面向晶圆级模板计算的 MLIR 降低流水线

Nicolai Stawinoga*†
柏林工业大学，德国柏林
nicolai@aes.tu-berlin.de

David Katz*
爱丁堡大学 EPCC，英国爱丁堡
d.kacs@sms.ed.ac.uk

Anton Lydike
信息学院，爱丁堡大学，英国爱丁堡
anton.lydike@ed.ac.uk

Justs Zarins
EPCC 爱丁堡大学 爱丁堡，英国
j.zarins@epcc.ed.ac.uk

Nick Brown
EPCC 爱丁堡大学 爱丁堡，英国
nick.brown@ed.ac.uk

George Bisbas†
伦敦帝国理工学院 伦敦，英国
georgios.a.bisbas@gmail.com

Tobias Grosser
剑桥大学 英国剑桥
tobias.grosser@cst.cam.ac.uk

arXiv:2601.17754v1 [cs.DC] 25 Jan 2026

摘要

Cerebras 晶圆级引擎（WSE）以超过 900,000 个计算单元的前所未有规模提供性能，所有计算单元均通过单晶圆片上互连连接。WSE 最初为人工智能而设计，其 WSE 架构同样非常适合高性能计算（HPC）。然而，其分布式异步编程模型与针对给定数学程序描述通常推导出的简单顺序程序或批量同步程序存在显著差异。面向 WSE 移植现有代码时，需要进行定制化重新实现。由于 MLIR 等编译器缺乏 WSE 支持，几乎无法实现这一过程的自动化。

模板在 HPC 中无处不在。在本文中，我们探讨这样一种假设：编译器可以利用有关模板的领域特定信息，自动针对 WSE 进行优化，而无需应用级代码更改。我们提出了一条编译器流水线，可将基于模板的内核转换为针对 WSE 高度优化的 CSL 代码，从而弥合问题数学表示与 WSE 异步执行模型之间的语义鸿沟。基于三个 HPC 编程技术中的五项基准测试，并在 Cerebras WSE2 和 WSE3 上运行，我们的方法达到了与手动优化代码相当，甚至略高的性能。此外，在无需任何应用级代码更改的情况下，采用我们的方法时，WSE3 的性能比 128 个 Nvidia A100 GPU 快约 14×，比基于 CPU 的 Cray-EX 超级计算机的 128 个节点快 20×。

*两位作者对本工作的贡献相同
†贡献该工作时的所属机构

作者联系方式：Nicolai Stawinoga，柏林工业大学，柏林，德国，nicolai@aes.tu-berlin.de；David Katz，EPCC，爱丁堡大学，爱丁堡，英国，d.kacs@sms.ed.ac.uk；Anton Lydike，信息学院，爱丁堡大学，爱丁堡，英国，anton.lydike@ed.ac.uk；Justs Zarins，EPCC，爱丁堡大学，爱丁堡，英国，j.zarins@epcc.ed.ac.uk；Nick Brown，EPCC，爱丁堡大学，爱丁堡，英国，nick.brown@ed.ac.uk；George Bisbas，伦敦帝国理工学院，伦敦，英国，georgios.a.bisbas@gmail.com；Tobias Grosser，剑桥大学，剑桥，英国，tobias.grosser@cst.cam.ac.uk。

本作品采用知识共享署名 4.0 国际许可协议授权。*ASPLOS '26*，美国宾夕法尼亚州匹兹堡© 2026年版权归所有者/作者所有。ACM ISBN 979-8-4007-2359-9/2026/03https://doi.org/10.1145/3779212.3790124

CCS 概念：• **计算机系统组织** → 多指令、多数据；脉动阵列；多核架构；• **计算方法** → 并行编程语言；• **软件及其工程** →**领域专用语言**。

关键词：Cerebras、晶圆级引擎、领域专用语言、中间表示、模板计算、MLIR、xDSL

ACM 参考格式：
Nicolai Stawinoga、David Katz、Anton Lydike、Justs Zarins、Nick Brown、George Bisbas 和 Tobias Grosser。2026年。《面向晶圆级模板的 MLIR降低流程》。发表于第 *31* 届 *ACM* 编程语言与操作系统架构支持国际会议论文集，第 *2* 卷（*ASPLOS '26*），*2026*年*3*月*22*日至*26*日，美国宾夕法尼亚州匹兹堡。ACM，纽约州纽约，美国，文章编号 1，16页。https://doi.org/10.1145/3779212.3790124

1 引言

Cerebras 晶圆级引擎（WSE） [26] 是一个脉动阵列 [22]横跨整片晶圆，由 900,000 个相互独立且互连的处理单元（PE）构成，位于WSE3 上。它最初被设计为一种新型人工智能加速器，如今已发展到第三代架构。Cerebras WSE 已应用于多种具有重要影响的高性能计算应用 [20, 28, 32,33, 41, 44, 45, 47, 49]。事实上，Ltaief 等人 [28]证明，由于拥有大量分布式静态随机存取存储器，WSE 能够提供其他架构受内存瓶颈所限时无法实现的性能。高性能计算工作负载对极高性能的潜在需求，使得人们对将 WSE 用于此类工作负载抱有浓厚兴趣；然而，实际使用该架构的人寥寥无几，主要障碍在于移植代码所需的工作量巨大。

尽管将代码重写为 Cerebras 专有的通用 CSL 编程语言需要耗费大量时间，但程序员面临的首要挑战是进行算法变换，以适应 WSE。重新构建算法需要程序员投入大量时间和专业知识，且往往涉及架构专用技术。因此，尽管已有若干卓有成效的尝试，将简单内核或基准测试移植到 WSE 上，但由于这些编程挑战，这项技术尚未在 HPC 领域得到广泛应用或普及。

本文介绍一种基于 MLIR 的编译器流程，无需修改代码即可使其在 WSE 上运行。我们的总体研究假设是：利用用户代码中存在的领域专用抽象，并以此驱动编译器自动化，就有可能解决 WSE 这类特殊架构在可编程性和性能方面的根本挑战。我们面向两个科学计算域中的三种前端提出，诸如 MLIR 这类最初为人工智能工作负载而设计的软件工具，可用于解决如何在高性能计算（HPC）中发挥基于人工智能的硬件优势这一问题所涉及的一些根本挑战。

本文的贡献如下：

- 讨论 WSE 的高度异步执行模型，以及针对该模型进行代码生成时它对程序表示的影响（第2节）。
- 一组中间领域专用和领域无关方言，用于将模板表达式转换为 CSL
  通过我们的低级 `csl-ir` 方言重新实现了 CSL 的很大一部分（第4节）。
- 一套降低变换策略，用于从更高层级的 MLIR 抽象生成`csl-ir` 方言（第5节）。
- 对我们面向 WSE2 和 WSE3 的模板降级流水线进行了评估，最终表明，我们的方法比支持多代 WSE 的手写代码性能略胜一筹。

WSE——据我们所知，这可能是撰写本文时首次对 WSE2 和 WSE3 的性能进行比较（第 6 节）。

2 动机：Cerebras WSE 及其异步执行模型

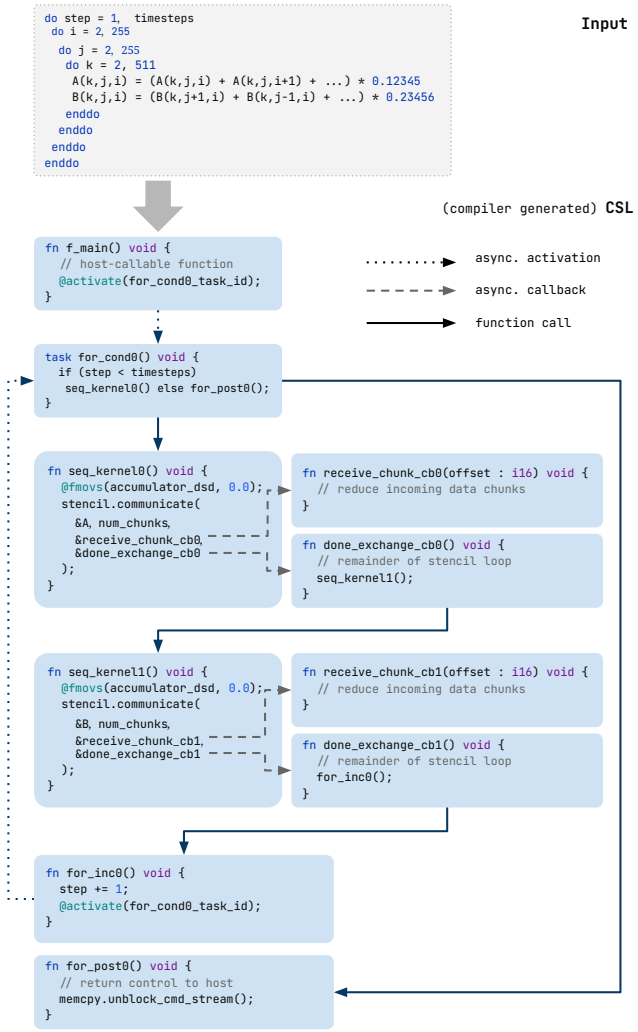

图1。CSL 中围绕异步数据交换的时间步循环。

Cerebras 晶圆级引擎（WSE）具有独特且高度异步的执行模型，其特点由计算架构所决定。它采用脉动阵列，最多包含 900,000 个处理单元（PE），每个处理单元都由一个计算核心、路由器、程序计数器、时钟和 48KB SRAM 存储器组成。处理单元可以单独编程，彼此完全独立地运行，并管理各自的存储器；每个周期内，它们都能够执行一次 128 位存储器读取和一次 64 位存储器写入。本研究采用的 WSE2（WSE3）架构具有惊人的 40GB（44GB）片上存储器总容量，但并不

构成统一的存储系统，也不存在缓存或存储层次结构。

处理单元之间的数据通过四条双向通信通道显式共享，这些通道将处理单元的路由器与其正北、正南、正东和正西相邻处理单元的路由器相连。另有一条名为斜坡通道的通道，将路由器与连接的核心相连。一个 32 位消息（称为波元）可在单个时钟周期内沿每个方向在相邻处理单元之间传输，因此 WSE3 的互连结构峰值总带宽达到 214 Pb/s。24 条虚拟通道（称为颜色）为程序员提供了一种抽象，用于配置路由和交换机。异步数据传输完成后，处理单元可以将一个内部绑定的任务标记为解除阻塞或已准备好激活。

2022年，Cerebras 发布了 Cerebras 软件语言（CSL）SDK，使用户能够直接针对 WSE 的架构进行编程。CSL 在语法上基于 Zig，并包含许多扩展，以显式支持 WSE，例如任务、波元、颜色和数据结构描述符（DSD），这些描述符能够高效地对数组执行操作。然而，实际中暴露 WSE 执行模型的做法，使 CSL 编程与 Zig 编程有着根本性的不同。除了深入了解架构之外，程序员还必须从根本上改造算法，将其从适用于中央处理器或图形处理器的形式，转变为适合 WSE 的形式。

