探索 Versal AI 引擎在加速基于模板的大气平流模拟中的应用

Nick Brown
n.brown@epcc.ed.ac.uk 爱丁堡
大学 EPCC，英国爱丁堡

arXiv:2301.13016v1 [cs.DC] 28 Dec 2022

摘要
AMD Xilinx 推出的新型 Versal 自适应计算加速平台（ACAP）是一种 FPGA 架构，将可重构逻辑阵列与其他片上硬化计算资源相结合。AI 引擎是其中一种资源，它们通过高度向量化的方式运行，可提供强大的原始计算能力，因此有望适用于包括高性能计算模拟在内的多种工作负载。然而，这项技术仍处于早期阶段，尚未证明能够加速高性能计算代码，相关基准测试和最佳实践也仍然缺乏。

本文介绍了一份经验报告，探讨如何将 Piacsek 和 Williams（PW）平流方案移植到 Versal ACAP 上，并使用芯片的 AI 引擎加速计算。平流是一种基于模板的算法，广泛应用于大气建模，包括英国气象局最初开发该方案的多个代码。我们以此算法为载体，探讨构建 AI 引擎计算内核的最佳方法，以及如何更好地将 AI 引擎与可编程逻辑连接起来。我们使用 VCK5000，与该平台上不使用 AI 引擎的 FPGA 配置、Alveo U280、24 核 Xeon Platinum Cascade Lake CPU 以及 Nvidia V100 GPU 进行性能评估。结果发现，尽管可编程逻辑阵列与 AI 引擎之间的通道数量构成限制，但借助 ACAP，我们仍能将性能提升至 Alveo U280 的两倍。

关键词
Versal ACAP、AI 引擎、FPGA、基于模板的算法、VCK5000、大气平流、高性能计算

1 引言

Versal 自适应计算加速平台（ACAP）是一种新型 FPGA，将可编程逻辑（PL）与其他组件相结合，包括基于 CPU 的可编程子系统（PS）和 AI 引擎 [4]。这些 AI 引擎，即 AIE，本文中交替使用这两个术语，因其旨在加速高度并行的向量运算而受到特别关注。Versal AI 系列包含多达 400 个引擎，运行频率介于 1 GHz 和 1.2 GHz 之间；每个引擎均采用超长指令字（VLIW）设计，每个周期最多可发出七条指令。AI 引擎每个周期能够执行 8 路向量化单精度浮点运算，以及最多 128 次 8 位定点算术运算。

AIE 提供的大量原始计算能力对于高性能计算（HPC）工作负载颇具吸引力：利用 Versal 的 PL 针对应用定制内存访问，并借助 AI 引擎加速计算，具有很大的潜力。迄今为止，针对 AIE 的初步研究数量非常有限 [5] [10], ，因此一个尚未解决的重要问题是能否有效利用这些引擎处理实际高性能计算内核。在本研究中，我们采用英国气象局 NERC 云模型（MONC）的 atmospheric advection 内核 [3], 作为探索 AI 引擎的载体。MONC 是一个开源高分辨率大气建模框架。基于高性能计算代码中非常常见的基于模板的计算模式，本文将探讨如何最好地将这种计算模式映射到 AIE 上，以及其性能与其他方法相比如何。本文结构如下：第 2 节介绍本研究的背景，第 3 节总结实验设置。第 4 节探讨如何构建 AIE 内核并将其与 PL 连接，随后在第 5 节中与其他硬件进行性能比较。最后，我们在第 6 节总结全文并讨论相关建议。

本文的新颖贡献包括：**1**）探索如何最有效地构建 AIE 内核**2**）对 AIE 与其他硬件进行初始性能比较 **3)**突出介绍当前 AIE 技术的一些局限性，使用该硬件时必须加以考虑。

2 背景

2.1 Versal AI 引擎

Xilinx 新型 AI 引擎采用 VLIW 设计，因此每个周期内，每个引擎最多可发出两条加载指令、一条存储指令、一条标量运算指令、一条定点或浮点向量运算指令以及两条移动指令。向量单元的宽度为 256 位。本文重点讨论单精度浮点运算，因此每个引擎每个周期最多可执行八次单精度浮点计算。因此，确保代码经过正确的向量化，对于在 AIE 上获得最佳性能至关重要。基于 VCK5000 上运行的 400 个 AI 引擎（时钟频率为 1.2 GHz），其理论单精度浮点性能为每秒 3.6 万亿次浮点运算。

