Altera 和 Microchip；然而，先前的研究表明，它生成的设计性能特征较差 [15]。

PandA Bambu HLS 可将 C/C++ 编译为静态调度、与厂商无关的硬件设计 [17]。欧洲空间局等机构[12, 18, 32] 已广泛使用它来实现定制加速器。

ScaleHLS 和 HIDA 基于 MLIR，可接受 C/C++ （使用 Clang 解析为 LLVM 中间表示，再通过 Polygeist [39]提升为 MLIR），也可接受 PyTorch-MLIR 设计。HIDA [51] 可创建动态调度的已优化设计，并输出 C++ 代码，以便传递给厂商工具链（例如 Vitis）。ScaleHLS [50] 可创建静态调度的设计，并输出 C++ 或 LLVM 中间表示，以便传递给后续工具链。

Dynamatic 可根据 C/C++ 输入生成动态调度的 VHDL 设计。最初的 Dynamatic 工具 [26] 使用 LLVM 中间表示执行内部优化遍；不过，Dynamatic v2.0 已重写为使用 MLIR 执行内部优化遍（通过 Polygeist [39]解析 C/C++ ）。

Julia 高层次综合工具链- Biggs 等人的初期工作 [9]修改了 Julia 编译器以提取控制数据流图，随后将其转换为由弹性组件构成的点图，并传递给 Dynamatic 进行寄存器传输级生成。它仅支持非常有限的一小部分 Julia 语言（且不支持矩阵/线性代数），并且由于与 Julia 编译器内部实现紧密耦合，难以维护。

Lounes 等人 [36]最近提出了一种基于 MLIR 的工具链，可将 Julia 编译为适用于 Vitis 高层次综合的 C语言++，并将 Julia 类型化中间表示直接转换为 MLIR 结构化控制流（*scf*）方言，以便由 ScaleHLS 处理。该工作流会将矩阵直接转换为低层级的内存缓冲区（memref）语句，从而丢失有关矩阵类型及后续操作的高层级信息。

3 工具链设计

选择 Julia 语言作为该 HLS 工具链的源语言，是因为它专注于科学与工程计算，并且符合第 2.1 节概述的四点要求。JuliaHLS 的一个关键愿景是：只要算法采用抽象类型以泛型形式实现，编译函数的现场可编程门阵列实现就不应要求对算法实现进行任何更改。为此，我们不在 Julia 语言中实现用于描述加速器的领域专用语言，而是直接将纯 Julia 代码编译为硬件加速器的寄存器传输级代码。

3.1 中间表示

Biggs 等人[9]尽可能长时间地将设计保留在 Julia 类型化 SSA 中间表示中，只在生成硬件描述语言时，才将设计作为点图交给 Dynamatic v1。不过，这导致了性能问题，因为该工具链无法利用任何现有的高层次综合优化遍；评估结果显示，与使用 Dynamatic v1 相比，使用 Julia 工具链时资源使用量大幅增加。为

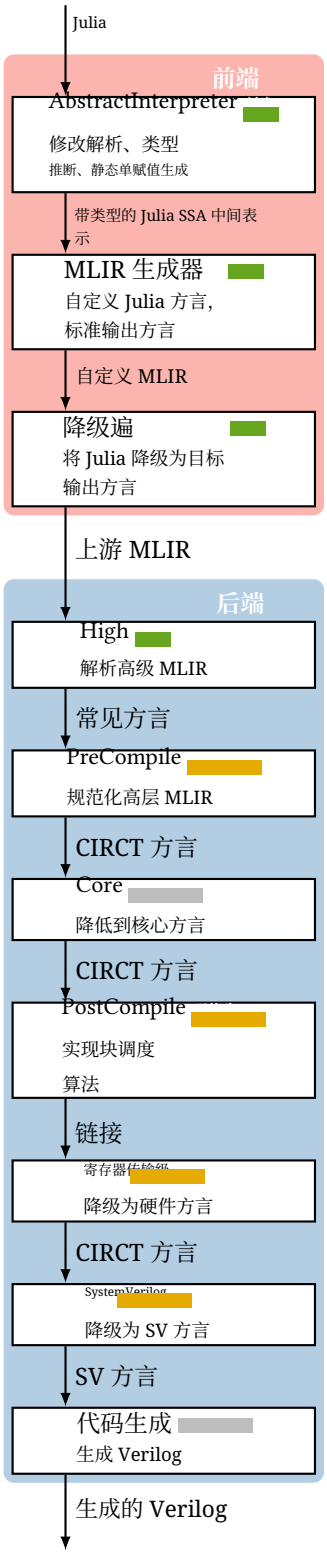

图 1：高层系统架构，其中每个遍都标记为　　、　　，或　　。

解决这一问题，作者建议将基于 LLVM 的工具链中的优化遍移植到 Julia 中，使其在生成硬件描述语言之前作用于 Julia 类型化 SSA 中间表示；此外，还建议直接通过 Julia 生成硬件描述语言，从而打造一套完全基于 Julia 的工具链。然而，构建一套所有组件和优化遍都以 Julia 编写的完全基于 Julia 的工具链难以维护，而且会使 JuliaHLS 工具链的开发与更广泛的高层次综合社区隔离开来。

其他用于将任务卸载到 GPU[6, 7] 和 Graphcore IPU [22] 等加速器的 Julia 工具链，会利用 Julia 基于 LLVM 这一特点，拦截 LLVM 中间表示，并将其重定向到所需加速器的 LLVM 目标后端。类似的流程也可用于生成寄存器传输级代码：将 Julia 生成的 LLVM 中间表示传入 AMD Vitis 高层次综合（通过版本降级转换器，类似于 Fortran 高层次综合中采用的方法 [44]）。不过，这种流程会非常依赖特定厂商，而且随着时间推移，将 LLVM 降级转换为 Vitis 可用的版本可能会变得愈发困难。

Julia 类型化 SSA 中间表示中包含大量关于高层构造的信息，例如矩阵及其操作；Julia 到 LLVM 中间表示的降级遍会移除这些信息，导致高层次综合工具链无法获取它们。此外，LLVM 中间表示以命令式和过程式方式对程序建模，无法充分表达硬件固有的并行特性，因而削弱了编译器生成高效硬件设计的能力。这被称为“过早降级”，Lattner 等人甚至指出：“过早降级是一切罪恶的根源”[31]，，以此强调 LLVM 不足以胜任高层次综合。

相反，JuliaHLS 将 MLIR 生态系统用作工具链的核心，并将 MLIR 方言用作其中间表示。MLIR 是一个模块化框架，由一组核心“方言”构成，这些方言在单一命名空间中定义了一系列操作、属性和类型。与 LLVM 不同，LLVM 只允许使用单一方言，且无法原生表示嵌套代码或通用循环；MLIR 则允许在同一生成的中间表示中叠加不同方言，从而能够混合高层操作（例如矩阵加法）和低层操作（例如标量加法），同时保留程序结构和控制流。MLIR 包含许多不同的方言，包括用于表示 for 和 if 等结构化控制流结构的 *scf*、用于表示矩阵/向量操作的 *linalg*，以及用于表示通用张量的 *tosa*。