图1 展示了 Fortran（上）和 CSL（下）中包围朴素雅可比模板程序的时间步循环的表示方式。这里假设时间步中的两个模板运算需要处理单元之间进行异步通信。因此，尽管 CSL 提供了各种循环结构，但由于 CSL 缺乏在代码块中实现重新同步的机制（例如，在单线程和多线程上下文中分别使用 等待/屏障 ），因此无法将该程序表示为顶层循环。因此，任何包围通信等异步操作的控制流都必须转换为任务和回调函数。程序员必须将程序分解为独立的任务，然后根据数据到达情况安排这些任务执行。在面向中央处理器和传统加速器的语言中，诸如 执行步骤 *= 1,* 时间步 这样的外层循环是一种简单的语言结构；然而，在 WSE 上，由于外层循环的迭代之间需要通信，同一算法必须改写为由回调函数驱动的任务。

对 `stencil.communicate()` 在 *seq_kernel0()* 和*seq_kernel1()* 中的调用图 1封装了用于分块数据交换的样板代码，并在内部使用了额外的任务。架构的实际限制，尤其是每个处理单元的内存容量有限，严格限制了缓冲区的大小，因此数据必须以数据块为单位进行交换。因此，该函数接受两个用户提供的回调（如虚线箭头所示，这些回调会被间接触发）；第一个回调会在处理每个传入数据块时触发，第二个回调则在整个数据交换完成后执行。随后，算法的整体执行流程会从第二个回调继续进行，这展示了程序员在尝试将一个看似简单的内核移植到 WSE 时所面临的复杂性，因为 CSL 直接暴露了底层架构概念，使编程变得复杂且容易出错。

2.1 基于参与者的异步执行模型

图 1 所示程序表示中的语义鸿沟，直接源于底层架构执行模型的差异。与传统的数据并行加速器不同，WSE 主要采用任务并行架构，其方式类似于参与者范式。该形式体系最初以用于人工智能的通用模块化 *ACTOR* 形式体系为题发表 1973 [17]（距今已逾 50 年），早于硬件和软件实现的出现，却能很好地描述 WSE 在硬件和软件层面的执行模型。

在硬件层面，我们将处理单元视为硬件参与者，并将其看作静态排列的单元。参与者模型的一个关键特征是，数据以消息的形式在参与者之间传递；在 WSE 上，每个处理单元都包含一组队列，充当邮箱以接收数据。参与者还拥有各自的状态；在 WSE 上，每个处理单元独立运行，并拥有自己的时钟和私有内存。硬件参与者之间可以传递两种消息，或称波元：包含数据载荷的数据波元，以及用于指示另一个参与者更新其路由配置的控制波元。根据处理单元的状态，消息会被转发或消耗；在某些情况下，这会激活一个软件参与者。作为硬件参与者的处理单元具有物理位置，这一概念并不适用于参与者模型的软件实现。尽管有库支持在任意处理单元之间传递消息，但在实践中，利用空间邻近性实现邻居间通信对性能至关重要。

在软件层面，我们将任务视为软件参与者。每个处理单元中运行的代码通过任务实现参与者模型的软件版本。CSL 提供三种任务类型，每种都具备硬件支持：

- 数据_任务是消息监听器，在接收到消息或数据波元时触发；消息或数据波元以数据形式存在于它们所配置监听的队列中。
- 控制_任务与数据任务类似，但由控制波元触发；这类消息不携带数据载荷。
- 本地_任务只能在内部触发，在完成数据的异步发送或接收时尤其适合作为回调函数。

由于程序状态由 PE 维护，任务具有共享的程序状态，尽管程序员理论上可以强制使用独立的状态。作为软件参与者，任务并不表现出真正的并行，因为每个 PE 都严格采用单线程运行；相反，任务提供了一种异步执行模型。在任务之间传递执行流会显著增加复杂性，文献中通常将其称为“回调地狱”。 [12]主要挑战在于，CSL 将这一问题直接暴露给程序员，却没有提供高级语言抽象，例如期物 [10]，来帮助解决这一问题。将程序从同步表示重构为异步模型，会产生传递执行流的续延复杂性问题 [48] ；正如第5节所述，这是我们在中间表示转换中必须解决的一项挑战。

总而言之，为 WSE 编程所涉及的大部分复杂性，都源于必须将同步算法映射到 WSE 的异步特性上。不过需要注意的是，CSL 严格来说并不是一种基于参与者的编程语言，因此不会让用户面临异步类参与者编程中常见的问题，包括在无屏障并行中管理交互的复杂性、防止竞争条件等。

2.2 脱离经典数据流架构

虽然 WSE 属于数据流架构，但其设计的粗粒度特性使其与现场可编程门阵列等其他细粒度数据流架构截然不同，这给分布式模板计算和编译带来了独特的挑战。与细粒度数据流架构中以流式方式处理数据的单个逻辑单元不同，WSE 中的每个处理单元都包含一颗运行任务且计算能力强得多的向量化中央处理器。因此，现场可编程门阵列等细粒度架构上自然而然采用的并行流水线方法并不适用于 WSE，因为其工作负载既包含本地计算，也包含远程计算。实际上，在 WSE 上，程序员必须结合运用并行方法和优化技术，同时兼顾数据流架构（例如，处理单元之间的通信涉及流式传输许多独立数据项）和分布式内存架构（例如，模板在处理单元之间按几何方式分解）；这种混合方法非常困难。

例如，利用移位缓冲区 [11] 作为现场可编程门阵列上的定制内存来实现高效的模板计算，是一种众所周知的技术 [6]。然而，WSE 的特性意味着这种方法既不可行（因为内存对于处理单元而言是本地且私有的），也不是最优方案（因为必须谨慎设计处理单元间通信）。相反，要在 WSE 上实现高效执行，必须采用不同的方法，将问题分解到这种数据流架构上；正如本节所述，需要利用异步任务，将本地计算与远程数据到达的处理交错进行。

3 编译器生态系统与模板

MLIR。 MLIR [24] 提供了一组可复用的中间表示方言及其间的变换；不同的方言可以混合到中间表示中，并独立进行操作。与更底层的 LLVM-IR [23], 类似，MLIR 遵循静态单赋值（SSA）形式。复用现有的方言和变换，可以大幅提高编译器基础设施的共享程度，显著降低开发新编译器所需的总体软件工作量。此外，MLIR 还是一个框架，编译器开发者可以在其中创建自己的方言和变换。该框架的可插拔特性支持构建深层编译链，其中包含范围广泛的可复用方言、后端或前端。

xDSL。 MLIR 的学习曲线对开发者而言相当陡峭。例如，他们必须理解 LLVM 的相关概念，使用 Tablegen 格式描述方言，并持续跟进快速演进的 MLIR 代码库。xDSL 是一款基于 Python 的编译器设计工具包，与 MLIR 完全兼容。方言在 Python 类中以 IRDL [13] 格式表示，便于快速探索和原型验证 MLIR 概念；待方案验证成熟后，再将其提交至 MLIR 代码库。MPI 方言就是这种做法的一个例子：它最初在 xDSL [4] 中开发，之后才被纳入 MLIR 标准。由于 xDSL 与 MLIR 完全兼容，编译期间，中间表示可以在两个框架之间自由传递。我们使用 xDSL 开发了本文所述的变换和优化。由于 xDSL 基于 MLIR，除非另有说明，我们会交替使用这两个术语。

清单 1 Fortran 模板代码示例草图：将两个网格单元中的值相加，并将结果乘以一个常数

```
1 执行i = 2, 255 2 执行 j = 2, 255 3 执行k = 2, 511
4 数据(k,j,i) = (数据(k,j,i) + 数据(k,j,i+1)) * 0.12345
5 结束执行6 结束执行7 结束执行
```

模板方言。 该模板方言最初作为 Open Earth Compiler 的一部分开发， [15] 现已成为 xDSL 的一部分。模板是一组相邻网格单元的几何排列，通过数值近似例程与某个特定的目标网格单元建立关联。清单 1给出了一个 Fortran 模板代码示例，该代码将两个网格单元中的值相加，再将结果乘以一个常数；清单 2 （见第 5 节）概述了相应的基于 SSA 的 MLIR，使用 模板 和 算术方言来表达这一计算。清单 1 第 1、2 和 3 行中的嵌套循环**已转换为模板应用**

操作，其中 模板。**访问**操作对应于清单 2 中对 Fortran 代码第 4 行数据数组的访问。在 模板。**应用**操作中，模板方言决定数据访问，而算术方言决定计算。需要注意的是，模板。**应用**运算符会遍历整个网格，其下界和上界由输入和输出字段的类型决定，并针对每个网格单元执行 模板。**应用**操作的主体。

来自 PSyclone、Devito 和 Flang 的模板生成。 PSy-clone [1] 是一种气候与天气领域专用语言，使科学家能够使用 Fortran 编写内核，同时抽象掉并行的底层细节。英国气象局下一代天气与海洋模型便使用了该语言 [30, 34]。PSyclone 编译器会分析科学家的代码，并在内部中间表示上执行优化和变换，例如在循环级别应用 OpenMP 或 OpenACC，分别实现线程并行和 GPU 加速。

Devito [5, 27, 29], 是一个开源的、基于 Python 的领域特定语言和编译器框架，广泛应用于学术界和工业界。它基于 SymPy [31], 符号领域特定语言，旨在简化 HPC 有限差分偏微分方程求解器的开发。Devito 被应用于多个行业，尤其是专注于地震与医疗工作负载的行业。

Bisbas 等人 [4] 强调，尽管这两种领域专用语言看起来大不相同，但它们在编译器优化方面有许多共同之处。因此，Bisbas 等人 [4] 通过 xDSL 将 PSyclone 和 Devito 与模板方言集成起来，旨在共享编译器基础设施。研究发现，借助 MLIR 生态系统，这两种领域专用语言都受益于更广泛社区（包括厂商）开发的编译器遍；事实上，这种集成还带来了中央处理器和图形处理器上的性能提升。这些优化将在第5.7节中进一步讨论。这促成了将模板方言与 Flang Fortran 编译器集成， [9], 而后者本身构建于 MLIR 之上。Flang 提供的性能明显落后于其他编译器，[7], 通过添加一个能够识别并提取 Fortran 代码中模板的编译器遍，研究证明可以缩小这一差距 [9].