AI 引擎排列成二维阵列，两个维度上的引擎都与相邻引擎相连。每个引擎包含 16KB 程序存储器和 32KB 本地数据存储器；对于后者，引擎可直接访问三个相邻引擎的存储器，从而提供总计 128KB 的连续可寻址数据存储器 [8]。此外，每个引擎有两个 32 位输入流和两个 32 位输出流，并通过先进先出队列组合，实现每四个时钟周期 128 位的访问。最后，AI 引擎通过一条级联流与一个相邻引擎相连，该流宽 384 位，旨在支持算术运算的链式处理。

AIE 代码由两部分组成：将映射到 AI 引擎的内核，以及通过其流和内存将内核彼此连接起来并连接到可编程逻辑的图。从编程角度来看，数据进入或离开内核有两种方式，

窗口和流 [9]。窗口提供一个缓冲区，其中会跟踪窗口中的当前数据位置。对于输入窗口，内核会从该缓冲区消耗数据；对于输出窗口，则会写入数据。另一种方法是使用流，它为内核提供无限数量的标量和向量，以供读取和写入。这两种方法之间存在一个重要区别：数据窗口仅会在内核的外部迭代之间前进到下一个窗口，这由 AIE 图驱动；而流可以在内核内部持续读取和写入。因此，使用窗口时，必须频繁启动和停止内核以刷新窗口数据，而使用流则无需如此。

虽然 AI 引擎是本文的主要关注点，但同样重要的是，需要强调 Xilinx 对其 Versal 系列中的 PL 所做的整体架构改进。基于 7 纳米工艺技术构建，与前几代产品相比，包括 DDR 控制器和 PCIe 接口在内的众多组件均已实现硬化 [1]。此外，还提供了专用的片上网络（NoC），它不仅连接 PL 与 AIE，还可用于连接 PL 上的 IP 模块。

2.2 Piacsek 和 Williams 平流内核

平流是由于风而导致数值在大气中移动的过程，约占运行时间的 40%，是 MONC 模型中单个运行时间最长的功能 [3]。代码循环处理三个场：*U*、*V* 和 *W*，分别表示 x、y 和 z 维度上的风速。该 Piacsek 和 Williams（PW） [6] 平流方案在模型的每个时间步都会调用，并计算每个场的平流结果，也称为源项。该平流方案是一种深度为 1 的基于模板的算法，计算某个网格单元的值时，需要汇总来自三个维度上相邻值的贡献。

在先前的工作中， [2] 该内核通过高级综合（HLS）移植到 Alveo U280，并利用 数据流 HLS pragma 使多个组件能够并发运行。该 HLS 内核的结构如图1所示，其中方框表示数据流区域，方框之间的箭头表示内部 HLS 流。三维移位缓冲区提供了一种定制的存储器解决方案，能够每个周期向平流计算阶段提供全部 27 个模板值。事实证明，这种方法是最优方案，尽管平流计算并不需要全部 27 个相邻模板值。在这一既有结构的基础上，我们的假设是，可以将平流计算阶段替换为连接 AI 引擎的输入和输出流，同时继续利用可编程逻辑（PL）上现有的内存访问定制方案；该方案在 [2], 中与 AI 引擎的原始计算能力结合后表现良好。

3 实验设置

在本工作中，我们使用的是一块 Xilinx VCK5000，其中包含一个 Versal VC1902 ACAP 和 16GB DDR4-DRAM。所有 VCK5000 运行均使用 Vitis 2022年.1 构建，PL 运行频率为 300MHz，VC1902 包含 400 个运行频率为 1.2GHz 的 AI 引擎。我们将其与 Alveo U280 进行比较，后者包含 8GB HBM2，同样运行频率为 300MHz，其 Alveo 内核使用 Vitis 2021年.1 构建。VCK5000 和 Alveo U280 都是基于 PCIe 的卡，由一台配备 32 核 AMD EPYC 7502 处理器和 256GB DRAM 的机器承载。