3.2 硬件描述语言生成

基于 MLIR 的高层次综合工具链可以通过两种主要途径将程序降级为硬件描述语言：将 MLIR 降级为 LLVM 方言，或通过 emitc 方言转译为 C语言，再使用 AMD Vitis HLS 等工具生成寄存器传输级；或者使用支持输出硬件描述语言的基于 MLIR 的 HLS 工具，例如 CIRCT [35] 或 Dynamatic v2.0 [26]。第一种方案的主要问题在于，它依赖通过不同层级的表示形式（LLVM 中间表示或 C语言/C++代码）将数据传递给另一个工具，例如 Vitis HLS。这种流程还可能导致供应商锁定，因为处理 LLVM/C/C++代码的外部工具通常要求特定的代码结构或库，才能识别构造或接口；而且许多此类工具严重依赖 pragma 和其他指令来控制设计流程、生成高效的设计和接口，这些要求可能因工具而异。

JuliaHLS 使用 CIRCT [35] 基础设施，在整个处理过程中始终以 MLIR 形式保留设计，然后将其降级为 SystemVerilog 方言，再写出设计的 SystemVerilog 描述。CIRCT 是一套面向硬件设计的 MLIR 方言、转换和优化遍集合。它最初是作为数据流加速器设计工具起步的，近期还通过 Calyx 工具链 [40] 集成了对静态调度设计的初步支持。

4 编译器栈

JuliaHLS 实现了一种如图 1 所示的三阶段模块化编译器架构，其中前端提取程序，后端的前半部分对程序进行优化和转换，后半部分生成 SystemVerilog。

4.1 前端

JuliaHLS 前端的主要作用是从符合 Julia 语言习惯的源代码中提取类型稳定的控制数据流图，并将其转换为 MLIR。该转换分为三个阶段：解析类型稳定性、解决中间表示之间的不匹配问题，以及生成由标准方言组成的优化后 MLIR。图 1 的上半部分展示了该组件架构的高级概览。

4.1.1 类型稳定性。由于 MLIR 采用静态且强类型，而符合 Julia 语言习惯的代码不必如此，因此第一个编译阶段是提取 Julia 代码的静态强类型版本，即所谓的“类型稳定”代码。要为 JuliaHLS 工具链生成可用的类型稳定代码，需要完成两个关键步骤：执行分析，为运算符和变量标注单一具体类型；消除代码中的运行时多态（即动态分派）。消除运行时多态、仅保留编译时多态（即静态分派），对于生成高效硬件至关重要，因为动态分派可能由于增加额外的硬件和控制流而导致设计规模呈指数增长，进而产生不可预测的特性或造成资源浪费。然而，这也带来了固有的挑战，因为标准 Julia 编译器的类型推断阶段只会尽力实现类型稳定性，并且当其判断在取指执行处理器上静态获知类型所带来的性能提升微不足道时，便会提前退出。其他 Julia 工具链（如 GPU 编译器 [7],）并不严格强制要求强类型稳定性或静态分派，而是利用 CPU 与 GPU 之间的紧密耦合，将代码中类型不稳定的部分委托给 CPU，将类型稳定的部分交给 GPU。

在 JuliaHLS 中，我们要求只能使用静态分派和类型稳定代码，这与近期的 Julia 二进制编译工作流 [11]的要求类似。为此，我们创建了一条新的 MLIRInterpreter 编译流程，它是 Julia 的“AbstractInterpreter”的具体实现，使 JuliaHLS 能够修改 Julia 编译器内部流程的行为，而无需修补内部代码。为强制使用静态分派，我们利用 Julia 的“方法表覆盖”功能，注入有问题方法的特殊版本，以便仅在 MLIRInterpreter 编译器中提供具体实现，同时不影响标准 CPU 或 GPU 后端使用的方法。此外，MLIRInterpreter 引入了自定义内联策略，使我们能够在适当情况下提取高级或通用程序内在操作。例如，Julia 默认的 CPU 优化内联策略会假定线性代数运算按顺序执行，并可能内联向量/矩阵上的某些计算。然而，为了在 HLS 工具链中进行更多优化，应将矩阵运算直接提取为单个操作，然后传递给后端。

最终结果是原始程序在 Julia 带类型的静态单赋值中间表示中的类型稳定高级表示，其中使用基本块对控制依赖进行建模，并使用静态单赋值表示数据依赖。

4.1.2 生成 *MLIR*。下一步是获取上一阶段生成的 Julia 中间表示，并将其转换为非常高级的 MLIR。这一过程分为两个步骤：首先将 Julia 中间表示直接转换为为 JuliaHLS 构建的自定义 MLIR 方言，即 Julia 方言；然后应用转换遍，以解决中间表示层面的不匹配问题并生成惯用的 MLIR。将转换分为两个阶段，可以优先构建简单且正确的代码生成基础设施，因为这会降低整体转换例程的复杂性，并有助于更好地隔离所需的清理工作，因为这些工作可以作为独立的 MLIR 转换遍来实现。

实现这些程序变换的 MLIR 转换遍使用 Julia 编写，然后在 MLIR 引擎中注册为外部遍。主要应用的变换包括：

- 将 Julia 的按列数组转换为按行张量对象，
- 将 Julia 的 SSA 违规映射为符合 SSA 规则的等价形式（例如，将 Julia 的内存缓冲区模型转换为 MLIR 的高级张量对象），
- 将 Julia 从 1 开始的数组索引转换为从 0 开始，
- 额外的类型转换，以及
- 将 Julia 标准库中的数学运算映射为符合 FPGA 规范的替代方案。

下一步是将 Julia 方言降低为下游后端（如基于 LLVM 的即时编译器或 CIRCT 框架）通常支持的标准 MLIR 方言。为此，我们在 Julia 中编写自定义遍，将 Julia 方言高效地降低为 TOSA、张量、算术和控制流 MLIR 方言。使用 TOSA 方言和张量方言，JuliaHLS 能够在生成的中间表示中直接对线性代数对象及其操作进行建模；这与先前研究形成对比，例如 [36],将矩阵/数组建模为使用内存引用方言的共享缓冲区，并通过该方言分别控制元素的访问，这类似于软件和硬件中的缓冲器。借助 MLIRInterpreter 的自定义内联逻辑，这些矩阵运算得以保留在 Julia 类型化 SSA 中间表示中，使我们能够直接从程序中提取表达式和线性代数，从而利用成熟且维护良好的 MLIR 方言和遍，在将程序降低为缓冲区化表示之前对其进行优化。因此，我们的工具链使硬件后端能够应用领域专用启发式方法，实现适合硬件的方案。

4.1.3 优化。标准 Julia 编译器会生成非常冗长的类型化静态单赋值中间表示，因为它将大部分优化工作交给了 LLVM 后端。这意味着由

MLIRInterpreter 生成的中间表示可能冗长且未优化，因此 JuliaHLS 的 MLIR 后端也必须应用相同的优化遍 —— 这些优化遍已在标准 MLIR 库中实现。