正是这三个前端——PSyclone、Devito 和 Flang——被我们应用于本工作中，并在第6节中作为基准测试。事实上，Rodriguez-Canal 等人 实现了从模板 方言到高效现场可编程门阵列表示的降级，表明无需对前端语言或领域专用语言进行任何更改。我们在第4 节和5节中采用了相同的总体设计方法；在本例中，我们将模板方言转换为适用于 Cerebras WSE 的形式。

4 面向 WSE 的中间 MLIR 方言

构成我们基于 MLIR 的 WSE 编译器流程的组件主要分为两大类：方言以及作用于这些方言的变换。在本节中，我们为 Cerebras WSE 提出三种新的 MLIR方言：`csl-stencil`、`csl-wrapper` 和 `csl-ir`。图 2 概述了它们如何集成：现有的 模板 方言会降级为 `csl-stencil` 方言，随后封装在 `csl-wrapper` 方言中。最后，所有内容都会转换为 `csl-ir` 方言中的操作。

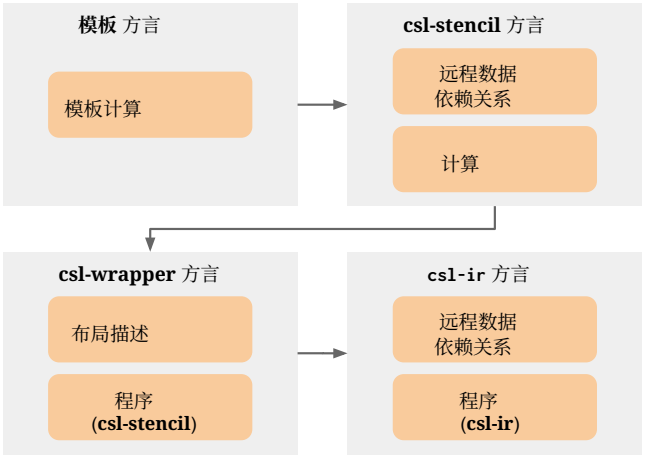

图 2. 通过使用三种 WSE MLIR 方言，我们逐步从模板的数学表示转向基于执行体的 WSE 执行模型。

4.1 csl-stencil 方言

模板从与架构无关的 `stencil`方言降级为 WSE 特定的 `csl-stencil` 方言，该方言明确指定了通信和模板计算操作。`csl-stencil` 方言包含以下操作：

1. `csl-stencil.prefetch` 描述将单个远程数据片段从其源位置提取到本地缓冲区的过程。2. `csl-stencil.apply` 包含两个区域，第一个区域描述对从远程依赖项接收的数据块 进行处理，第二个区域描述对本地持有的数据执行计算操作；这些操作会考虑第一个区域已执行的任何预处理。

3. `csl-stencil.access` 基于相对于当前索引的偏移量访问模板中的某个值：如果该值存储在本地，则直接访问；如果存储在远程位置，则从内部缓冲区访问。因此，该操作最终可能会降级为本地内存访问，或访问用于接收传入远程数据的缓冲区。

根据定义，模板计算描述的是对一组相邻值进行归约操作（参见清单 1）。`csl-stencil` 方言使我们能够在适用情况下执行双重部分归约。首先，通过支持将远程数据与本地数据分开归约，借助

`csl-stencil.apply` 操作的第一个区域。其次，将通信拆分为 n 数据块，并逐块归约远程数据。在这种情况下，第一个区域会执行 n 次，但能够立即将传入的数据块归约到单个累加器缓冲区中，从而显著降低总体内存需求；每个 PE 仅提供有限的 48千字节。为支持 [20], 提出的一项优化（已证明在 WSE 上效率很高），`csl-stencil.apply` 操作还允许将常量应用于传入数据，而不是将常量放在计算区域中。随后在流水线中，这些操作会映射为 CSL 原语，以零开销将常量应用于传入数据，从而有效地交错执行通信与计算。

4.2 csl-wrapper方言

CSL 程序由两个主要部分组成：程序本身，以及一个布局文件。布局文件是一种元程序，用于确定程序在 WSE 上的布局与布线，并包含编译时常量。CSL 采用分阶段编译方法：编译器执行此布局文件，然后利用它针对各个核心对程序进行特化和编译。在此阶段，标记为编译时 的 处理单元 程序的代码会被求值，参数会被降阶为常量，生成的处理单元程序则由 CSL SDK 进行积极优化。如果处理单元程序的参数值存在差异，则会分别编译所有遇到的变体。我们提供了 `csl-wrapper`方言来处理这种分阶段编译方法。它是一种领域无关方言，将程序级参数、布局元程序和处理单元程序封装在一起，并提供用于实例化程序模块和元程序模块的实用功能。它包含以下主要操作：

1. csl-wrapper**模块** 定义一个模块，其中包含可选的程序参数和两个区域。第一个区域控制 布局 在 WSE 上的布局，第二个区域则用于映射到处理单元的程序。2. csl-wrapper**导入** 将导入 CSL 专用库和辅助函数，例如 CSL *memcpy* 库，该库支持与主机之间的数据传输。

布局文件以 CSL 编写，通常包含一个循环嵌套，遍历 WSE 的 `x` 和 `y` 维度，并将内核分别放置到各个处理单元上。由于我们在每个处理单元上运行相同的程序，因此我们的方法中的布局对应于这个内层循环嵌套的一个视图。

4.3 csl-ir 方言

`csl-ir` 方言是对 CSL 编程语言很大一部分内容的直接重新实现，其中，语言中的构造在 `csl-ir` 中以相同的方式表示。这是因为，最终我们会调用一个打印器，它根据 `csl-ir` 中间表示生成 CSL 代码。因此，上文所述的所有操作最终都会被降级为 `csl-ir` 方言，然后由此生成 CSL 代码，作为 Cerebras SDK 编译器的输入。

5 WSE 的编译器变换

上一节介绍了用于表示变换过程中不同阶段中间表示结构的领域特定和领域无关中间方言。图3概述了编译流水线的详细整体流程及作用于中间表示的主要变换。该流程以提供给 Devito、PSyclone 或 Flang 编译器的 Python 或 Fortran 代码为输入，并借助现有工作 [4, 9]，在 xDSL 框架内将其变换为模板方言。

该流程涉及对中间表示执行一系列变换，并在每一步中对其进行修改和完善。我们发现，面向 WSE 这类机器的编译器流程对 MLIR 的一个基本假设提出了挑战。诸如 [4], 面向传统架构的工作能够利用 MLIR 丰富的、基于图形处理器和中央处理器的基础设施；在流水线早期就完全消除问题本身（此处即模板）的表示，效果也非常显著。相比之下，我们发现，对于 WSE 这类数据流、无缓存架构，在流水线的大部分阶段保留有关内核性质的高级信息（此处即模板）至关重要，这样才能利用这些信息，尽可能有效地生成最适合 WSE 的表示。

本节介绍一种策略：通过应用我们的一组变换，逐步降低模板中间表示；这些变换利用了第4节所述的领域专用和领域通用中间方言。清单2 概述了模板多级中间表示，该表示由清单1 中的 Fortran 代码经 [9]生成。在编译流水线的初始阶段，模板方言使用值语义对计算进行数学描述：变量表示彼此独立的数据副本，不涉及副作用，也不考虑如何分配到内存等操作层面的因素。我们工作的一个重要贡献在于，展示了编译器如何将这种模板表示转换为针对 Cerebras WSE 优化的形式。本节将以清单2 作为贯穿始终的示例，说明这些变换。

代码清单2 本节中用作贯穿示例的模板代码。

```
%result = "stencil.apply"(%input) ({ ^0(%data : !stencil.temp<[-1,255]x[-1,255]x[-1,511]xf32>): %c0 = arith.constant 0.12345 : f32 %d0 = stencil.access%data[1,0,0] : f32 %d1 = stencil.access%data[0,0,0] : f32 %t0 = 算术浮点加法 %d0, %d1 : f32 %r0 = 算术浮点乘法 %c0, %t0 : f32 "stencil.return"(%r0) : (f32) -> () }) : ... -> !stencil.temp<[0,254]x[0,254]x[0,510]xf32>
```

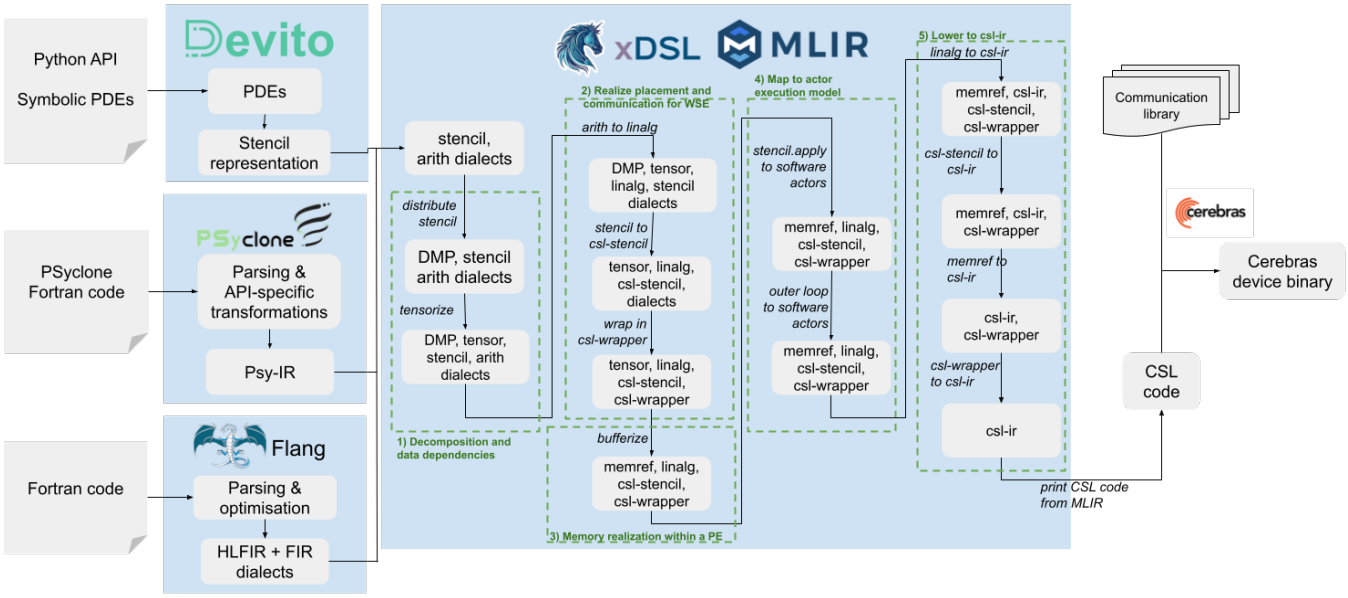

图3. 我们的分阶段编译流程扩展了现有基础设施，通过多个阶段面向新型硬件，从而为 Cerebras WSE 高效生成代码。

如图中绿色虚线所示3，有五组主要的变换作用于中间表示，最终生成*csl-ir*，后者可编译为适用于 WSE 的代码。现在我们将逐一探讨这些变换组。

5.1 组 1：分解与数据依赖关系

我们方法中的第一组变换将模板的三维表示分解并映射到 WSE 的二维处理单元网格上，同时确定每个处理单元进行模板计算时的数据依赖关系。总体策略是将大小为 $x \times y \times z$ 的三维模板转换为大小为 $x \times y$ 的二维模板，使每个处理单元持有一列 z 个值。事实证明，这种分解方式在 WSE 上效果良好 [20]，；尽管理论上可以将多列 z 个值映射到每个处理单元，但鉴于处理单元数量众多且本地内存容量较小（48千字节），因此采用了这种方法。

中间表示层级的分解由现有的 pass 分布式模板 驱动。我们发现，尽管它最初旨在通过 MPI 将模板映射到高性能计算集群[4], 但相同的抽象逻辑也适用于跨 WSE 的 x 和 y 尺寸网格进行分解。最终，这会在我们的中间表示中加入 DMP 方言，并在每个dmp.**swap**操作之前插入到中间表示中模板。**apply**操作，如清单 3 所示。此操作指出执行特定的 apply 之前必须进行的必要数据交换，因此描述了远程数据依赖关系。此转换的结果是将数学表示分解到 WSE 上，高级通信操作标明了在执行模板操作之前必须在处理单元之间交换数据的时机。