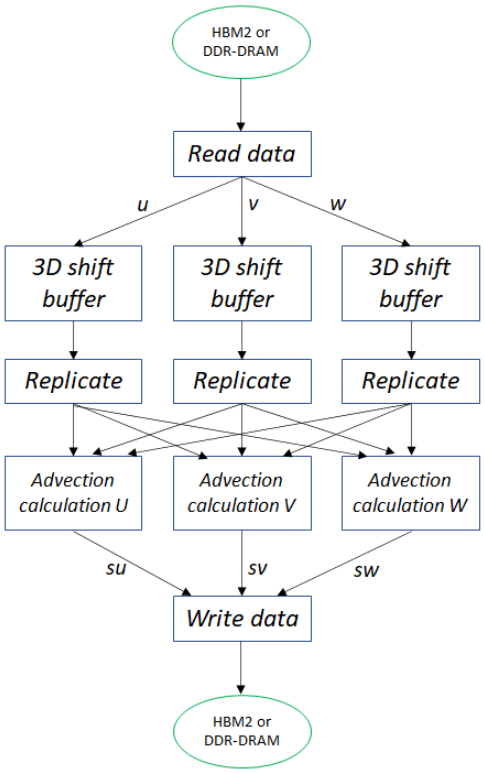

图1： [2]中 HLS 平流内核的数据流设计

所有报告的结果均为五次运行的平均值，性能结果以有效浮点运算次数表示，即对计算结果有贡献的浮点运算次数。我们的性能数据仅测量设备端执行时间，不包括将输入数据复制到主机和设备之间，或将结果数据从设备复制回主机所用的时间。这是因为我们在本工作中最关注 AIE 的性能，因此，在与具有不同主机设备数据传输开销的其他技术进行比较时，设备端性能能够提供更清晰的对比。

4 AIE移植与优化

我们首先将基于平流模板的计算分解为各项基本操作。对于每个网格单元，代码依次执行六次加法、六次乘法、四次减法，最后通过加法归约将这些减法结果求和。工具链不支持大小为六的浮点向量，因此我们额外填入两个空值，将向量扩展至大小为八。这就是我们报告有效浮点运算次数而非总浮点运算次数的原因：有效浮点运算次数不计入对这些空值的处理，只考虑实际有助于得到平流结果的浮点运算。

该内核的结构如图2所示，其中第一次 8 路向量加法需要由十六个浮点数组成的操作数。乘法还需要额外的

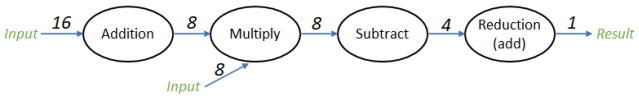

图2：每个网格单元的 AIE 计算示意图，其中数字表示所提供的单精度浮点数的数量。

八个输入数，并将它们与前一次加法的结果相乘。我们将其封装为单个 AIE 内核，清单 1 给出了代码的部分示意。为了准备向量加法，在第 11 至 14 行读取四个数的流，并将其加载到 *左端项* 和*右端项* 向量的相应位置。随后，在第 16 行将这些向量作为参数传递给 *aie::add* 方法，该方法执行向量化加法。乘法、减法和归约操作的处理方式类似，因篇幅所限此处省略。第 6 行显示我们正在遍历网格单元，而第 7 行和第 8 行的指令则指示 AIE 编译器在可能的情况下执行软件流水线，并按照 Xilinx 的最佳实践尝试填满 VLIW 槽位 [8]。

```
1  void cell_advection（输入_流<浮点数>* __restrict in_A,
       输入_流<浮点数>* __restrict in_B，输出_流<
       浮点数>* __restrict out）{
2    aie::vector<浮点数，4>in_data；
3    in_data=readincr_v<4>(in_A);
4
5    int32 cells=(int32) in_data.get(0);
6    for(int i=0;i<cells;i++)
7    chess_准备_进行_流水线处理
8    chess_循环_范围(64,) {
9      aie::vector<浮点数,8> lhs _数值，右端项_数值;
10
11     lhs_数值.insert(0,readincr_v<4>(in_A));
12     lhs_数值.insert(1,readincr_v<4>(in_A));
13     右端项_数值.insert(0,readincr_v<4>(in_B));
14     右端项_数值.insert(1,readincr_v<4>(in_B));
15
16     aie::vector<浮点数，8> vadd=aie::add(lhs_数值，rhs_数值);
17     ....
18   }
```

清单 1：AIE 平流内核代码示意

AIE API 提供了自适应数据流图，可在运行时动态设置参数。但 VCK5000 外壳不支持此功能，因此需要采用替代方案来设置第 6 行中的循环迭代次数。这就是为什么在清单 1 的第 2 至第 5 行中，会从 *in A* 流读取四个浮点数，并提取其中第一个数_，将其转换为整数，然后用作循环遍历的网格单元数量（该对应值在启动时已由可编程逻辑以流的形式传入）。我们必须将网格单元数量读取为浮点数，因为每个内核最多有两个输入和两个输出，而加载操作数需要两个输入。

```
1 类 simpleGraph：公共 图 { 2 私有：
```