Julia 的 LLVM 优化通道与 JuliaHLS 使用的 MLIR 优化通道之间的主要区别在于，JuliaHLS 必须将 ‘canonicalize’ 通道默认执行的控制流简化级别从 ‘aggressive’ 降至 ‘normal’，因为过于激进的简化可能导致控制流被折叠得过度，以至于 CIRCT 在生成数据流电路时无法判断控制数据流图（CDFG）是否会导致死锁。例如，如果执行流水线分裂成许多不同的并发流，并且这些流共享不同的硬件单元，那么要么需要极其复杂的控制流网络来实现足够的反压（开销极高且容易出错），要么工具必须保证这些资源可以安全共享。因此，该框架一律禁止所有「不规则」控制流，以缓解这一问题。此外，框架还编写并应用了额外的通道，将 MLIR「重新格式化」为 CIRCT 支持的子集，也就是不包含静态变量。实践中，这一组件对于综合大型程序至关重要，可确保程序既能装入现场可编程门阵列，又能达到可与最先进的工具链相媲美的性能特征，正如我们将在第5节中展示的那样。

4.2 后端

JuliaHLS 后端组件接收前端生成的 MLIR，并使用 CIRCT 库生成以 Verilog 或 SystemVerilog 描述的 RTL 设计。我们将 CIRCT 中选取的现有（未经修改）方言和变换与若干自定义 pass，以及对 CIRCT 库的直接修改相结合，从而生成能够满足 100MHz 时序要求的 RTL 设计，并兼容多种厂商综合工具，包括 Altera 和 AMD 的工具。

该后端包含六个抽象层：
(1) *High*：高级 MLIR 方言和操作；(2) *PreCompile*：降低为 CIRCT 支持的 MLIR 方言；(3) *Core*：变换程序，使其仅使用 CIRCT 中合法的操作；(4) *PostCompile*：分别使用 calyx [40] 和 handshake [41] 进行静态或动态调度；(5) *RTL*：降低为 RTL 电路；(6) *sv*：生成 MLIR 格式的 SystemVerilog

每个抽象层都通过一条遍的流水线连接起来，该流水线逐步将设计降级为寄存器传输级，最终转换为硬件描述语言。

High 到 *PreCompile* 流水线是 JuliaHLS 独有的，专门用于处理线性代数运算的降级；而*PreCompile* 到 *Core*、*Core* 到 *PostCompile* 和 *PostCompile* 到寄存器传输级流水线则基于经过修改的上游 CIRCT 遍，以确保生成的寄存器传输级代码正确且可综合（例如，能够在小型 FPGA 上实现并以 100MHz 运行）。

4.2.1 High-to-PreCompile 流水线。此流水线是 Juli-aHLS 独有的，会将高级 MLIR 方言降级为 CIRCT 所接受的方言，具体包括算术方言、memref、函数方言、仿射方言，以及用于数据流生成的控制流方言，或使用 Calyx 进行静态调度的 scf。此流水线中最重要的组件是*TosaToLinalg* 转换，它会先将 TOSA 方言降级为 linalg，然后对程序进行缓冲器化（即，将符合静态单赋值的张量对象转换为非静态单赋值的存储器缓冲器）。值得注意的是，沿函数边界对程序进行全局缓冲器化的一项限制是：使用递归时，这种做法将不再有效。不过，Juli-aHLS 不支持递归，因为递归无法映射为高效且可靠的硬件。

在*TosaToLinalg* 中的下一步是应用一个名为 OutputMemrefPassByRef 的新遍，将返回 n 维数组的函数转换为通过参数传入输出数组引用的形式。此项修改是必要的，以确保 CIRCT 能够生成加载存储队列（LSQ），并允许在设计之外（即片外 DDR3）与存储器进行数据输入和输出。

最后一步是应用修改后的 LowerLinalgToAffine 遍，然后应用 LowerAffinePass，以创建矩阵运算的控制流表示。JuliaHLS 中需要修改该遍，因为原始遍会贪婪地将优化模式应用于所有已注册的方言，从而过度折叠控制流。然而，CIRCT 的数据流生成不支持过度折叠的控制流，因为这意味着不同分支可以共享资源，而“Handshake”方言目前尚不能正式保证这不会导致死锁。

4.2.2 PreCompile-to-Core 流水线。 该降级流水线会将 MLIR 从 *PreCompile* 阶段转换为可被 CIRCT 中的数据流生成器或 Calyx 中的静态调度器接受的 MLIR。对于静态调度，该流水线是 CIRCT 中的单个通道 SCFToCalyxPass，它会将结构化控制流转换为用于静态调度的 calyx 方言。对于动态调度，该流水线包含七个通道，其中最后五个通道来自标准的 CIRCT 握手方言降级流水线，但在流水线开头加入了一个名为 DeleteUnusedMemory 的新通道，并修改了 InsertMergeBlocksPass。

DeleteUnusedMemory 通道通过检查内存缓冲区的读写影响来分析程序，以发现设计中不必要的数据移动，然后删除不必要的复制操作、缓冲器和存储器。该通道对于确保内存密集型和数学计算密集型程序既具备有竞争力的性能特征，又能适配预算有限的 FPGA（例如 Pynq Z1）至关重要；使用该通道时，我们观察到平均性能提升了 25%–40%。

JuliaHLS 还改进了标准 CIRCT 库中的 InsertMergeBlocks 通道。该通道用于在控制数据流图中插入额外的“合并”块，以确保单个块最多只有两条回边。具体来说，我们修改了该通道，使其能够处理块只有一条回边的情况，并且仅在确有需要时才插入额外的合并块，而不是到处插入。这些改动降低了控制数据网络的复杂度，并使综合后的寄存器传输级设计的资源使用量平均最多降低了 20%。

4.2.3 Core 到 *PostCompile* 流水线。使用动态功耗调度时，此流水线会具体化连接控制数据流图各节点的分叉和汇聚操作，然后插入设计所需的各种缓冲器。使用静态调度时，此流水线首先会将 Calyx 计算出的控制调度降级为有限状态机（FSM），然后将各组件的控制端口（即，时钟信号、启动标志和复位线）连接到顶层控制端口。

4.2.4 RTL 到 *SV* 流水线。此流水线会将经过调度的寄存器传输级设计转换为 SV 方言，后者用于实际生成该设计的 SystemVerilog。

对于动态调度设计，JuliaHLS 包含一个经过修改的 HWLoweringPipeline pass，可将操作映射到动态生成的 IP 核。默认情况下，该 pass 会实例化一个异步输出的基于 LUT 的 RAM；然而，只有当存储器具有同步输出（即输出经过寄存器注册）时，现场可编程门阵列综合工具才会将其映射到现场可编程门阵列原语（例如块 RAM）。基于 LUT 的存储器通常速度太慢，无法满足快速时序要求（例如 100 兆赫），因此我们修改了该 pass，在输出路径上实例化一个寄存器。这样做以增加存储器读取的一个访问周期为代价，提高了生成硬件的可综合性。这也降低了高频使用的查找表的扇出，从而减轻了现场可编程门阵列互连结构的整体负载。

对于静态调度设计，此流水线会将有限状态机转换为 SystemVerilog，然后将其与相关计算 IP 核链接起来。JuliaHLS 对原始的 CIRCT/Calyx pass 做了少量修改，以确保生成可综合的 SystemVerilog 代码，并遵循 CIRCT Verilog 生成基础设施中的配置。