清单 3 在将 x, y 维度分配到处理单元并对 z 维度进行张量化。

```
%1 = "dmp.swap"(%input) {strategy = #dmp.网格_切片_2d<
  #dmp.拓扑<254x254>, false>, 交换 = [...]}
  (!模板.temp<[-1,2]x[-1,2]xtensor<512xf32>>) ->
  !模板.temp<[-1  ,2]x[-1  ,2]x张量<512xf32>>
%result = "stencil.apply"(%1) ({
  ^0(%d : !stencil.temp<[-1,1]x[-1,1]x张量<512xf32>>):
    %c0 = arith.constant <dense 0.123450> : 张量<510xf32>
    %d0 = stencil.access%data[1,0] : 张量<510xf32>
    %d1 = 模板。访问%data[0,0] : 张量<510xf32>
    %t0 =算术。浮点加法 %d0  , %d1 : 张量<510xf32>
    %r0 =算术。浮点乘法 %c0  , %t0 : 张量<510xf32>
    "模板.return"(%r0) : (张量<510xf32>) ->()
}) : ...-> ! 模板。临时<[0,1]x[0,1]x张量<512xf32>>
```

这组中的第二个变换是沿张量化维度进行 z，将由 FP32 标量组成的三维网格转换为由 FP32 张量组成的二维网格。MLIR 张量方言提供了一种表示抽象数据集合的方式。此变换保留了数据表示的值语义，同时也提供了秩多态性，因为arith方言中的操作作用于值张量，而非标量。此重写如清单 3所示：将模板转换为在张量上运行的二维模板后，每个模板元素（即一列张量）都会映射到 WSE 上的一个独立处理单元。

5.2 第2组：实现 WSE 的布局和通信

这组变换将第一组变换产生的问题分解后数学表示映射到 WSE 的物理表示，并将数据依赖转换为处理单元之间的通信。如第 2节所述，由于缓冲区大小有限，通信必须以数据块为单位进行，这也为重叠通信与计算、提升性能提供了机会。因此，模板到 *csl-stencil* 变换将模板方言转换为 `csl-stencil` 方言，其中所有dmp。**swap** 操作均替换为csl-stencil。**prefetch**。这些csl-stencil。**prefetch** 操作会添加到各自对应的csl-stencil。**apply** 操作中；后者本身同时结合了计算与通信。

模板 的计算区域**apply** 被拆分为两个区域：第一个区域描述基于远程保存的数据块进行的计算，第二个区域则处理本地保存的数据。列表 4对此进行了说明，其中通信涉及两个数据块，第一 个区域产生的 累加器 随后由第二个区域使用。第一个远程数据处理区域有两个用途：首先是数据打包；在列表 4 中，具体表现为将接收到的数据块插入结果张量；其次是在数据被打包之前，对接收到的数据执行可选的计算。

列表 4 转换后 from the 模板 到 csl-stencil 方言。

```
%result = "csl_stencil.apply"(%input) <{swaps =
  [#csl_stencil.交换<到 [1,  0]>], num_数据块 = 2: i64}>({
  ^0(%d : 张量<4x255xf32>,  %offset : 索引,  %acc : 张量<510xf32>):
    %0 =CSL_模板。访问 %d[1  , 0]: 张量<255xf32>
    ...
    %1 =张量。插入_切片 %0 到 %acc[%offset][255][1]
    CSL_模板。生成%1 : 张量<510xf32>
}, {
  ^1(%d :!模板.temp<[-1,1]x[-1,1]xtensor<510xf32>>,
     %acc : 张量<510xf32>):
    %c0 ≡ arith.constant <dense 1.234500e-01> : 张量<510xf32>
    %d0 ≡ csl_模板.access %d[0  ,0]: 张量<510xf32>
    %t0 = 算术运算浮点加法 %acc , %d0 : 张量<510xf32>
    %r0 = arith.mulf %t0, %c0 : 张量<510xf32>
    csl_模板.yield%r0 : 张量<510xf32>
}) : ...
```

此组中的另一项转换 将 *csl-stencil* 封装在 *csl-wrapper* 中会生成布局元程序，将内核映射到 WSE 上的处理单元，并提取 csl-stencil 操作中的程序级常量参数。尽管 `csl-wrapper` 是与计算域无关的方言，但此转换仍会用来自 `csl-stencil` 方言的、与计算域高度相关的信息填充它；该方言由 csl-wrapper 封装并保留。

5.3 第3组：处理单元内的内存实现

到目前为止，模板上的计算操作均以值语义表示，关注的是值本身，而内存中的存放位置等操作细节则被抽象掉了。这组转换会将中间表示转换为引用语义，具体做法是将张量转换为内存引用，从而将张量映射到物理内存分配。这样做的一个主要原因是，CSL 数学运算遵循目的地传递风格（DPS），即在物理内存上进行操作，从作为操作数传入的物理内存中读取输入，并将结果写入其中。相较之下，张量是不可变的，涉及张量的算术运算总是会产生一个新的张量。

一个转换过程会将我们提出的方言重写为引用语义，并执行部分缓冲区化，之后再使用 MLIR 的缓冲区化将张量转换为内存引用。如果某项计算需要额外的中间缓冲区，那么 MLIR 的缓冲区化过程会充当故障保护机制，自动按需分配这些缓冲区。事实上，这体现了更广泛地使用 MLIR 的一项优势：MLIR 生态系统已经为物化等常见操作提供了丰富的功能，因此可以直接复用，而无需在完全定制的编译器栈中重新实现。

清单 5 第三组方言的变换之后
这些变换在处理单元内实现了内存。

```
%result = "csl_stencil.apply"(%input) ... ({ ... }, {
  ^1(%d : !模板.temp<[-1,1]x[-1,1]x内存引用<510xf32>>,
      %acc : 内存引用<510xf32>):
    %c0 = 算术。常量 <稠密 1.234500e-01> : 内存引用<510xf32>
    %d0 = CSL_模板。访问%d[0,0] : 内存引用<510xf32>
    linalg。加 输入(%acc,  %d0) 输出(%acc)
    linalg。乘 输入(%acc,  %c0)  输出(%acc)
    CSL_模板。产生 %acc : 内存引用<510xf32>
}) : ...
```

该 `arith` 方言不支持 CSL 的 DPS 形式操作，而线性代数 `linalg` 方言支持。因此，我们将`arith` 方言中的计算操作转换为 *linalg* 方言中的对应操作。清单 5 展示了这些变换完成后的中间表示，可以看到 内存引用取代了 张量，并将它们作为操作数提供给 `linalg` 操作。此外，为了充分利用 WSE 上有限的内存，清单 *acc*内存引用 在清单 5 中被重复用作输入，以保存中间结果和最终计算结果。

5.4 Group4：映射到 actor 执行模型

图 1 说明，在 CSL 程序中进行通信，必须将代码组织为任务，并在收到相应消息时调用这些任务。第 2.1 节强调了，将 WSE 视为

遵循参与者模型是有效的：处理单元是硬件参与者，任务则是软件参与者。这组转换通过拆分 `csl-stencil.apply` 操作，将其分解为各项活动并映射到参与者，从而降低到 WSE 的异步执行模型。由于 `csl-stencil.apply` 操作在 WSE 上执行异步通信，其远程数据和本地数据子区域会被降低为软件参与者：每当收到远程数据块时，前者就会被激活；而后者则在收到所有邻居发送的数据块后，于末尾激活一次。这与图 *stencil.communicate* 函数非常相似，该函数由 *seq_kernel0* 和*seq_kernel1* 调用，见图 1。

通过这种向参与者的降低而产生的第二个参与者，通常会处理本地保存的数据，并作为该操作的最后一部分恰好执行一次。与图 1类似，控制流会从这个第二个任务继续执行，后续函数由该软件参与者调用。

如第2节所述，为了使天然同步的模板能够以异步方式运行，不能使用顶层时间步循环，也不能在没有循环的情况下连续执行多个 `csl-stencil.apply` 操作。这是因为`csl-stencil.apply` 操作会触发异步通信，之后程序流便无法以同步方式继续；而第6节探讨的基准测试都要求这两种配置。该组的第二项转换将包围 `csl-stencil.apply` 操作的顶层控制流转换为控制流任务图，其中包含由可调用的零参数函数以及我们 `csl-ir` 方言中的本地任务构成的软件参与者。例如，顶层时间步 `scf.for` 循环会降低为由 CSL 函数组成的控制流图，而非基本块。

5.5 第5组：降低到 csl-ir 方言 ct

最后一组变换将降低为 `csl-ir`方言，并由此生成 CSL 代码。第五组中的第一个变换 *linalg* 到 *csl-ir* 会将 `linalg` 方言中的计算操作降低为 `csl-ir` 方言中的对应操作。这些操作针对数据数组执行，最简单直接的方法是生成一个循环，为每个元素执行算术运算。然而，在 WSE 上的最佳实践是改用 CSL 的数据结构描述符（DSD），它们本质上是通过原生硬件支持在缓冲区上运行的仿射迭代器。CSL 提供了一组高吞吐量的 DSD 算术内置函数，与不基于 DSD 内置函数的计算相比，这些函数能显著提升性能。例如，通过 DSD，列表 5 中所示的乘法和加法操作会降低为`@fmacs(dest, src1, src2, f32_val)`，从而实现 FP32 融合乘加。

此时，模板和计算组件已转换为csl-ir方言。然而，内存仍由memref方言表示。尽管memref类型贯穿`csl-ir` 方言使用，但对`memref`的操作（例如分配和释放）仍需降级为`csl-ir` 方言。此外，数据结构描述符会初始化为缓冲区上的视图和子视图。因此，将 *memref* 降级为 *dsd*过程会基于 memref 缓冲区分配生成 DSD 定义，这些定义在内置计算函数和通信中均会用到。

此组的最后一步将顶层csl-wrapper**模块**操作降级为*csl-ir*。此转换会生成两个输出模块：第一个包含根据布局元程序的抽象表示生成的*csl-ir*操作，第二个则包含此前编译器遍生成的*csl-ir* 操作。

5.6 运行时通信库

硬件参与者（处理单元）之间的数据交换由我们设计的基于 CSL 的库处理。该库采用Jacquelin 等人[20] 提出的可分区通信策略，适用于最多三维、模板尺寸可变的星形模板。该库封装了用于发送和接收数据的样板代码，并将数据拆分为尺寸可配置的数据块。起初，它会在四个方向上调度异步发送和接收数据交换，并在每个方向使用多个内部任务来处理异步通信步骤的完成和路由模式的更新，进而触发用户提供的回调函数。CSL 的分阶段编译机制提供了足够的灵活性和能力，能够编译任意星形模板模式，使我们可以专注于每个处理单元内部软件执行单元层级的代码生成。若要支持不同的模板形状（例如盒形通信模式），则需要更新库。作为补充，Sai 等人 [40] 提出了基于软件的路由模式；而硬件路由配置和交换更新的代码生成仍有待未来研究，即使是在受限的模板模式情形下也不例外。