3 内核单元_平流_内核[3]；4 public：5 输入_plio in_A[3],in_B[3]；6 输出_plio out[3]；7 8 simpleGraph(){ 9 单元_平流_内核[0]=kernel::create(单元_平流)；10 ... 11 in_A[0] = 输入_plio::create("krnl_0_in0", plio_128_位数, "data/input_A.txt")；12 in_B[0] = 输入_plio::create("krnl_0_in1", plio_128_位数, "data/input_B.txt")；13 out[0] =输出_plio::create("krnl_0_out1", plio_32_位数, "data/output_0.txt")；14 ... 15 for (int=0；i<3；i++){ 16 connect< 流 >(in_A[i].out[0]，单元 _ 平流 _ 内核 [i].in[0])；17 connect<流>(in_B[i].out[0], 单元_平流_内核[i].in[1])；18 connect<流>(单元_平流_内核[i].out[0], out [i].in[0])；19 }}};

清单 2：AIE 图构建代码示意

清单 2 中的代码构建了高级 AIE 图，将内核映射到各个 AI 引擎。代码为每个字段创建了一个平流内核，共创建三个，并且第 11 至 13 行定义了第一个内核在可编程逻辑与 AIE 之间的输入和输出端口（为简洁起见，省略了第二和第三个内核）。这些 AIE 端口在第 15 至 19 行的循环中与内核端口相连。如第 2.1 节所述，AI 引擎之间的物理连接宽度为 32 位，但可以看到，对于输入端口，我们在第 11 和 12 行指定了*plio_128*_位。这会指示 AIE 编译器，PL 上的流宽度为 128 位（类型为 *qdma_axis<128, 0,0,0>*），因此数据将以 128 位数据包的形式到达，并被拆包为四个 32 位流值。这样设计是出于性能考虑：PL 的运行速度远低于 AIE（本例中分别为 300MHz 和 1.2 GHz），因此在一个时钟周期内，PL 可提供四个 32 位数值，随后 AIE 会在每个周期内将其拆包。128 位是支持的最大宽度，这也是清单 1 中组成计算两侧的八个数值通过两次*readincr v* 调用读取的原因；每次调用读取四个数值，分别位于第 11 至 12 行和第 13 至 14 行。_

从清单 2 可以看出，针对三个场分别创建了一个独立的内核实例，并且每个实例都运行在独立的 AI 引擎上。尽管每个场的计算有所不同，但这种差异仅在于所使用的特定模板位置，而底层的算术运算是相同的。因此，我们能够复用相同的内核代码，但从 PL 侧为每个场提供不同的值。该版本的性能以表 1 中的 初始版本行进行报告，可以看出，与改为在 PL 上执行所有算术运算（仅 *PL*（无 *AIE*））相比，其速度明显更慢。

版本	性能 (*GFLOPS*)	相比仅PL
仅 PL（无 AIE）	14.32	-
初始版本	1.99	14%
多内核	4.06	28%
级联流	2.78	19%
级联复用	3.87	27%
多内核窗口	0.91	6%
分块窗口	10.32	72%
主机端归约	16.13	113%
双重向量化	18.48	129%

表 1：不同版本的 AIE 设计与仅PL的非AIE实现的计算性能对比。所有运行均在 Xilinx VCK5000 上使用单精度浮点进行，问题规模为 6,700 万个网格点。

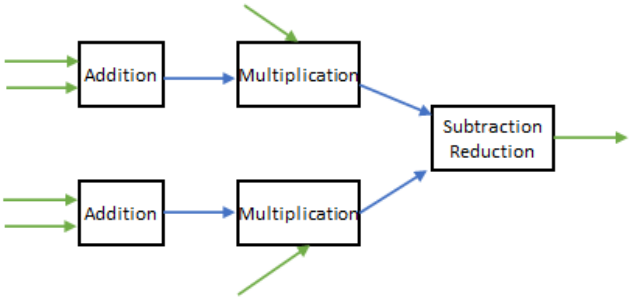

图 3：多内核设计，各组成操作分布在多个 AIE 上运行，并通过流连接。蓝色箭头表示内部流，绿色箭头表示 AIE 与 PL 之间的外部流。

4.1 优化数据传输

AIE 与 PL 之间最大 128 位的数据宽度，是表 1 中报告的初始版本性能不佳的主要原因之一。这是因为每个周期内，PL 每个流只能向 AIE 传输四个单精度浮点数，而实际需要传输 24 个（加法需要 16 个，乘法需要 8 个）。AIE 内核的输入数量限制为两个，这意味着 PL 每个周期最多只能提供八个值。因此，每个网格单元的每个流都需要执行三次写入，这一冲突导致 PL 上的高层次综合代码的启动间隔为三。