4.3 存储器

JuliaHLS 支持动态调度设计中的内存操作。底层握手方言采用共享内存模型，可生成片上 RAM 实例，也可通过 LSQ 与外部存储器通信。外部存储器支持通过可参数化的 LSQ 到 AXI4-Stream 适配器实现，该适配器使用轮询仲裁器管理每个读写通道，并通过向数据流电路施加反压来保留数据依赖，暂停执行，直至请求成功完成。该 AXI 核可与模拟器以及真实硬件无缝集成，例如使用 AMD DataMover IP 核读写外部 DDR3 存储器的 Pynq Z1 FPGA。

4.4 Julia 集成

JuliaHLS 后端采用 C语言++ 编写，包含两种主要设计流程：用于动态调度和静态调度硬件描述语言的 HLSToolDynamic 和 HLSToolStatic。由于 Julia 与 C语言++的接口比与 C++ 的接口更好，因此我们为这两种流程创建了一个精简的 C API 封装器，并将其编译为共享库。Julia 通过 Clang.jl[2] 自动生成的绑定访问后端工具流程，这些绑定公开了各种选项以及用于创建和运行工具流程的一些基本函数。

完整的编译流程通过一个 Julia 包进行抽象，可直接从 Julia REPL 安装。[3] 该包将基于 Julia 的前端与后端封装在一起，提供黑盒式编译流程，可直接从交互式解释器调用和自定义，也可在 Julia 脚本中使用。RTL 生成流程也可通过 CIRCT 的导出选项进行自定义，以便使 RTL

[2]https://github.com/JuliaInterop/Clang.jl [3]该软件包已为双盲评审隐去，最终出版版本中将予以收录。

```
函数 坐标旋转数字计算机(theta::Fixed{T,N}, K::Fixed{T,N})其中 {T,N}
    # 反正切查找表
    角度 = @MMatrix [
       Fixed{T,N}(atan(1));
       固定{T,N}(atan(0.5));
    ...
    ]

    x = K
    y =固定{T,N}(0)
    z = theta

    对于i 在 one(T): T(length(angles))
        di::固定{T,N} = z ≥ 固定{T,N}(0) ?
            固定{T, N}(1) : 固定{T, N}(-1)
        移位::T = i - one(T)

        # 坐标旋转数字计算机旋转更新
        x_新 = x - (di * (y >> 移位))
        y_新 = y + (di * (x >> 移位))
        z_新 = z - di * 角度[i]

        x、y、z = x_新，y_新，z_新
    end

    返回x
end
```

代码清单 1：CORDIC 案例研究片段。

面向多种综合工具，包括 Altera Quartus、AMD Vivado 和 Verilator。

5 案例研究

为展示 JuliaHLS 工具流程，我们介绍两个案例研究，以突出工具链的各个方面：CORDIC 算法的实现和二维卷积内核。除了这些示例外，我们还在其他项目上测试了该工具链，涵盖从简单操作和控制流到牛顿-拉夫森法的实现，以及用于边值问题的复杂常微分方程求解器。

5.1 案例研究描述

5.1.1 坐标旋转数字计算机（*Cordic*）。本案例研究将 CORDIC 算法分为十个阶段 [48]，以迭代方式计算余弦函数。该示例使用定点数实现，这些定点数由名为 FixedPointNumbers 的 Julia 包提供，并展示了如何通过方法表覆盖为 JuliaHLS 添加对新类型的支持。该程序还明确展示了对广泛语言特性的支持，包括控制流、通过数组索引实现的“查找表”、位运算、整数运算和泛型方法。该程序的片段见代码清单 1。

5.1.2 二维卷积_*im2col*。本案例研究使用“im2col”算法执行二维卷积，程序如清单 2 所示。我们使用一个 3x3 输入矩阵和一个 2x2 内核来实例化该算法，但它可以扩展到任意大的向量大小。此示例展示的主要特性包括全面

```
函数 conv2d_im2col(A::MMatrix{MA,NA,T},
K::MMatrix{KH,KW,T}) where {MA,NA,KH,KW,T}
OM = MA - KH + 1ON = NA - KW + 1# im2col 维度
R = KH * KW C = OM * ON # 构建补丁矩阵 S =
MMatrix{R, C, T}(undef) for j in 1:ON, i in 1:OM, vin
1:KW, u in 1:KH row = (v-1)*KH +u col = (j-
1)*OM +i @inbounds S[row, col] = A[i+u-1, j+v-1]
end # 构建扁平化表示 k =MMatrix{R, 1, T}(undef) f
or v in 1:KW, u in 1:KH idx = (v-1)*KH +u
@inbounds k[idx, 1] = K[u, v]end ycol = S' * k # 通
用矩阵乘法 # 将结果散射回 Y Y =MMatrix{OM, ON,
T}(undef) for j in 1:ON, i in 1:OM idx = (j-1)*OM +i
@inbounds Y[i, j] = ycol[idx, 1]end return Y end
```

清单 2：二维卷积核。

```
函数隐式_否则(A, B) 结果 = 0
如果 A <B 结果 = A + B否则
如果 A >B 结果 = A - B结束
返回 结果结束
```

清单 3：隐式_否则测试用例。

可参数化加速器开发、矩阵运算与操作、保留写后读（WAR）和读后写（RAW）依赖关系、返回矩阵、实例化大型片上存储器、与外部存储器交互，以及处理高度复杂的控制流。

	延迟（周期）	查找表/自适应逻辑模块	FF	DSP
动态功耗	13	2.29%	2.52%	0.0%
静态	14	0.27%	0.14%	0.0%

表 2：调度比较的综合结果 。

5.2 动态功耗与静态调度

遗憾的是，由于 Calyx 工具及其方言最近才集成到 CIRCT 中，因此尚无法支持存储器或综合循环。为展示 JuliaHLS 创建动态调度和静态调度代码的能力，我们改用一个更简单的示例：包含隐式 else 子句的 if 语句，如清单 3 所示。

使用两种调度策略编译该程序，以考察性能和资源使用量，结果如表 2 所示。随后，使用 AMD Vivado 针对 Pynq Z1 FPGA 对两种设计的寄存器传输级代码进行综合，以开展时序分析；两者均成功通过了 100 兆赫时钟频率下的测试。请注意，这两种设计的延迟非常相近，但静态调度的设计实现了近乎 8× 的资源利用率降低。

5.3 工具链性能

在本节中，我们仅使用动态调度工具流程，分析 JuliaHLS 如何编译第 5.1 节介绍的两个科学案例研究，因为采用 Calyx 的静态调度工具流程不支持这些示例使用的许多功能。

5.3.1 程序复杂度。为了评估 JuliaHLS 工具链的性能，我们检查了每个案例研究在工具链前端和后端*High-to-PreCompile* 流程中生成的 MLIR 所包含的操作数，并比较了启用和未启用 MLIR 优化通道时的情况。具体而言，我们记录了以下信息：