5.7 优化过程

尽管本节的大部分内容都聚焦于变换数学模板表示，使其成为适合在 WSE 上执行的形式，但我们还引入了若干额外变换，专门用于在模板和算术层面进行优化。其中第一项，模板内联，在编译流水线开始时执行，尝试将连续的模板。**应用**合并为一个融合内核。这是一个现有的编译器遍，最初在 [4] 中开发，旨在消除连续模板之间不必要的主机与设备上下文切换开销。该变换会确保第一个模板的所有输出都传递给第二个模板，即使某些值并未改变；随后，两个模板执行的操作会合并到一个模板。**应用**中。能够利用这一现有变换，正是

与 MLIR 等更广泛的编译器生态系统集成，具有一项关键优势：无需重复开发通用编译器遍。在 UVKBE 基准测试中（我们将在第 6 节探讨），该编译器遍能够将所有模板。**应用**合并为单个操作。随后，在流水线的后续阶段，融合内核会根据缓冲区通信进行拆分，以便通信和计算能够交错执行。

`Varith`是一个专用于 xDSL 的方言，可将多个算术加法或乘法表示为单个变参算术操作。我们在编译流水线的早期使用此方言，在可能的情况下表示算术操作，因为它能显著降低操作相关的处理复杂度，例如，将计算划分为多个区域，分别处理本地和远程持有的数据时。此外，`varith` 方言是 `varith-fuse-repeated-operands` 转换所必需的。我们利用该转换将对同一参数连续进行的一系列加法转换为乘法。对于第 6 节探讨的 Acoustic 内核等代码而言，这项优化尤为重要，因为它可以消除多个算术操作；在该内核中，我们成功将三个 DSD 加法操作替换为一次乘法。

此外，`varith-fuse-repeated-operands`转换还使我们能够将乘法和加法融合为一个*fmac* 操作。将此转换与用于建模 DSD 操作的`linalg` 方言结合使用后，我们的 `linalg-fuse-multiply-add` 转换便能够识别可合并的乘法和加法操作。最终，应用此转换后，对于适当的算术操作链，便会生成`@fmacs` CSL 内置函数；该函数已在第 5.3节中提及。由于模板中乘法后接加法的情况十分常见，我们发现这项优化能将大量乘法和加法操作转换为 `@fmacs`，从而提升性能。

一种高效的底层优化策略，是检查整个模板形状是否都应用了同一个归约函数。在这种情况下，我们可以利用张量广播技术对累加器缓冲区进行扩展，为其增加与通信缓冲区相匹配的虚拟维度，从而通过一次内置函数调用对整个通信缓冲区执行归约，生成高度优化的代码。反之，如果模板形状中的各点需要使用不同的函数处理，或者有一部分点被完全忽略，编译器就需要获取指向通信缓冲区的单独指针，并生成独立的内置函数调用来处理通信数据，这样性能较低。另一种底层优化是在可能的情况下，将系数应用融入通信代码中，以便交错通信与计算。实际上，我们可以不使用类似 `@mov` 的内置函数将输入队列中的数据移至 PE 内存，而是将一组系数与输入队列中的数据相乘后再写入内存。这两种底层技术可以结合使用。

6 评估

本节将比较我们的方法在 Cerebras CS-2 和 CS-3 系统上的性能，并与中央处理器和图形处理器基准进行对比。对于每项基准测试，我们考虑了三种不同的问题尺寸，根据 X 和 Y 维度上的元素数量划分为：小型 – 100×100、中型 –500×500 和大型 – 750×994。大型尺寸的选择旨在充分占用 WSE2 上的处理单元网格。以下五项基准测试的 Z 维度各不相同：

雅可比矩阵（***Flang***）从求解三维拉普拉斯方程以模拟扩散的 Fortran 代码中提取而来。这是一个六点模板，在 Z 维度有 900 个元素的情况下运行 100,000 次迭代。**扩散**（***Devito***）用于求解热扩散方程。它以 Devito 的基于 Python 的领域特定语言编写，是一个三维 13 点模板，在 Z 维度有 704 个元素的情况下运行 512 次迭代。**声学**（***Devito***）在 Devito 中对各向同性声波方程建模，采用 2nd 阶时间近似。与“扩散”类似，这是一个三维 13 点模板，在 Z 维度有 604 个元素的情况下运行 512 次迭代。**25 点地震**（***Cerebras***）是一个用 CSL 编写的地震内核，采用 1st 阶时间近似。该内核由Jacquelin 等人的工作转译而来，可直接与手动调优的 CSL 进行比较。这是一个三维 25 点模板，在 Z 维度有 450 个元素的情况下运行 100,000 次迭代。**UVKBE**（***PSyclone***）包含四个场，其中两个需要跨处理单元通信，此外还有两个连续的模板。**应用**操作。此基准测试通过 PSyclone 领域特定语言以 Fortran 编写，在 Z 维度有 600 个元素的情况下运行一次迭代。

对于所有基准测试，我们都使用 32 位单精度浮点数。报告的结果是三次运行所得的平均吞吐量，以十亿点/秒（即十亿单元/秒）为单位。需要特别指出的是，图 4、5 和 6包含误差线（95% 置信区间），但由于各次运行之间没有显著的离群值，误差线非常小。我们使用 xDSL v0.35 版本，并以 Cerebras SDK 1.3.0 和 Cerebras 机器学习软件 2.4 为目标。图 4展示了采用我们的编译方法，在 WSE2 和 WSE3 上，使用来自 Flang（雅可比矩阵）、Devito（扩散）、Cerebras（地震）和 PSyclone（UVKBE）的较大问题尺寸，对其中四项基准测试进行的性能比较。各项基准测试的迭代次数分别为 100000、512、100000 和 1 次。
WSE2 与 WSE3 之间性能差异的主要原因是 WSE3 升级后的交换逻辑。

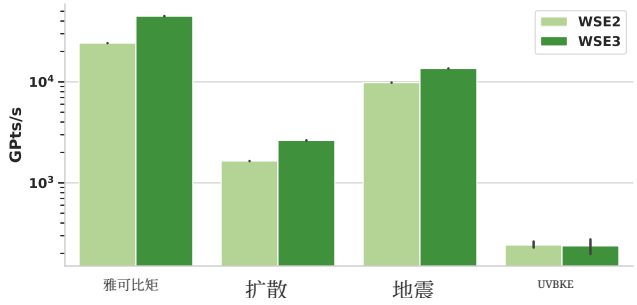

图4. 采用我们的方法，在大尺寸问题规模下运行各基准测试时，WSE2 和 WSE3 的性能对比。我们的代码生成器充分利用最新的硬件特性，以实现最佳性能。

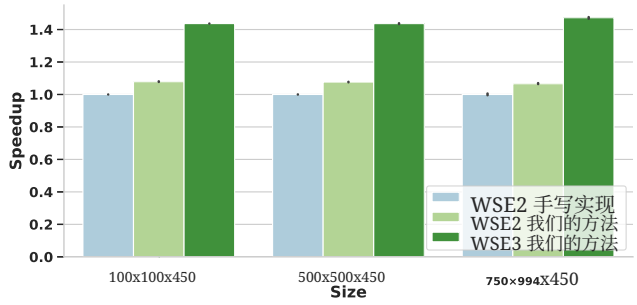

图5. 我们基于编译器的方法与 Cerebras 开发的手写等效实现，在地震基准测试中的对比 [19]。高效的通信和任务利用率赋予了生成代码卓越的性能。

WSE2 上的硬件限制要求在交换配置中，每个处理单元不仅要向相邻单元传输数据，还要向自身传输数据（分别通过东、西、北、南路由）[20]。在 WSE3 上则不再需要这样做，而我们的 WSE3 通信库利用了这一架构改进。

6.1 我们的方法与手写 WSE 代码的对比

在 Cerebras 工程师的合作下，Jacquelin 等人 [20]开发了一个手工编写并优化的地震建模基准测试，它基于25点模板。该基准测试可用于 WSE2 的 Cerebras SDK，但不支持 WSE3 [19]。图 5 比较了针对不同问题尺寸，该手工编写的内核在 WSE2 上的性能，以及我们的方法为 WSE2 和 WSE3 生成的等效实现的性能。[20] 报告指出，手工编写的实现达到了 WSE2 理论峰值性能的 28.2%，而我们的编译流水线生成的版本性能最高提升了 7.9%。性能受多种因素影响。我们生成的版本内存效率更高，因此可以通过单个数据块进行通信

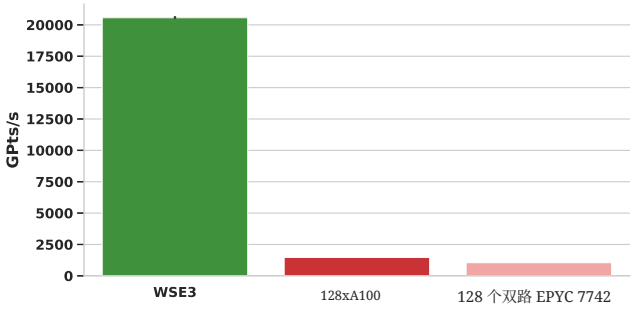

图6. Devito 声学基准测试在 WSE3 上的性能（大尺寸问题规模，100,000 次迭代），与 Tursa 上 128 个 A100 图形处理器运行的 MPI + OpenACC 内核，以及 ARCHER2 Cray-EX 上 128 个 CPU 节点运行的 MPI + OpenMP 内核进行对比，数据来自 [5]。单个晶圆即可达到大型图形处理器或中央处理器集群的吞吐量，展现了 WSE3 的高吞吐能力。

而手工编写的版本使用两个数据块。不过，Jacquelin 等人的数据表明，这可能无法解释所有性能差异。我们的编译方法只传输计算所需的数据。对于此基准测试，这意味着不传输第一列和最后一列的值，而手工编写的代码会传输整列数据，包括不需要的值。我们的通信库还可以更高效地管理任务，将所需任务总数减少约 50 %，从而降低任务激活开销。

由于不同硬件架构代际的差异，需要重写内核，因此手写代码仅适用于 WSE2。相比之下，我们的方法能够无缝支持 WSE2 和 WSE3。总体而言，为 WSE3 生成的代码比 WSE2 实现的性能最高提升 38.1%，这主要归功于前文所述的架构改进。如第 7, [40] 节所述，采用 Python 编写的模板领域专用语言为 WSE 提供了一套完全定制的编译流水线。其编译器未公开提供，但在 [40] 中，他们将性能与图 5所示的同一手动实现的 25 点模板内核进行了比较。他们报告称，其性能略逊于手动实现的代码，而 [40] 中的论点则侧重于程序员生产力。相比之下，正如本节所展示的，对于这一相同内核，我们的方法兼具程序员生产力和性能优势。

6.2 WSE3 与图形处理器和中央处理器的性能对比

为了将我们在 WSE3 上的方法与传统高性能计算架构、图形处理器和中央处理器进行比较，我们评估了 Devito 的 声学 基准测试性能，并与 [5] 报告的结果进行了比较。其研究评估了在 Tursa