为解决这一问题，我们尝试了替代性的内核结构。如图 3 所示，我们将代码拆分为多个内核，每个内核对应一项特定操作。通过将加法和乘法拆分开来，使每个内核分别处理八项计算中的四项，我们将流的总数增加到了六个（每个内核两个）。这样做也存在缺点：由于每个内核现在每个周期只执行四个向量化操作，而不是八个，因此其利用率有所降低；但这种拆分使连接 PL 与 AIE 内核输入的 128 位流从两个增加到了六个。因此，在 PL 上运行的高层次综合内核能够在每个周期传输网格单元所需的全部数据，从而将启动间隔降至 1。

表 1 的多内核行报告了该方法的性能。虽然与初始版本相比，性能提升了一倍，但仍然比仅PL实现慢。在使用 Vitis Analyzer 对我们的多内核代码进行性能分析时，我们发现内核有超过 60% 的时间因读取流而停滞。这是因为，如第 2.1 节所述，AI 引擎之间的物理数据流宽度为 32 位，而内核每次向量化操作会生成 128 位数据。因此，内核会因等待这些数据到达而停滞，无法进行后续运算。

通过级联流连接 AIE 内核是另一种方法。与普通的 32 位流不同，级联流的宽度为 384 位。我们将 128 位结果打包到级联流的 *accfloat* 类型中，并在一个周期内传输全部所需数据。然而，级联流的局限在于，它们通过水平方向传输，在到达一行的边缘时连接到上方的核心，因此在物理上连接 AIE 核心；当到达一行的边缘时，则连接到上方的核心。因此，其连接方式缺乏灵活性：每个 AIE 核心只能从一个预定义的邻近核心接收级联流输入，并向另一个邻近核心提供级联流输出。对于图 3 所示的多内核设计而言，这是一个问题，因为 减法归约 内核需要来自两个内核的输入，实际上需要通过两条级联流向一个 AIE 提供数据，而该架构无法实现这一点。

因此，为了验证级联流是否能够提升性能，我们采用了图 4 所示的设计：一个加法内核执行全部八次加法操作，随后由一个独立的内核执行乘法、减法和归约。表 1 中的级联流项报告了该配置的性能，其性能不佳的主要原因是 PL 上的启动间隔增加到了 2，因为加法内核的流需要每个 PL 周期执行两次写入，而单个内核需要全部八对操作数。为了解决这一问题，我们对级联流方法进行了多路复用，在 AI 引擎上使用两个独立副本，使得平均而言，每个 AIE配置每两个时钟周期接收一次数据。表 1 中的级联复用行报告了该方法的性能。虽然性能有所提升，但仍然低于非AIE纯PL方法的性能。此外，我们还尝试了四路和八路多路复用，但这对性能没有带来可测量的改善。

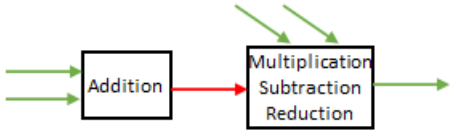

图4：级联流方法，红色箭头表示级联流，绿色箭头表示与可编程逻辑之间的外部流。

到目前为止，我们已经探讨了通过流连接内核和可编程逻辑，但也可以使用能够提供缓冲区的窗口。重要的是，与流提供的每周期 32 位相比，AIE 每周期最多可以从存储器读取 256 位。因此，我们

又恢复了图 3 中的多内核设计，并在内核之间以及可编程逻辑与输入、输出数据之间使用窗口而不是流。这如图 5 所示，多内核窗口的性能数据列于表 1 中。可以看出，性能极差，这是因为我们以逐网格单元为基础操作窗口。这意味着每个内核中不再存在流水线循环，因为在每个网格单元之间，AIE 图都会根据 AIE 工具链的要求停止并重新启动内核，以填充和清空窗口。