- 基本块：控制流复杂度的衡量指标
- 总操作数：所需硬件单元最大数量的衡量指标（不进行资源共享时）
- 算术操作：算术操作的数量
- 内存拷贝：数据移动量

结果如表 3 所示，减少幅度越大，意味着 JuliaHLS 工具链发现了更多优化机会。总体而言，得益于内置的 MLIR 优化，工具链的性能提升显著：在坐标旋转数字计算机示例中，总操作数减少了近一半；在二维卷积示例中，内存拷贝次数从 87 次降至 6 次。这些结果还表明，前端优化似乎更侧重于控制流和整数组件，而后端优化对内存操作的影响最大。

5.3.2 性能分析。接下来，我们使用 Verilator 5.030 构建测试平台，考察了两个案例研究的预期性能。我们将设计的延迟测量为从 token Handshake/启动信号的上升沿到结果_有效信号/完成信号的上升沿之间的时钟周期数，并假设频率为 100 兆赫，根据延迟计算吞吐量。延迟和吞吐量的结果分别列于表 4a 和表 4b。在这次测试中，Verilator 无法模拟为二维卷积案例研究生成的未优化 Verilog

设计组件		基本块			总操作数			算术操作			内存拷贝		
		未优化	Opt		未优化	Opt		未优化	Opt		未优化	Opt	
坐标旋转数字计算机	前端	14	7	**2**×	129	58	**2.22**×	44	17	**2.59**×	3	2	**1.5**×
	后端	14	5	**2.8**×	88	76	**1.16**×	18	18	-	4	1	**4**×
二维卷积	前端	49	49	-	486	379	**1.28**×	70	56	**1.23**×	84	84	-
	后端	79	63	**1.25**×	306	203	**1.51**×	82	78	**1.05**×	87	6	**14.5**×

表 3：JuliaHL 组件中优化与未优化时的程序复杂度 **S.**

	未优化	已优化	加速比
坐标旋转数字计算机	5765	241	**23.92**×
二维卷积	>1,000,000,000	1946	∞

（a）周期延迟（越低越好）

	未优化	已优化	加速比
坐标旋转数字计算机	17346.05	414937.76	**23.92**×
二维卷积	≈ 0	17346.05	∞

（b）每秒运算吞吐量（越高越好）

表4：性能特征

设计工具链		查找表/自适应逻辑模块			触发器／寄存器			DSPs		
		未优化	Opt		未优化	Opt		未优化	Opt	
坐标旋转数字计算机	Vivado	12629 (23.73%)	4791 (9.01%)	2.64×	20813 (19.56%)	5155 (4.84%)	4.04×	9（4.09%）	9（4.09%）	-
	Quartus	7936 (9.88%)	2903 (3.61%)	2.73×	14249 (6.48%)	4618 (2.10%)	3.09×	5 (3.21%)	5 (3.21%)	-
二维卷积	Vivado	262659 (493.7%)	33517 (63%)	6.42×	476794 (448.1%)	74224 (69.76%)	7.84×	16 (7.27%)	16 (7.27%)	-
	Quartus	155750 (193.8%)	19908 (24.78%)	7.82×	246519 (112.1%)	39185 (17.83%)	6.29×	12 (7.69%)	12（7.69%）	-

表5：资源利用率及综合设计采用优化后的利用率降低情况。 **资源利用率，以及使用优化进行综合设计时利用率的降低**。

，但能够模拟优化后的版本；对于坐标旋转数字计算机案例研究，采用优化后获得了近 24× 倍加速比。

5.3.3 综合。我们使用 AMD Vivado 2024.02 和 Quartus Prime Pro 2025 对各案例研究中的设计进行综合，以测试 Verilog 的可移植性；此外，还成功在 Pynq Z1 上运行了生成的 Vivado 设计。各综合工具的结果见表5，其中展示了资源利用率，以及启用 JuliaHLS 优化后资源利用率的降低情况。请注意，若不启用 JuliaHLS 优化，二维卷积设计无法适配 AMD 或 Altera 的 FPGA；启用优化后则可以。总体而言，这些结果清楚地表明，我们实现的优化对于确保资源得到高效利用、并使设计能够适配 FPGA 至关重要。

5.4 与其他工具链的比较

为了评估 JuliaHLS 相较于其他工具链的效果，我们将其与基于 MLIR、采用动态调度的 Dynamatic v2.0 工具链进行比较 [26], ; 该工具链使用自定义的 MLIR后端将 C语言编译为 VHDL。我们用 C语言重写了 ‘implicit_else’、‘cordic’ 和 ‘conv2d_im2col’ 程序，然后使用 Dynamatic 的默认设置进行编译。

‘implicit_else’ 程序是唯一一个 Dynamatic 能够正确编译，且生成的设计满足 100 兆赫时序约束的程序。‘cordic’ 程序未能满足时序要求，而 ‘conv2d_im2col’ 程序则完全无法编译，因为其中一个提升遍在将 C风格数组提升为‘仿射方言’时导致工具崩溃。经过分析，我们认为 ‘cordic’ 的时序违例是因为标准缓冲区插入算法过于保守，这意味着 Dynamatic 生成的设计流水线化程度不足，因此性能结果仅能提供一个下界。

性能结果和资源使用量见表6，结果显示，与 JuliaHLS 相比，Dynamatic 设计的延迟更低，使用的资源也更少。

6 结论

总体而言，第5节所展示的结果表明，JuliaHLS 工具链能够将 Julia 程序编译为性能出色且与供应商无关的硬件描述语言，是一种切实可行的方案；其吞吐量最高可达到 Dynamatic v2 等“最先进”工具链的 82.6%。0 [26]同时，它还原生支持线性代数。

该工具链也是对 Biggs 等人提出的第一代 Julia HLS 工具链的一次重大变革。 [9], 它不仅能够编译简单的

设计	工具链	查找表/自适应逻辑模块	触发器	DSPs	延迟（周期）	最大吞吐量（千次操作/秒）
隐式 else	JuliaHLS（本研究）	2.29%	2.52%	0%	13	27548.21
	Dynamatic v2.0	1.06%	0.63%	0%	3	33333.33
坐标旋转数字计算机	JuliaHLS（本研究）	9.01%	4.83%	9%	241	615.61
	Dynamatic v2.0	2.23%	0.48%	9%	97	1030.93

表6：JuliaHLS 与 Dynamatic v2.0 的性能和资源使用量对比。

控制流结构，还能处理矩阵和向量等基于存储器的对象，并支持二维卷积核这类复杂程序。我们还展示了使用高层次综合社区支持的现有框架来构建高层次综合工具链的优势，因为借助 CIRCT，我们可以专注于开发和完善 Julia 所特有的前端和初始 *High-to-PreCompile*工作流，而不必花费时间实现所有底层调度和生成例程。

如第5节所述，由于 Calyx 与 CIRCT 框架的集成目前仍有限，JuliaHLS 目前能够通过静态调度综合的设计也受到限制。随着 Calyx 与 CIRCT 的集成日趋成熟，这些限制应会消除，届时 JuliaHLS 工具链便能够使用动态调度或静态调度编译符合 Julia 惯用风格的代码。