超级计算机上运行的 128 个 Nvidia A100 GPU 上的 MPIOpenACC 内核，以及在基于中央处理器的 Cray-EX ARCHER2 超级计算机的 128 个 CPU 节点上运行的 MPI + OpenMP 内核。每个 Tursa 节点包含两颗 24 核 AMD 7413 EPYC 处理器和四个通过 NVLink 连接的 NVIDIA Ampere A100-80 GPU，节点之间由四条 200 Gbps InfiniBand 链路连接。每个 ARCHER2 节点包含两颗 64 核 AMD Zen2（Rome）EPYC 7742 处理器，节点之间通过 HPE 专有的 Slingshot 互连网络连接。实验分别在 Tursa 上使用 Nvidia nvc++ 编译器 23.5-0 版，在 ARCHER2 上使用 Cray Clang 11.0.4 版。所有实验均采用单精度浮点运算。

Bisbas 等人使用了大得多的网格尺寸，1024^3 并且 1158^3 分别在中央处理器和图形处理器上使用了更多的点数，因此通过降低通信开销略占优势。图6表明，对于特定问题尺寸，WSE3 的吞吐量显著更高，在与 128 个 A100 GPU 和 128 个 CPU 节点（16384 个 CPU核心）竞逐更快求解速度时拔得头筹。WSE3 的求解速度约为 128 个 A100 GPU 的 14 倍，比 128 个 CPU 节点快 20 倍。

我们承认，由于问题尺寸不同，这并非严格意义上的直接比较。在 [5], 进行的强扩展研究中，将问题分解到多个图形处理器上，会使每个设备计算更小的子域，且通信计算比更高，从而无法充分发挥单GPU性能潜力。相比之下，WSE3 在单个晶圆上处理整个计算域。虽然弱扩展比较——即每个图形处理器求解更大的问题——可能会缩小 WSE3 的加速幅度，但其优势仍将十分显著。此外，图形处理器基准测试采用的是 OpenACC，而非经过手动调优的 CUDA（后者仅在 Devito 的专有版本中提供），因此未能充分发挥其性能。尽管如此，这项实验再次证明，在竞逐"针对特定问题尺寸的求解时间"时，WSE3 的表现优于由传统硬件构成的大型集群 [33, 41, 44]。

确实，若假设中央处理器上的扩展效率达到完美水平，那么要在此基准测试中匹配单个 WSE3 的性能，大约需要动用 ARCHER2 超级计算机 50% 的资源。此外，尽管这些性能数据令人印象深刻，但也必须指出，此处的 Devito 代码在 WSE3 上运行时并未作任何修改，

图形处理器和中央处理器。

6.3 WSE3 屋顶线性能

屋顶线图以浮点运算次数/秒为纵轴、算术强度（浮点运算次数/字节）为横轴，直观呈现内核的性能，并以系统的峰值计算能力和内存带宽上限为边界。虚线表示机器的理论性能上限。应用若落在这条曲线的起始对角线部分下方，则受内存性能限制；若落在水平部分下方，则受计算能力限制。基准测试的性能越高，就越接近理论上限。事实上，Ltaief 等人正是用屋顶线图

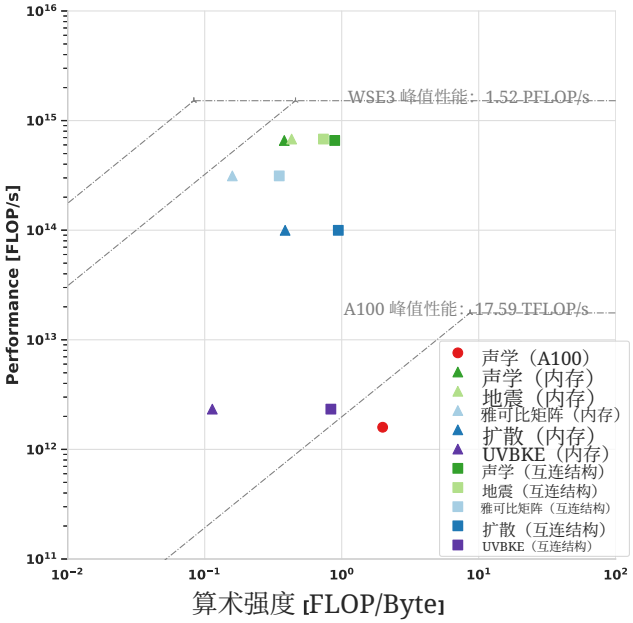

图 7. WSE3 上五项基准测试以及单个 A100 上声学基准测试的屋顶线图。

来论证 WSE 在内存方面的优势：他们观察到，在其他架构上受内存限制、位于对角线部分下方的代码，迁移到 WSE 后则位于水平部分，受计算能力限制。 [28]

图 7展示了 WSE3 的屋顶线图，其中包含我们的五个基准测试，以及在单个 A100 图形处理器上运行的声学基准测试。WSE3 的本地内存带宽高于通信互连网络，因此对于每个 WSE 基准测试，我们都绘制了两个值：第一个假设所有数据访问都来自本地内存，第二个假设所有数据访问都经由互连结构。可以看出，对于所有假设数据驻留在内存中的 WSE3 基准测试，性能都受计算能力限制；除了雅可比矩阵之外，假设所有数据访问都经由互连结构时，其余基准测试的性能也受计算能力限制。相比之下，我们还绘制了来自 [5] 的 A100 图形处理器上的声学基准测试，可以看出该代码在 A100 上受内存带宽限制。实际上，本文所述的方法使用户能够将内核迁移到性能出色得多的架构上，这主要是因为代码不再受内存带宽限制，而且无需进行任何应用层面的更改。Jacquelin 等人 [20] 观察到，25 点地震模板的性能受计算能力限制；我们的实验表明，这一发现同样适用于更多种类的模板程序。

6.4 开发者生产力

如第2节所述，为 WSE 编写代码十分困难，而且会显著增加复杂性，尤其是因为

程序员必须重新思考其执行模型。因此，从程序员生产力的角度来看，为该架构开发和优化代码可能需要大量时间 [8]。不过，尽管代码行数（LoC）指标无法提供完整的情况，它仍能在一定程度上反映所需的工作量。

基准测试	CSL 内核 only（代码行数）	CSL 整个（代码行数）	领域特定语言与我们的方法（代码行数）
25 点地震	196	980	81
声学	211	995	81
扩散	192	976	40
雅可比矩阵	180	964	28
UVKBE	203	987	44

表 1. 生成的 CSL 与采用我们的编译方法使用领域特定语言编程的代码行数（LoC）比较。

表 1 报告了各基准测试中，生成的 CSL 与使用领域特定语言编写代码并采用我们的编译方法所需代码行数的比较。对于 CSL 代码，我们提供了两个数字： 仅 *CSL* 内核仅包含内核本身，不包括在 WSE 上进行放置、通信或与主机交互所需的任何支持代码。 完整 *CSL* 则表示完整的 CSL 代码长度，既包括内核，也包括运行时库中所有必需的支持功能；该运行时库已在第 5.6 节中介绍。可以看出，无论基准测试和领域特定语言如何，采用我们的方法都能大幅减少所需编写的代码量。此外，需要指出的是，科学程序员可能会更容易理解领域特定语言的源代码；而 CSL 不仅高度依赖特定架构，而且往往难以将科学模板代码与其周围的 WSE 特定实现细节区分开来。

7 相关工作

Rocki 等人 [36] 探索了 WSE 首次迭代产品 CS-1 上的模板代码。作者将迭代式 BiCGSTAB Krylov 子空间方法的主要组成部分——稀疏矩阵-向量乘法、AXPY 和内积——移植到了 WSE 上。这项工作完成于 Cerebras 发布 CSL 之前，因此使用专有编程接口开展，达到 0.86 PFLOPS，性能超过 NETL Joule 超级计算机的 200 倍。认识到 WSE 的可编程性挑战后，Woo 等人 [47] 在这项早期工作的基础上，开发了用于编程 WSE 的基于 Python 的 API。重要的是，他们的 API 在保持 CS-2 性能的同时，使 WSE2 的运行时间相较于 NETL Joule 2.0 超级计算机缩短了两个数量级以上。我们的工作解决了两个主要局限 Woo 等人的方法 [47],首先要求程序员将代码改写为使用特定 API，而我们的工作则着重于让领域特定语言开发者与模板方言集成；之后无需修改任何用户代码，这些代码便可在 WSE 上运行。第二个主要区别在于，Woo 等人 [47] 生成的是汇编级代码，该代码本身受保密协议约束，而不是 CSL。相反，通过以 CSL 为目标，我们的方法可在不同代际之间移植，例如从 WSE2 移植到 WSE3；此外，由于它仅依赖公开 API，我们也可以发布编译器源代码。

有人提出了一种基于 Python 的 WSE 模板领域专用语言Sai 等人 [40]，它与我们的方法类似，会生成 CSL 代码。他们主要关注以一种与我们工作互补的方式，为不同的模板形状生成基于软件的路由算法，而计算的生成或整体程序结构的生成则未作讨论。与 Woo 等人 [47], 类似，他们开发了完全定制的编译器，而我们的方法则在 MLIR 生态系统中提供了一系列构建模块。此外，与以往工作（见 [40, 47]）不同，我们将前端编程技术与面向 WSE 的编译流程解耦，因此应用程序代码可以保持不变，无需先移植到新的领域专用语言。

SPADA [14] 是一种面向 WSE（尤其是 CS-2）的空间数据流语言，Gianinazzi 等人为其开发了定制编译器，并在其上集成了 GT4Py 模板 DSL。SPADA 与我们的方法互为补充：SPADA 通过提供 WSE 专用抽象和优化，对软件栈的较低层进行了大量优化。相较之下，我们利用 MLIR，逐步经过多个抽象层向 CSL 降低，在适用时利用各层及现有的编译器构建模块，为 WSE 架构的 CS-2 和 CS-3 版本生成最合适的目标代码。从根本上说，SPADA 旨在解决与我们不同的问题；我们优先考虑在各前端之间共享基础设施，以便于集成。因此，[14]GT4Py 模板 DSL 与 SPADA 之间的语义鸿沟更大，这意味着各前端需要付出更多工作才能将模板接入 SPADA，也有重复开发的风险。相比之下，采用我们的方法后，只要生成模板方言，我们的编译流水线便会以不依赖前端的方式处理从该方言到 WSE 的全部转换。事实上，未来若 SPADA 开发出 MLIR 方言，将我们的工作与 SPADA 集成会很有意思；这样一来，模板领域专用语言前端便能透明地利用Gianinazzi 等人开发的 WSE 专用抽象和优化。

尽管实验设置上的差异使性能比较颇为困难，但鉴于 SPADA 与我们的工作相辅相成，仍值得重点指出其中的几个方面。按照第 6.3 节所述的方法，一个实现拉普拉斯

扩散方程的三维雅可比矩阵在 CS-2 上达到 169 TFLOP/s，在 CS-3 上达到 313 TFLOP/s；相比之下，二维拉普拉斯算子在 [14]上于 CS-2 达到 120 TFlop/s。UVKBE 是 [14] 中性能最高的内核，在 CS-2 上达到约 150 TFlop/s。其性能超过了我们工作中的 UVKBE 基准测试，但我们采用的 UVKBE 基准测试来自 [15]，它只执行一次迭代。这与 [14] 中的 UVKBE 不同，后者有更多机会减少开销。另一方面，我们性能最高的内核（25点模板）在 CS-2 上达到 491 TFlop/s，在 CS-3 上达到 678 TFlop/s。无论如何，Gianinazzi 等人还展示了使用基于模板的领域特定语言驱动 WSE 所带来的显著程序员生产力优势；与手动编写 CSL 相比，他们大幅减少了所需的代码行数。