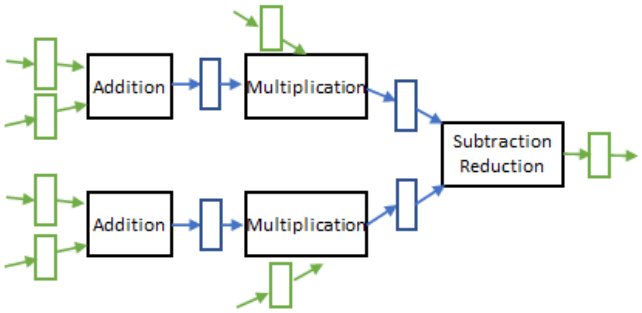

图 5：多内核窗口化方法，彩色方块表示窗口，蓝线连接内部内核，绿色箭头表示往返于可编程逻辑的外部流。

我们将窗口化方法修改为分块处理方式：将多个网格单元的数据缓冲到窗口中，并以乒乓方式运行这些窗口——当一个副本从生产者（可编程逻辑或另一个 AIE 内核）接收并填充数据时，另一个窗口副本则被读取；在 AIE 图的外部迭代之间，这两个副本交替切换。因此，加法、乘法和减法归约 AIE 内核会根据可用数据，同时处理不同的网格单元数据块，从而有效地形成流水线运行。另一个问题是，由于 AIE 内核的运行不同步，例如乘法内核比其对应的加法内核落后一个数据块，这会导致可编程逻辑停滞。这是因为向 AIE 传输数据时，写入乘法流的操作会被阻塞，等待窗口变为空闲状态；而可编程逻辑的这种等待也会阻塞对加法流的写入，而这些写入是推动加法 AIE 内核运行并解锁其乘法内核所必需的。解决方案是在可编程逻辑上为乘法流实现显式的乒乓缓冲，并让数据流区域以 数据块 为单位运行：该区域会同时将当前数据块的数据填入缓冲区，并把前一个数据块的数据流式传输到 AIE 内核。

性能通过表 1 中的分块窗口 进行报告，可以看出，这种方法显著提升了 AIE 的性能，但仍略慢于仅PL。通过 Vitis Analyzer 进行性能分析后，我们发现减法和归约内核的执行时间约为其他内核的两倍，这种工作负载不平衡导致了额外的停顿。因此，我们修改了内核，使其仅执行减法，并将 4 个浮点数流回可编程逻辑，由可编程逻辑将其相加。这一结果在表 1 的 主机端归约 行中进行了报告，其性能优于仅PL。

如前文所述，在这种多内核方法中，每个内核仅处理大小为四的向量，而硬件每个周期能够执行八次单精度浮点运算。借助窗口，可以轻松地并发读取两个网格单元，将第一个网格单元置于向量的低位部分，将第二个网格单元置于高位部分。因此，向量运算现在会对八个操作数执行，实际上每次 AIE 向量化运算可处理两个网格单元。表 1 中将此称为双重向量化，并带来了性能提升，不过提升幅度有限，因为受限于 128 位的最大端口宽度，每个周期 PL 与 AIE 之间仍只能为一个网格单元传输流式数据。

5 个高层次综合计算单元

在第 4 节中，我们重点讨论了单个 PL 高层次综合计算单元（CU）。通过沿平流问题的网格空间进行分解，我们可以扩展到多个高层次综合计算单元；每个单元都拥有网格的一个独立三维部分并独立运行，同时连接到各自的 AIE。采用我们优化后的 AIE 方法（每个高层次综合计算单元需要 15 个 AIE），我们将其性能与其他硬件选项进行了比较，表 2 报告了这些结果。

在 VCK5000 的 AI 引擎上运行的平流内核，其性能由表 2 中的行 *VCK5000 AIEs* 报告。虽然文档中未直接说明，但随着规模扩展，编译过程中会发现最多存在 78 个 128 位 PLIO输入流接口。这是因为每个 AIE瓦片包含八个 32 位的AI引擎到AXI4-Stream通道 [7]，而系统共有 39 个瓦片。因此，连接 AIE 与 PL 的 32 位通道总数为 312 个；由于每个 128 位通道由四条 32 位链路构成，最多可形成 78 个 128 位通道。顺带一提，即使 AIE 直接访问 DRAM 而不经过 PL，也会遇到这一限制，因为数据仍需通过这些相同的物理链路。