与 Julia 的交互式解释器和即时编译器配合使用时，这一工具链有望提供独特的开发周期——通过透明地允许同一段代码在中央处理器和现场可编程门阵列上运行，开启交互式算法与加速器协同开发的可能性。因此，我们的工作能够让非硬件工程师更容易地对硬件加速器进行原型设计和部署，从而扩大设计和部署硬件加速器的用户群体。此外，JuliaHLS 能够缩短工程开发周期，并有助于消除算法与加速器协同设计过程中冗长的开发流程；这一流程目前限制了面向科学算法的定制硬件加速器的广泛应用 [15, 28]。

7 未来工作

尽管该工具链已经涵盖了 Julia 语言的很大一部分，并能够生成在功能上“正确”的硬件设计，但该工具及其所依赖的生态系统仍处于起步阶段。要生成能够与手工编写的寄存器传输级设计相竞争，并且性能超越 AMD Vitis 等成熟、由业界支持的 HLS 工具的设计，还需要持续进行开发和改进。总体而言，推动 JuliaHLS 工具链发展并在 Julia 中构建现场可编程门阵列生态系统，主要有三个方向：改进 Julia 基础设施、构建新的 MLIR 方言和变换，以及改进 CIRCT 框架和高层次综合方言。

改进 Julia 基础设施有两种方式：一是参与上游 Julia 的开发，以改进类型推断基础设施等方面；二是构建 FPGA 生态系统的其他组件。目前，如果不创建一个“自定义”Julia 版本，类型推断算法以及编译器整体都很难修改，正如 Fischer 和 Saba 所示 [19]。未来工作可以通过允许对 Julia 的“中端编译器”算法及其复用部分进行更细粒度的控制，改进 MLIRInterprer 与 Julia 编译器的集成。

JuliaHLS 生态系统的下一步应当是允许用户在 Julia 中实际使用编译后的加速器，而不仅仅是设计它们。为此，可以使用 Xilinx Runtime (XRT) 或 Coyote v2 [43] 等框架封装生成的 IP 核，从而实现核与 Julia 之间数据和控制信号的传递。

另一个方向是继续开发与硬件相关的 MLIR 方言，例如支持定点算术的 quant 方言。我们的工作针对定点数尝试了这一方法，但由于现有使用该方言处理此用例的目标平台都是闭源的，因此没有可用的开源遍或变换。此外，我们发现许多科学与工程代码依赖标准库中的数学运算符，例如三角函数、平方根、幂等。开发这些函数的“FPGA 标准库”或相应生成器，将使 Julia 中更多应用能够面向 FPGA，以实现硬件加速。

最后，研究发现，CIRCT 库的功能是限制硬件性能、可纳入功能以及在 Julia 中执行协同仿真能力的主要因素。因此，未来工作可着重于进一步开发“calyx”和“handshake”方言，从而在保持完全开源的同时，生成性能超越最先进设计的设计，并与更广泛的 CIRCT 生态系统集成。此外，能够将 CIRCT 硬件方言编译为 LLVM 代码（以便通过即时编译运行）的“arcilator”项目 [16] 将直接支持在 Julia REPL 中进行门级协同仿真，从而将硬件设计的验证过程集成到工具链本身。

参考文献

[1] R. Aggleton, L.E. Ardila-Perez, F.A. Ball, M.N. Balzer, G. Boudoul, J. Brooke, M. Caselle, L. Calligaris, D. Cieri, E. Clement, S. Dutta, G. Hall, K. Harder, P.R. Hobson, G.M. Iles, T.O. James, K. Manolopoulos, T. Matsushita, A.D. Morton, D. Newbold, S. Paramesvaran, M. Pesaresi, N. Pozzobon, I.D. Reid, A.W. Rose, O. Sander, C. Shepherd-Themistocleous, A. Shtipliyski, T. Schuh, L. Skinnari, S.P. Summers, A. Tapper, A. Thea, I. Tomalin, K. Uchida, P. Vichoudis, S. Viret, and M. Weber. 2017. 面向高亮度大型强子对撞机 CMS实验一级触发器的现场可编程门阵列径迹寻找器。仪器学杂志 12, 12（2017年12月），P12019。doi:10.1088/1748-0221/12/12/P12019[2] Nicolas Bohm Agostini, Serena Curzel, Jeff Jun Zhang, Ankur Limaye, Cheng Tan, Vinay Amatya, Marco Minutoli, Vito Giovanni Castellana, Joseph Manzano, David Brooks, Gu-Yeon Wei, and Antonino Tumeo. 2022. 从 Python 到硅：SODA工具链。*IEEE Micro* 42, 5（2022年9月），78–88。doi:10.1109/ MM.2022.3178580[3] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. 2010. 使用 Haskell 对同步硬件进行结构化描述。在*2010*年第*13*届 *Euromicro* 数字系统设计会议：架构、方法与工具。IEEE，法国里尔，714 721 10 1109 2010 21 –。doi: . /DSD. . [4] Suhail Basalama and Jason Cong. . Stream-HLS：迈向自动数据流加速。在 *2025* 年 *ACM/SIGDA* 现场可编程门阵列国际研讨会论文集（*FPGA '25*）。计算机协会，美国纽约州纽约市，2025 103 114 10 1145 3706628 3708878 –。doi: . / . Samit Basu. . Rust 作为[5] 硬件描述语言。在加速器设计语言、工具与技术研讨会（*LATTE '24*）。美国加利福尼亚州圣迭戈。[6] Tim Besard, Christophe Foket, and Bjorn De Sutter. 2019. 高效的可扩展编程：释放 Julia 在图形处理器上的潜力。*IEEE* 并行与分布式系统汇刊 2024 30, 4 2019 827 841 10 1109 2018 2872064 ( ), –。doi: . /TPDS. . Tim [7] Besard, Pieter Verstraete, and Bjorn De Sutter. . 使用 Julia 进行高层次图形处理器编程。*arXiv* 预印本 *arXiv:1604.03410*（2016）。[8] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. 2017. Julia：数值计算的新思路。*SIAM* 评论 59, 1（2017），65–98。doi:10.1137/141000671[9] Benjamin Biggs, Ian McInerney, Eric C. Kerrigan, and George A. Constantinides. 2022. 使用 Julia 语言进行高层次综合。在第*2*届加速器设计语言、工具与技术研讨会论文集（*LATTE'22*）。瑞士洛桑。arXiv:2201.11522[10] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. 2013. LegUp：面向现场可编程门阵列处理器/加速器系统的开源高层次综合工具。*ACM* 嵌入式计算系统汇刊 13, 2（2013年9月），1–27。doi:10.1145/2514740[11] Fredrik Bagge Carlson, Cody Tapscott, Gabriel Baraldi, and Chris Rackauckas. 2025. C代码生成并非必要：直接生成二进制代码，无需经过 C语言。面向基于模型的工程的 Julia 代码编译与部署。arXiv: 2502 01128 [ ] 2502 01128 . eess.SY [12] https://arxiv.org/abs/ 。S. Cecchi, L. Palestini, P. Peretti, A. Primavera, F. Piazza, F. Capman, S. Thabuteau, C. Levy, J.-F. Bonastre, A. Lattanzi, E. Ciavattini, F. Bettarelli, R. Toppi, E. Capucci, F. Ferrandi, M. Lattuada, C. Pilato, D. Sciuto, W. Luk, and J.G. De Figueiredo Coutinho. 2012. 异构多核平台的软硬件协同设计：*hArtes*工具链。施普林格·荷兰，第 In 汽车音频章，155 192 10 1007 978 94 007 1406 9_5 –。doi: . / - - - - Radosław Cieszewski, Krzysztof [13] Pozniak, and Ryszard Romaniuk. 2014. 基于 Python 的高层次综合编译器。在*2014*年天文学、通信、工业与高能物理实验中的光子学应用，第9290卷。SPIE，981–988。doi:10.1117/12.2075988[14] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. 2022. 当今的 FPGA高层次综合：成果、挑战与机遇。*ACM* 可重构技术与系统汇刊 15, 4（2022），51:1–51:42。doi:10.1145/3530775[15] Serena Curzel, Michele Fiorito, Patricia Lopez Cueva, Tiago Jorge, Thanassis Tsiodras, and Fabrizio Ferrandi. 2023. 从 Simulink模型到面向航空航天应用的现场可编程门阵列的综合方法探索。在第*20*届 *ACM* 计算前沿国际会议论文集（意大利博洛尼亚）（*CF '23*）。计算机协会，美国纽约州纽约市，243–249。doi:10. 1145/3587135.3592766[16]Martin Erhart, Fabian Schuiki, Zachary Yedidia, Bea Healy, and Tobias Grosser. 2023. 在 CIRCT 中进行快速且周期精确的硬件仿真。在*2023*年 *LLVM* 开发者会议论文集。