Jacquelin 等人 [20] 针对 CS-2 上的地震建模，开发了经过优化的25点模板实现；Sai 等人 [39]实现了三维9点模板，展示了如何将数据间接传输到对角偏移位置的处理单元，以支持秩为 1 的模板所需的对角数据依赖。此外，Sai 等人 [38] 提出了一个有限体积内核，先执行三维7点模板运算，随后执行全互连结构的全归约运算。我们方法的一个基石，是将这些论文中的技术与最佳实践融入编译器，使非专家程序员也能受益于这些经验，而无需精通 WSE 编程。

科学计算社区长期以来一直在探索抽象层，以提高生产力，涵盖高层领域专用方法以及低层循环/表达式级优化。较高层的符号框架包括基于 Python 的 OpenSBLI [18]，它采用爱因斯坦求和约定进行符号建模，并通过基于 C/C++ 的 OPS库 [35]自动实现并行。ExaStencils [21, 25] 将用于有限差分的分层符号领域专用语言相结合，并通过代码生成支持 CPU/GPU架构。STELLA[16],以及后续的 GridTools [2], 采用数学离散化语法，尤其支持了 COSMO天气模型的 GPU 加速 [43]。

然而，现有的这些模板框架都不支持 WSE 等架构。在本文中，我们展示了 MLIR 的一项主要优势：各种前端语言和领域专用语言都能利用构建模块，将现有的模板 方言降低到 WSE。事实上，上述模板领域专用语言都可以与我们的方法集成。此外，尽管我们的方法是首个面向 Cerebras WSE 的 MLIR流水线，但也有一些针对其他新型架构开发的 MLIR方言和变换，例如 AMD的AIE [3], Tenstorrent Tensix [42], 以及 Wang等人 [46]为粗粒度可重构阵列编译器提供的 MLIR 基础设施。不过，尽管这些工作也为不同架构提供了 MLIR 构建模块，但与我们面向 Cerebras WSE 的方法不同，它们都不支持在目标加速器上加速运行现有语言和领域专用语言中未经修改的代码。

8 结论与后续工作

本文介绍了一个用于模板代码的 MLIR 降低流水线，展示了如何基于现有的高级模板计算域抽象，在无需修改代码的情况下，通过代码生成有效地面向 Cerebras WSE。我们使用来自三种不同编程前端的五个基准测试进行评估，并证明我们的方法具有竞争力，且性能优于由 Cerebras 工程师部分手工调优的模板实现。与一组由 128 个 Nvidia A100 GPU 组成的集群以及由 128 个节点构成、基于中央处理器的 ARCHER2 Cray-EX 超级计算机相比，我们的方法分别快了 14 倍和 20 倍。

此前，要将代码移植到 WSE3，始终需要付出大量努力；不仅需要用 CSL 重写代码，还必须从根本上重构算法，使其适应 Cerebras 系统异步、基于参与者的特性。在这项工作中，我们观察到，采用基于参与者的范式有助于在 WSE 上建立通信与计算的执行模型，从而实现一致性。未来，我们认为社区有必要进一步探索这一点：它不仅可能成为 WSE 的一种编程模型，还可能有助于描述面向其他各类类似 CGRA 的加速器时遇到的一些问题。我们的实现以开源形式发布于 https://github.com/xdslproject/wse-stencil，该实现使用 xDSL v0.35，其中包含本文讨论的许多方言和变换 https://github.com/xdslproject/xdsl。

致谢

本研究得到工程与物理科学研究委员会（EPSRC）资助（项目编号 EP/W007940/1），以及爱丁堡皇家学会个人研究奖学金（编号 3271）的支持。我们感谢爱丁堡大学的爱丁堡国际数据中心（EIDF）提供 CS-3 的使用权限，并感谢莱布尼茨超级计算中心（LRZ）提供 CS-2。作者感谢 Cerebras 提供技术支持，也感谢 P. Kelly、E. Bauer 和 N. Heinimann 提供深思熟虑的早期反馈，最后感谢 shepherd 和各位审稿人的意见，以及他们为改进本研究所提供的支持。

参考文献

[1] Samantha V Adams、Rupert W Ford、M Hambley、JM Hobson、I Kavčič、Christopher M Maynard、Thomas Melvin、Eike Hermann Müller、S Mullerworth、Andrew R Porter 等。2019年。LFRic：应对天气与气候模型中的可扩展性和性能可移植性挑战。《并行与分布式计算杂志》132（2019年），383–396。[2] Anton Afanasyev、Mauro Bianco、Lukas Mosimann、Carlos Osuna、Felix Thaler、Hannes Vogt、Oliver Fuhrer、Joost VandeVondele，以及

Thomas C. Schulthess。2021年。GridTools：面向可移植天气与气候应用的框架。SoftwareX 15（2021年），100707。doi:10.1016 2021 100707 /j.softx. 。AMD。n.d.。IRON API 和基于[3] [ ] MLIR 的 AI Engine 工具链。https: //github.com/Xilinx/mlir-aie[4] George Bisbas、Anton Lydike、Emilien Bauer、Nick Brown、Mathieu Fehr、Lawrence Mitchell、Gabriel Rodriguez-Canal、Maurice Jamieson、Paul H. J. Kelly、Michel Steuwer 和 Tobias Grosser。2024年。面向模板计算领域专用语言中分布式内存并行的共享编译栈。载于第 29 届 ACM 架构支持编程语言与操作系统国际会议论文集，第 3 卷（美国加利福尼亚州拉霍亚）（ASPLOS '24）。计算机协会，纽约州纽约，38–56。doi:10.1145/3620666.3651344[5] George Bisbas、Rhodri Nelson、Mathias Louboutin、Fabio Luporini、Paul H.J. Kelly 和 Gerard Gorman。2025年。面向可扩展有限差分求解器的自动化 MPI-X 代码生成。载于2025年 IEEE 并行与分布式处理国际研讨会（IPDPS）。689–701。doi:10.1109/IPDPS64566.2025.00067[6] Nick Brown。2021年。在赛灵思和英特尔 FPGA 上加速大气建模中的平流。载于2021年 IEEE 国际集群计算会议（CLUSTER）。IEEE，767–774。[7] Nick Brown。2024年。将 Flang Fortran 编译器与标准 MLIR 完全集成。载于 SC24-W：高性能计算、网络、存储与分析国际会议研讨会。IEEE，939–949。[8] Nick Brown、Brandon Echols、Justs Zarins 和 Tobias Grosser。2022年。探索 Cerebras 晶圆级引擎对模板计算代码的适用性。载于欧洲并行处理会议。施普林格，51–65。[9] Nick Brown、Maurice Jamieson、Anton Lydike、Emilien Bauer 和 Tobias Grosser。2023年。通过利用基于 MLIR 的领域专用抽象，实现 Flang 中的 Fortran 性能优化与自动并行化。载于高性能计算、网络、存储与分析国际会议 SC '23 研讨会论文集（美国科罗拉多州丹佛）（SC-W '23）。计算机协会，纽约州纽约，904 913 10 1145 3624062 3624167 –。[10] doi:。/。Denis Caromel、Ludovic Henrio 和 Bernard Paul 2009 Serpette。。异步顺序进程。Information and Computation 207，4（2009年），459–495。[11] Johannes de Fine Licht、Maciej Besta、Simon Meierhans 和 Torsten Hoefler。2020年。面向高性能计算的高级综合代码变换。IEEE 并行与分布式系统汇刊 32，5（2020年），1014–1029。[12] Jonathan Edwards。2009年。相干反应。载于第 24 届 ACM SIGPLAN 面向对象编程系统、语言与应用会议伴随会议论文集（美国佛罗里达州奥兰多）（OOPSLA '09）。计算机协会，纽约州纽约，925–932。doi:10.1145/1639950.1640058[13] Mathieu Fehr、Jeff Niu、River Riddle、Mehdi Amini、Zhendong Su 和 Tobias Grosser。2022年。IRDL：面向 SSA 编译器的中间表示定义语言。载于第 43 届 ACM SIGPLAN 编程语言设计与实现国际会议论文集（美国加利福尼亚州圣迭戈）（PLDI 2022）。计算机协会，纽约州纽约，199–212。doi:10.1145/3519939.3523700[14] Lukas Gianinazzi、Tal Ben-Nun 和 Torsten Hoefler。2025年。SPADA：空间数据流架构编程语言。arXiv 预印本 arXiv:2511.09447（2025年）。[15] Tobias Gysi、Christoph Müller、Oleksandr Zinenko、Stephan Herhut、Eddie Davis、Tobias Wicky、Oliver Fuhrer、Torsten Hoefler 和 Tobias Grosser。2021年。面向 GPU 的领域专用多层中间表示重写：用于 GPU 加速气候模拟的 Open Earth Compiler。ACM 体系结构与代码优化汇刊（TACO）18，4（2021年），1–23。[16] Tobias Gysi、Carlos Osuna、Oliver Fuhrer、Mauro Bianco 和 Thomas C. Schulthess。2015年。STELLA：天气与气候模型中结构化网格方法的领域专用工具。载于《高性能计算、网络、存储与分析国际会议论文集》，SC 2015，美国得克萨斯州奥斯汀，2015年11月15日至20日，Jackie Kern 和 Jeffrey S. Vetter（编）。ACM，美国纽约州纽约，41:1–41:12。doi:10.1145/2807591.2807627[17] Carl Hewitt、Peter Bishop 和 Richard Steiger。1973年。用于人工智能的通用模块化 ACTOR形式体系。载于第三届国际人工智能联合会议论文集（美国斯坦福）（IJCAI'73）。Morgan Kaufmann Publishers Inc.，美国加利福尼亚州旧金山，235–245。[18] Christian T. Jacobs、Satya P. Jammy 和 Neil D. Sandham。2017年。OpenSBLI：用于在多种计算机架构上自动推导并行执行有限差分求解器的框架。《计算科学杂志》18（2017），12–23。doi:10 1016 2016 11 001 。/j.jocs。。Mathias Jacquelin、[19] Mauricio Araya–Polo 和 Jie Meng。 [n. d.]。Csl-Examples/Benchmarks/25-Pt-Stencil at Master · Cerebras/Csl-Examples。https://github.com/Cerebras/csl-examples/tree/master/benchmarks/25-pt-stencil[20] Mathias Jacquelin、Mauricio Araya–Polo 和 Jie Meng。2022年。可扩展分布式高阶模板计算。载于 SC22：高性能计算、网络、存储与分析国际会议。1 13 10 1109 41404 2022 00035 – 。doi：。/SC 。 。[21] 2016 Sebastian Kuckuk 和 Harald Köstler。 。从 ExaSlang 自动生成大规模并行代码。《计算》4，3（2016）。doi: 10 3390 4030027 。/[22] computation Hsiang Tsung Kung 和 Charles E Leiserson。1979年。脉动阵列（用于超大规模集成电路）。载于《1978年稀疏矩阵会议论文集》，第1卷。工业与应用数学学会，美国宾夕法尼亚州费城，256–282。[23] C. Lattner 和 V. Adve。2004年。LLVM：用于终身程序分析与转换的编译框架。载于《代码生成与优化国际研讨会，2004年》。CGO 2004。75–86。doi:10.1109/CGO.2004.1281665[24] Chris Lattner、Mehdi Amini、Uday Bondhugula、Albert Cohen、Andy Davis、Jacques Pienaar、River Riddle、Tatiana Shpeisman、Nicolas Vasi- lache 和 Oleksandr Zinenko。2021年。MLIR：扩展领域专用计算的编译器基础设施。载于2021年 IEEE/ACM 代码生成与优化国际研讨会（CGO）。2–14。doi:10.1109/CGO51591.2021.9370308[25] Christian Lengauer、Sven Apel、Matthias Bolten、Shigeru Chiba、Ul- rich Rüde、Jürgen Teich、Armin Größlinger、Frank Hannig、Harald Köstler、Lisa Claus、Alexander Grebhahn、Stefan Groth、Stefan Kro- nawitter、Sebastian Kuckuk、Hannah Rittich、Christian Schmitt 和 Jonas Schmitt。2020年。ExaStencils：高级多重网格求解器生成。载于《面向百亿亿次级计算的软件》- SPPEXA 2016-2019，Hans- Joachim Bungartz、Severin Reiz、Benjamin Uekermann、Philipp Neu- mann 和 Wolfgang E. Nagel（编）。Springer International Publishing，Cham，405 452 10 1007 978 3 030 47956 5_14 – 。[26] 2023 https://doi.org/ 。/ - - - - Sean Lie。 。Cerebras架构深度解析：首次深入了解面向深度学习的软硬件协同设计。《IEEE Micro》43，3（2023），18–30。[27] M. Louboutin、M. Lange、F. Luporini、N. Kukreja、P. A. Witte、F. J. Herrmann、P. Velesko 和 G. J. Gorman。2019年。Devito（v3.1.0）：用于有限差分和地球物理勘探的嵌入式领域专用语言。《地球科学模型开发》12，3（2019），1165–1187。doi:10.5194/gmd-12-1165-2019[28] Hatem Ltaief、Yuxi Hong、Leighton Wilson、Mathias Jacquelin、Matteo Ravasi 和 David Elliot Keyes。2023年。利用 Cerebras CS-2系统上的代数压缩扩展多维地震处理的“内存墙”。载于《高性能计算、网络、存储与分析国际会议论文集》（美国科罗拉多州丹佛）（SC '23）。计算机协会，