每个场有六条输入流、每个计算单元有三个场，因此最多可使用四个高层次综合计算单元。我们总共使用 60 个 AIE；当计算单元数量增加到最多四个时，性能扩展表现良好。因此，这一硬件限制构成了主要瓶颈，因为如果能够扩展到更多计算单元，性能很可能会显著提升。第 4 节强调了在 PL 与 AIE 之间以每周期一个网格单元的速率传输整个网格单元的重要性。在能够实现这一点的两种 AIE 内核设计，即多内核和多路复用级联流中，从这一点来看，多内核更为合适，因为它每个场需要六条输入流，而多路复用级联流需要八条。

Alveo U280 配置了六个 HLS计算单元，这是受 Alveo外壳端口数量限制所能容纳的最大数量。可以看出，Alveo U280 的性能与 VCK5000 使用 AI 引擎时的性能相近，尽管 VCK5000 上只有四个计算单元。考虑到 U280 配备外部 HBM2内存，而 VCK5000 只有 DDR4，这一结果尤其令人瞩目。我们能够在仅PL的 VCK5000 配置中容纳八个计算单元；由于 Versal 内含一个供 HLS内核连接的 NoC，该配置不受端口数量限制。*VCK5000* 组合 结果报告了组合方案的性能

该方案结合了 VCK5000 上的四个 AIE计算单元和六个仅PL计算单元。它利用 AIE 和可编程逻辑进行计算，性能达到 U280 的两倍。

相比之下，通过 OpenMP 进行线程化并使用 GCC 10.2 版本编译、运行于 24 核 Cascade Lake Xeon Platinum处理器上的方案表现不佳，逊于其他所有硬件技术。V100 GPU 版本使用 OpenACC 和 20.9 版本的 Nvidia编译器实现，其性能超过了所有其他 CPU 和 FPGA 配置，这与 [2] 基本一致。尽管 Versal 已缩小与 GPU 的差距，但遗憾的是，AIE 硬件限制最终将 AIE计算单元的数量限制为四个。

描述	性能（GFLOPS）
VCK5000 AIE（*4* 个计算单元）	68.73
VCK5000 仅PL（*8* 个计算单元）	101.78
VCK5000 组合（*4* 个和 *6* 个计算单元）	145.11
Alveo U280（*6* 个计算单元）	72.32
24 核 Xeon Platinum CPU	23.52
V100 GPU	227.89

表 2：FPGA AIE 和仅PL 方法的计算性能，与 24 核 Cascade Lake CPU 和 Nvidia V100 GPU 的对比。所有测试均采用单精度浮点数，问题规模为 6700 万个网格点

6 结论与建议

本文探讨了将 PW 大气平流方案移植到 Versal 上，利用可编程逻辑通过三维移位缓冲区定制内存访问，并由 AIE 执行计算。基于模板的算法广泛应用于高性能计算工作负载，本文所研究的方案是这类算法的一个代表。我们发现，主要挑战在于如何最有效地连接可编程逻辑与 AIE，确保数据在两者之间持续流动。实现方式有多种，我们探讨了硬件和工具链的限制如何影响具体方案的选择，以及这些选择对性能的影响。最终，我们发现最有效的方案是以乒乓方式使用窗口，在 AIE 内核中分块处理数据，并依靠软件流水线循环和完全填充的八路向量化。与其他硬件方案相比，我们发现影响性能的一个主要限制是可编程逻辑与 AIE 之间流的总数，因此无法扩展到四个以上的高层次综合计算单元。不过，VCK5000 上使用 AIE 的四个计算单元，性能与 Alveo U280 上的六个计算单元相当，而后者得益于 HBM2。VCK5000 上的仅 PL 方案相较于其他 FPGA 和 CPU 表现出色，这主要归功于可编程逻辑能够容纳八个高层次综合计算单元。结合 AIE 与可编程逻辑进行计算时，其性能也显著优于其他 FPGA 方案和 CPU。因此，撇开 AIE 不谈，我们的仅 PL 实验表明，Versal 是一种强大的架构，性能优于 Alveo。