https://llvm.org/devmtg/2023-10/slides/techtalks/Erhart-Arcilator-FastAndCycleAccurateHardwareSimulationInCIRCT.pdf

[17] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. 2021. 特邀论文：Bambu：用于复杂应用高层次综合的开源研究框架。在*2021*年第*58*届 *ACM/IEEE* 设计自动化会议（*DAC*）。IEEE，美国加利福尼亚州旧金山，1327–1330。doi:10.1109/DAC18074.2021.9586110[18] Fabrizio Ferrandi、Michele Fiorito、Claudio Barone、Giovanni Gozzi 和 Serena Curzel。2024年。欧洲空间技术研究背景下的高层次综合发展。载于 第 *15* 届并行编程与众核架构运行时管理技术研讨会暨第 *13* 届多核嵌入式计算平台设计工具与架构研讨会（*PARMA-DITAM 2024*）（开放获取信息学系列（*OASIcs*），第 *116* 卷），João Bispo、Sotirios Xydis、Serena Curzel 和 Luís Miguel Sousa（编）。Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 德国 Dagstuhl，1:1–1:12。doi:10.4230/OASIcs.PARMA-DITAM.2024.1[19] Keno Fischer 和 Elliot Saba。2018年。将 Julia 程序和机器学习模型完整自动编译至云端 TPU。arXiv:1810.09868 [cs.PL] https://arxiv.org/abs/ 1810.09868[20] Marco Gerards、Christiaan Baaij、Jan Kuper 和 Matthijs Kooijman。2011年。使用 C 进行硬件描述中的高阶抽象λaSH。载于 *2011* 年第 *14* 届 *Euromicro* 数字系统设计会议。IEEE，芬兰奥卢，495–502。doi:10.1109/ DSD.2011.69[21] Timothy C Germann。2021年。百亿亿次级计算项目中的协同设计。《国际高性能计算应用期刊》35，6（2021年11月），–。doi: 。/ Mosè Giordano。。使用 Julia 语言进行异构计算：从 A64FX 到 IPU。巴塞罗那超级计算中心研究研讨会。https://www.bsc.es/research-and-development/research-seminars/sors-heterogeneous-computing-the-julia-language-a64fx-the-ipu 访问日期：2025年1月28日。[23] Per Haglund、Oskar Mencer、Wayne Luk 和 Benjamin Tai。2003年。使用脚本语言进行硬件设计。载于 现场可编程逻辑与应用（*FPL*）*2003*（《计算机科学讲义》，第 *2778* 卷）。施普林格，葡萄牙里斯本，1040 1043 10 1007 978 3 540 45234 8_115 –。doi: 。/ - - - - Mathew [24] Hall 和 Vaughn Betz。。从 TensorFlow 计算图到查找表和连线：自动化稀疏且物理感知的 CNN 硬件生成。载于 *2020* 年国际现场可编程技术会议（*ICFPT*）。IEEE，美国夏威夷州毛伊岛，2020 56 65 10 1109 51103 2020 00017 –。doi: 。/ ICFPT 。。[25] Michael A Heroux、Lois Curfman McInnes、James Ahrens、Todd Gamblin、Timothy C Germann、Xiaoye Sherry Li、Kathryn Mohror、Todd Munson、Sameer Shende、Rajeev Thakur、Jeffrey Vetter 和 James Willenbring。2024年。ECP 库和工具：概览。《国际高性能计算应用期刊》38, 5 2024 381 408 10 1177 10943420241271005（2024年9月），–。doi: 。/ Lana Josipović、Andrea Guerrieri 和 Paolo Ienne[26]。。特邀教程：Dynamatic：从 C/C++ 到动态调度电路。载于 *2020* 年 *ACM/SIGDA* 国际现场可编程门阵列研讨会论文集（美国加利福尼亚州西赛德）（*FPGA '20*）。计算机协会，美国纽约州纽约市，2020 1 10 10 1145 3373087 3375391 –。doi: 。/ 。 Kristaps [27] Jurkans 和 Charles Fox。。面向机器人和物联网的 Python 子集到数字逻辑数据流编译器。载于 *2023 IEEE* 第 *22* 届计算与通信中的信任、安全与隐私国际会议（*TrustCom*）。IEEE，英国埃克塞特，2023 1893 1899 10 1109 60117 2023 00257 –。doi: 。/TrustCom 。。[28] Christoforos Kachris。。大语言模型硬件加速器综述。《应用科学》2025 15, 2 2025 10 3390 15020586（）。doi: 。/ [29] app Yi-Hsiang Lai、Yuze Chi、Yuwei Hu、Jie Wang、Cody Hao Yu、Yuan Zhou、Jason Cong 和 Zhiru Zhang。2019年。HeteroCL：面向软件定义可重构计算的多范式编程基础设施。载于 *2019* 年 *ACM/SIGDA* 国际现场可编程门阵列研讨会论文集 - *FPGA ' 19* 242 251 10 1145 3289602 3293910。ACM，美国加利福尼亚州西赛德，–。doi: 。/ 。[30] Chris Lattner 和 Vikram Adve。。LLVM：用于终身程序分析与转换的编译框架。载于 国际代码生成与优化研讨会论文集：反馈导向与运行时优化（美国加利福尼亚州帕洛阿尔托）（*CGO '04*）。IEEE 计算机学会，美国，75。[31] Chris Lattner、Mehdi Amini、Uday Bondhugula、Albert Cohen、Andy Davis、Jacques Pienaar、River Riddle、Tatiana Shpeisman、Nicolas Vasilache 和 Oleksandr Zinenko。2021年。MLIR：扩展面向领域专用计算的编译器基础设施。载于 *2021 IEEE/ACM* 代码生成与优化国际研讨会（*CGO*）。IEEE，韩国首尔，2–14。doi:10.1109/CGO51591. 2021.9370308[32] Marco Lattuada、Fabrizio Ferrandi 和 Maxime Perrotin。2016年。航空航天系统中 FPGA 加速器的计算机辅助设计与集成。载于 *IEEE* 航空航天会议论文集。IEEE，1–11。doi:10.1109/AERO.2016. 7500675[33] Charles E. Leiserson、Neil C. Thompson、Joel S. Emer、Bradley C. Kuszmaul、Butler W. Lampson、Daniel Sanchez 和 Tao B. Schardl。2020年。顶端仍有充足空间：摩尔定律之后，什么将推动计算机性能提升？《科学》368，6495（2020年），eaam9744。arXiv:https://www.science.org/doi/pdf/10.1126/science.aam9744 doi:10.1126/science.aam9744[34] Geng-Ming Liang、Chuan-Yue Yuan、Meng-Shiun Yu、Tai-Liang Chen、Kuan-Hsun Chen 和 Jenq-Kuen Lee。2023年。MLIR HLS 适配器对 LLVM 中间表示的支持。载于 第 *51* 届国际并行处理会议研讨会论文集（*ICPP Workshops '22*）。计算机协会，美国