美国纽约州纽约，文章6，12页。doi:10.1145/3581784.3627042[29] Fabio Luporini、Mathias Louboutin、Michael Lange、Navjot Kukreja、Philipp Witte、Jan Hückelheim、Charles Yount、Paul H. J. Kelly、Felix J. Herrmann 和 Gerard J. Gorman。2020年。Devito的架构与性能：一个自动化模板计算系统。《ACM 数学软件汇刊》46，1，文章6（2020年4月），28页。doi:10.1145/3374916[30] Thomas Melvin、Steve Mullerworth、Rupert Ford、Chris Maynard 和 Mike Hobson。2017年。LFRic：构建新的统一模式。载于《欧洲地球科学联盟大会会议摘要》。13021。[31] Aaron Meurer、Christopher P. Smith、Mateusz Paprocki、Ondřej Čertík、Sergey B. Kirpichev、Matthew Rocklin、AMiT Ku mar、Sergiu Ivanov、Jason K. Moore、Sartaj Singh、Thilina Rathnayake、Sean Vig、Brian E. Granger、Richard P. Muller、Francesco Bonazzi、Harsh Gupta、Shivam Vats、Fredrik Johansson、Fabian Pedregosa、Matthew J. Curry、Andy R. Terrel、Štěpán Roučka、Ashutosh Saboo、Isuru Fernando、Sumith Ku- lal、Robert Cimrman 和 Anthony Scopatz。2017年。SymPy：Python中的符号计算。PeerJ 计算机科学3（2017年1月），e103。doi: 10 7717 103 。/ peerj-cs。[32] Marcelo Orenes-Vera、Ilya Sharapov、Robert Schreiber、Mathias Jacquelin、Philippe Vandermersch 和 Sharan Chetlur。2023年。晶圆级快速傅里叶变换。载于第37届国际超级计算大会论文集。180–191。[33] Danny Perez、Aidan Thompson、Stan Moore、Tomas Oppelstrup、Ilya Sharapov、Kylee Santos、Amirali Sharifian、Delyan Z. Kalchev、Robert Schreiber、Scott Pakin、Edgar A. Leon、III Laros、James H.、Michael James 和 Sivasankaran Rajamanickam。2025年。打破范式：克服通用硬件上分子动力学的时间限制。《化学物理学杂志》162，7（2025年2月），072501。doi:10.1063/5.0249193[34] A. R. Porter、J. Appleyard、M. Ashworth、R. W. Ford、J. Holt、H. Liu 和 G. D. Riley。2018年。有限差分或有限元代码的可移植多核与众核性能——应用于 NEMO（NEMOLite2D 1.0）的自由表面组件。《地球科学模型开发》11, 8 2018 3447 3464 10 5194 11 3447 2018 （），–。[35] doi: 。/gmd- - - István Z. Reguly、Gihan R. Mudalige 和 Michael B. Giles。 。利用 OPS 在运行时对大规模模板代码进行循环分块。《IEEE 并行与分布式系统汇刊》29，4（2018），873–886。doi:10.1109/TPDS.2017.2778161[36] Kamil Rocki、Dirk Van Essendelft、Ilya Sharapov、Robert Schreiber、Michael Morrison、Vladimir Kibardin、Andrey Portnoy、Jean Francois Dietiker、Madhava Syamlal 和 Michael J ames。2020年。晶圆级处理器上的快速模板代码计算。载于《高性能计算、网络、存储与分析国际会议论文集》（佐治亚州亚特兰大）（SC '20 58，14）。IEEE 出版社，文章页码。[37] Gabriel Rodriguez-Canal、Nick Brown、Maurice Jamieson、Emilien Bauer、Anton Lydike 和 Tobias Grosser。2023年。Stencil-HMLS：一种在现场可编程门阵列上自动优化模板代码的多层方法。载于《高性能计算、网络、存储与分析国际会议 SC'23 研讨会论文集》。556–565。[38] Ryuichi Sai、François P. Hamon、Joh n Mellor-Crummey 和 Mauricio Araya-Polo。2024年。数据流架构上的无矩阵有限体积核。载于《高性能计算、网络、存储与分析国际会议论文集》（美国佐治亚州亚特兰大）（SC '24）。IEEE 出版社，文章28，11页。doi:10.1109/SC41406.2024.00034[39] Ryuichi Sai、Mathias Jacquelin、Francois Hamon、Mauricio Araya- Polo 和 Randolph R. Settgast。2023年。大规模分布式有限体积通量计算。载于《高性能计算、网络、存储与分析国际会议 SC '23 研讨会论文集》（美国科罗拉多州丹佛）（SC-W ' 23）。计算机协会，美国纽约州纽约，1713–1720。doi:10.1145/3624062.3624252

[40] Ryuichi Sai、John Mellor-Crummey、Jinfan Xu 和 Mauricio Araya- Polo。2024年。面向数据流架构的高阶模板自动代码生成。载于《高性能计算、网络、存储与分析国际会议论文集》（美国佐治亚州亚特兰大）（SC '24）。IEEE 出版社，论文 19，13 页。doi:10.1109/SC41406.2024.00025[41] Kylee Santos、Stan Moore、Tomas Oppelstrup、Amirali Sharifian、Ilya Sharapov、Aidan Thompson、Delyan Z. Kalchev、Danny Perez、Robert Schreiber、Scott Pakin、Edgar A. Leon、James H. Laros、Michael James 和 Sivasankaran Rajamanickam。2024年。利用晶圆级系统突破分子动力学的时间尺度屏障。载于 SC24：高性能计算、网络、存储与分析国际会议。 1 13 10 1109 41406 2024 00014 – 。 doi: 。/SC[42] 。 。Tenstorrent[ ]。无日期。tt-mlir。 https://github.com/tenstorrent/tt-mlir[43] Felix Thaler、Stefan Moosbrugger、Carlos Osuna、Mauro Bianco、Hannes Vogt、Anton Afanasyev、Lukas Mosimann、Oliver Fuhrer、Thomas C. Schulthess 和 Torsten Hoefler。2019年。将 COSMO 天气模型移植到众核CPU。载于《先进科学计算平台会议论文集》（瑞士苏黎世）（PASC '19）。计算机协会，纽约州纽约，美国，论文 13，11 页。doi:10.1145/3324989.3325723[44] John Tramm、Bryce Allen、Kazutomo Yoshii、Andrew Siegel 和 Leighton Wilson。2024年。人工智能加速器硬件上的蒙特卡洛粒子输运高效算法。《计算机物理通讯》298（2024），109072。doi:10.1016/j.cpc.2023.109072[45] Anda Trifan、Defne Gorgun、Michael Salim、Zongyi Li、Alexander Brace、Maxim Zvyagin、Heng Ma、Austin Clyde、David Clark、David J Hardy 等。2022年。智能分辨率：将冷冻电镜与人工智能驱动的多分辨率模拟相结合，观察严重急性呼吸综合征冠状病毒2型复制转录机器的运作。《国际高性能计算应用杂志》36，5-6（2022），603–623。[46] Yuxuan Wang、Cristian Tirelli、Lara Orlandic、Juan Sapriza、Rubén Ro- dríguez Álvarez、Giovanni Ansaloni、Laura Pozzi 和 David Atienza。2025年。一种基于 MLIR 的粗粒度可重构阵列应用部署编译框架。载于《应用可重构计算国际研讨会》。施普林格，33–50。[47] Mino Woo、Terry Jordan、Robert Schreiber、Ilya Sharapov、Shaheer Muhammad、Abhishek Koneru、Michael James 和 Dirk Van Es- sendelft。2022年。场方程建模的颠覆性变革：晶圆级引擎的简易接口。arXiv:2209.13768 [cs.DC] https://arxiv.org/abs/ 2209 13768 。Edgar Zamora-Gómez、Pedro[48] García-López 和 Rubén Mondéjar。2015年。延续复杂性：分布式系统中的回调地狱。载于《欧洲并行处理会议》。施普林格，286–298。[49] Maxim Zvyagin、Alexander Brace、Kyle Hippe、Yuntian Deng、Bin Zhang、Cindy Orozco Bohorquez、Austin Clyde、Bharat Kale、Danilo Perez-Rivera、Heng Ma 等。2023年。GenSLMs：基因组尺度语言模型揭示 SARS-CoV-2 的进化动力学。《国际高性能计算应用杂志》37，6（2023），683–705。