从开发角度来看，使用 AIE 有许多优势，这很可能使软件开发人员比使用传统 FPGA 更容易接触 ACAP。这些优势包括：整体编译速度快得多；可以通过仿真开展大部分开发探索，而且仿真本身也很快；如果可编程逻辑与 AIE 之间的接口没有变化，就无需重新构建可编程逻辑（这意味着重新生成比特流只需约一分钟）；以及丰富的性能分析工具链能够揭示代码中的瓶颈所在。不过，让工作负载与架构相匹配至关重要。鉴于可编程逻辑与 AIE 之间的带宽限制，浮点运算与字节比高于本文所述模板计算的内核，很可能更适合 AIE。因此，这项工作的一条重要经验是，应优先关注不会受当前一代可编程逻辑至 AIE 接口限制的内核；浮点运算与字节比高的算法，很可能最能发挥这一架构的优势。

考虑到未来对 Versal 的改进，如果 Xilinx 能在未来的 AIE 版本中将 AIE 之间的物理数据流扩宽至 32 位以上，并将 PL到AI引擎内存的大小从 128 位增加到 256 位，同时支持更多 PLIO数据流，将会很有益处。提高向量大小的灵活性也很有用。例如，无法使用多种大小的单精度浮点向量，包括大小为六的向量，因此我们不得不补齐空值，使其达到八个元素，这增加了 PL 与 AIE 之间传输的数据量。就更广泛的 Vitis技术而言，在高层次综合中无法创建外部 AXI数据流数组（例如，类型为 *qdma_axis<128,0,0,0>*），这导致我们在尝试多路复用时写出了杂乱的代码。这一点很重要，因为与 AIE 接口可能需要比目前高层次综合中最常见的情况更多的 AXI数据流，因此，提高灵活性将大有裨益。

7 致谢

作者谨向资助本项工作的 ExCALIBUR H&ES CGRA 项目表示感谢。我们还感谢 ExCAL- IBUR H&ES FPGA 测试平台和 AMD Xilinx HACC 计划为本项工作提供计算资源访问权限，后者还慷慨地提供了意见和技术建议。为实现开放获取，作者已对因本次投稿产生的任何作者接受稿版本采用知识共享署名（CC BY）许可。

参考文献

[1] Sagheer Ahmad、Sridhar Subramanian、Vamsi Boppana、Shankar Lakka、Fu-Hing Ho、Tomai Knopp、Juanjo Noguera、Gaurav Singh 和 Ralph Wittig。2019。Xilinx 首款 7nm 器件：Versal AI 核心（VC1902）。载于*2019 IEEE Hot Chips* 第 *31* 届研讨会（*HCS*）。IEEE 计算机学会，1–28。[2] Nick Brown。2021年。在 Xilinx 和 Intel FPGA 上加速大气建模中的平流。载于*2021*年 *IEEE* 集群计算国际会议（*CLUSTER*）。IEEE，767–774。[3] Nick Brown 等。2015。高度可扩展的英国气象局 NERC 云模型。载于第 *3* 届国际百亿亿次应用与软件会议论文集。爱丁堡大学，132–137。[4] Brian Gaide、Dinesh Gaitonde、Chirag Ravishankar 和 Trevor Bauer。2019。Xilinx 自适应计算加速平台：VersalTM 架构。载于 *2019* 年 *ACM/SIGDA* 国际现场可编程门阵列研讨会论文集。84–93。

[5] David Lee、Gregory Allen、Matthew Cannon、Hunter Earnest、Paul Thelen、Nathaniel Dodds、Jeffrey McCasland 和 Carol Chen。2021年。*Xilinx Versal ACAP* 重离子辐照的初步结果。技术报告。美国桑迪亚国家实验室（SNL-NM），阿尔伯克基，新墨西哥州（美国）。[6] Steve A Piacsek 和 Gareth P Williams。1970年。对流差分格式的守恒性质。《计算物理学杂志》6，3（1970年），392–405。[7] Xilinx。2021年。Versal ACAP AI 引擎架构手册（AM009）。https://docs.xilinx.com/r/en-US/am009-versal-ai-engine

[8] Xilinx。2022年。《AI 引擎内核编码最佳实践指南》（UG1079））。https://docs.xilinx.com/r/en-US/ug1079-ai-engine-kernel-coding [9] Xilinx。2022年。《Versal ACAP AI 引擎编程环境用户指南》（UG1076）。https://docs.xilinx.com/r/en-US/ug1076-ai-engine-environment[10] Chengming Zhang、Tong Geng、Anqi Guo、Jiannan Tian、Martin Herbordt、Ang Li 和 Dingwen Tao。2022年。H-GCN：基于 Versal ACAP 架构的图卷积网络加速器。*arXiv* 预印本 *arXiv:2206.13734*（2022年）。