美国纽约州纽约市，1 8 10 1145 3547276 3548515 –。doi: 。 / 。LLVM 项目。。电路 IR 编译器与工具（ CIRCT ）。https://github. co m/llvm/circt。https://github.com/llvm/circt。[36] Gaëtan Lounes、Robin Gerzaguet 和 Matthieu Gautier。2025年。利用异构编译实现高级综合的灵活前端。载于 *RAPIDO '25*：设计快速仿真与性能评估论文集。ACM，西班牙巴塞罗那，37 42 10 1145 3721848 3721854 –。doi: 。 / 。Gauthier Lovic 和 Ishikawa Yohei。。HDLRuby：用于硬件描述的 Ruby 扩展及其到可综合 Verilog 硬件描述语言的转换。*ACM* 嵌入式计算系统汇刊23，5（2023年2月），73:1–73:26。doi:10. 1145/3581757[38] Kingshuk Majumder 和 Uday Bondhugula。2024年。HIR：一种基于 MLIR 的硬件加速器描述中间表示。载于 第 *28* 届 *ACM* 编程语言与操作系统架构支持国际会议论文集，第 *4* 卷（*ASPLOS '23*）。计算机协会，美国纽约州纽约市，189 201 10 1145 3623278 3624767 –。doi: 。 / 。William S. Moses、Lorenzo Chelini、Ruizhe Zhao 和 Oleksandr Zinenko。2021年。Polygeist：将 C 提升为多面体 MLIR。载于 *2021* 年第 *30* 届并行架构与编译技术国际会议（*PACT*）。IEEE，美国佐治亚州亚特兰大，45 59 10 1109 52795 2021 00011 –。doi: 。 /PACT 。 。Rachit Nigam、Samuel Thomas、Zhijing Li 和 Adrian Sampson。2021年。面向加速器生成器的编译器基础设施。载于 第 *26* 届 *ACM* 编程语言与操作系统架构支持国际会议论文集（*ASPLOS '21*）。ACM，线上，美国，804–817。doi:10.1145/3445814.3446712[41] Morten Borup Petersen、Paolo Ienne-EPFL LAP 和 Stephen Neuendorffer。2022年。*MLIR* 中的动态调度 *HLS* 流程。博士论文。硕士论文。洛桑联邦理工学院。[42] Ariel Podlubne、Johannes Mey、Andreas Andreou、Sergio Pertuz、Uwe Aßmann 和 Diana Göhringer。2024年。利用混合调度器为机器人系统进行基于模型的硬件/软件架构生成。*IEEE* 计算机汇刊。73，7（2024年7月），1640–1654。doi:10.1109/TC.2023.3323804[43] Benjamin Ramhorst、Dario Korolija、Maximilian Jakob Heer、Jonas Dann、Luhao Liu 和 Gustavo Alonso。2025年。Coyote v2：提升数据中心 FPGA 的抽象层次。*arXiv* 2504.21538（2025年4月）。arXiv:2504.21538 [cs] doi:10.48550/arXiv.2504.21538[44] Gabriel Rodriguez-Canal、Nick Brown、Tim Dykes、Jess Jones 和 Utz-Uwe Haus。2023年。Fortran 高级综合：降低加速 HPC 的门槛

FPGA 上的代码。载于*2023*年第三十三届国际现场可编程逻辑与应用会议（*FPL*）。IEEE，瑞典哥德堡，10–18。doi:10.1109/ FPL60245.2023.00010[45] Gabriel Rodriguez-Canal、Nick Brown、Maurice Jamieson、Emilien Bauer、Anton Lydike 和 Tobias Grosser。2023年。Stencil-HMLS：一种针对 FPGA 上模板代码自动优化的多层方法。载于*SC '23*国际高性能计算、网络、存储与分析会议研讨会论文集（*SC-W '23*）。计算机协会，美国科罗拉多州丹佛，556–565。doi:10.1145/3624062.3624543[46] Marion Sudvarg、Longhao Huang、Boran Yang、Blake Bal、Jeremy Buhler、Leonardo Di Venere、Davide Serini 和 James Buckley。2025年。基于 FPGA 的数据处理：在先进粒子天体物理望远镜南极验证平台（ADAPT）上使用高层次综合。载于第*39*届国际宇宙射线会议（*ICRC2025*）。《科学会议论文集》，瑞士日内瓦，1–8。[47] Mike Urbach 和 Morten B Petersen。2022年。使用 MLIR 和 CIRCT 将 PyTorch 转换为 SystemVerilog 的高层次综合。载于第二届加速器设计语言、工具与技术研讨会（*LATTE'22*）。瑞士洛桑。[48] Jack E. Volder。1959年。CORDIC 三角函数计算技术。*IRE*电子计算机汇刊EC-8，3（1959），330–334。doi:10.1109/TEC. 1959.5222693[49] Zishen Wan、Bo Yu、Thomas Yuang Li、Jie Tang、Yuhao Zhu、Yu Wang、Arijit Raychowdhury 和 Shaoshan Liu。2021年。基于 FPGA 的机器人计算综述。*IEEE*电路与系统杂志21，2（2021），48–74。doi:10.1109/ MCAS.2021.3071609[50] Hanchen Ye、Cong Hao、Jianyi Cheng、Hyunmin Jeong、Jack Huang、Stephen Neuendorffer 和 Deming Chen。2022年。ScaleHLS：基于多级中间表示的新型可扩展高级综合框架。载于第*28*届*IEEE*国际高性能计算机体系结构研讨会（*HPCA-28* 15 10 1109 53966 2022 00060 ）。IEEE，韩国首尔，。. doi: . /HPCA . . Hanchen Ye、Hyegang Jun 和 Deming Chen。. HIDA：一种用于高层次综合的分层数据流编译器。载于第*29*届*ACM*编程语言与操作系统体系结构支持国际会议论文集，第一卷（美国加利福尼亚州拉霍亚）（*ASPLOS '24*）。计算机协会，美国纽约州纽约市，215–230。doi:10.1145/3617232. 3624850[51] 2024