Morpheus 释放：面向新兴体系结构的快速跨平台稀疏矩阵-向量乘法

Christodoulos Stylianou爱丁堡大学 *EPCC*英国爱丁堡 c.stylianou@ed.ac.uk

Mark Klaisoongnoen爱丁堡大学 *EPCC*英国爱丁堡 mark.klaisoongnoen@ed.ac.uk

Ricardo Jesus爱丁堡大学 *EPCC*英国爱丁堡 rjj@ed.ac.uk

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

Mich`ele Weiland爱丁堡大学 *EPCC*英国爱丁堡 m.weiland@epcc.ed.ac.uk

arXiv:2304.09511v1 [cs.DC] 19 Apr 2023

摘要——**稀疏矩阵和线性代数是科学模拟的核心。多年来，研究人员已经开发出 70 多种稀疏矩阵存储格式，面向各种硬件体系结构和矩阵类型；这些格式分别利用某种体系结构的特定优势，或矩阵特定的稀疏模式。**

在本研究中，我们探讨了坐标格式、压缩稀疏行格式和对角线格式等存储格式是否适用于 AArch64 CPU 和现场可编程门阵列（FPGA）等新兴体系结构。此外，我们详细介绍了针对这些目标平台的硬件特定优化，并评估了将各项成果集成到 *Morpheus* 中的潜力。Morpheus 是一个现代库，为稀疏矩阵提供抽象，目前支持 x86 CPU 以及英伟达/AMD GPU。最后，我们将 Morpheus 加速的 HPCG 基准测试与厂商优化的实现进行性能比较，以验证我们的工作。

索引词——**稀疏矩阵存储格式、AArch64、FPGA、性能可移植性、生产力**

I. 引言

自诞生以来，稀疏矩阵已成为众多科学与工程应用的核心。它们能够高效存储矩阵的非零值，从而减少矩阵的内存占用并消除冗余计算，使得处理规模更大的问题成为可能。多年来，人们已提出超过 70 种稀疏矩阵存储格式（即数据结构） [1],，每种格式都利用矩阵或目标硬件体系结构的不同特性来实现更高的性能。许多用于求解大规模线性系统和特征值问题的迭代方法——这些问题通常出现在各种科学与工程应用中——都包含大量稀疏矩阵-向量乘法（SpMV）操作，而这些操作往往占据应用运行时的大部分时间。文献表明，没有任何一种格式能够在所有类型的矩阵或硬件上都实现最优性能 [1]–[5],，因此每当新平台出现时，人们都会重新关注 SpMV 的优化。

Morpheus [6] 是一个 C++ 库，为稀疏矩阵提供抽象，使其能够在传统后端之间高效、透明地切换稀疏矩阵存储格式，包括 x86 CPU 和英伟达/AMD GPU。通过动态调整底层稀疏矩阵数据结构，以最佳适配矩阵的某项操作、目标架构或稀疏模式，*Morpheus* 带来了新的优化机会，从而提升性能 [6]。截至撰写本文时，*Morpheus* 支持三种核心格式：坐标格式（COO）、压缩稀疏行格式（CSR）和对角线格式（DIA）。

在本文中，我们研究了两种非传统高性能计算（HPC）体系结构——AArch64 CPU 和 FPGA——上不同格式的 SpMV 性能；目前，这两种体系结构在高性能计算领域都正获得越来越多的关注。此外，我们还利用每种目标平台的特定硬件特性（例如 AArch64 上的可扩展向量扩展（SVE））来优化内核，并研究将这些优化集成到*Morpheus*中的挑战。总而言之，我们的贡献如下：

- 我们介绍了如何将 Arm 性能库（ArmPL）集成到 Morpheus 中，以支持 AArch64 目标平台，使 Morpheus 用户能够开箱即用地使用 ArmPL 的 COO 和压缩稀疏行格式性能。
- 为弥补 ArmPL 不支持对角线格式的不足，并更全面地评估 ArmPL 的性能，我们针对坐标格式、压缩稀疏行格式和对角线格式矩阵开发了经过 SVE 优化的稀疏矩阵-向量乘法例程。
- 针对 SuiteSparse [7]集合中的 2100 多个矩阵，我们展示了稀疏矩阵-向量乘法内核在基于 A64FX（AArch64）的 HPE Apollo 80 集群上的性能。
- 我们在 HPE Apollo 80 系统上使用 HPCG[8],评估了这些工作（即将 ArmPL 和经过 SVE 优化的例程集成到 Morpheus 中），并将基于 Morpheus 的 HPCG 实现 [9] 与针对 Arm 优化的版本 [10] 进行了比较。
- 我们为三种格式分别提供了适用于现场可编程门阵列的稀疏矩阵-向量乘法内核实现。我们在 SuiteSparse 数据集上评估了性能，并介绍了将这些实现移植到*Morpheus* 时面临的挑战。

II. 动机

每当出现新的架构时，人们都会提出新的存储格式，旨在利用新硬件的特性和优势。尽管已有 70 多种格式，但没有一种格式能在不同硬件、操作和稀疏模式下都表现最佳。因此，能够动态切换格式为优化和提升性能带来了新的机会。采用新格式可能是一个繁琐的过程，因为这需要对源代码进行大量修改。诸如*PETSc* [11], *GINKGO* [12] 和*Morpheus* [6] 这样的库通过各种抽象提供多种格式，使用户能够在运行时切换格式。换言之，用户只需对源代码进行少量修改，就能利用库中新增的格式，从而减轻开发负担。

然而，在采用新硬件时，库的接口可能需要进行重大调整，尤其是当硬件采用了库尚未集成的新编程模型时。例如，当 GPU 在高性能计算领域兴起时，库必须不断演进，以管理 CPU 与 GPU 之间两个大体独立的内存空间，并支持加速器模型，即由 CPU 将工作卸载到 GPU。为了确保软件在硬件体系结构的整个生命周期（乃至更长时间）内始终保持高性能，我们需要确保软件能够适应当前和未来硬件迭代所提出的要求。

在这项工作中，我们研究了 AArch64 CPU 和 FPGA 这两种截然不同、目前在高性能计算领域日益受到关注的体系结构。我们考察了它们在稀疏线性运算（例如稀疏矩阵-向量乘法）中的性能，以及将其集成到 *Morpheus* 中所面临的挑战。将代码集成到 *Morpheus* 以支持上述目标平台，会带来不同的挑战，也需要采用不同的集成方法，因此这些平台可作为集成其他体系结构的代表。

III. 背景

A. 新兴体系结构

我们提出将两种新兴体系结构集成到 *Morpheus* 库中：AArch64 CPU 和 FPGA。通过将三个核心 *Morpheus* 存储格式（坐标格式、压缩稀疏行格式和对角线格式）的稀疏矩阵-向量乘法例程移植到 AArch64 CPU 和 FPGA，我们探索了将其集成到 Morpheus 库时在性能优化和性能可移植性方面面临的挑战。

1) AArch64 CPU：尽管 Arm CPU 在高性能计算领域还是相对新兴的架构，但与更为传统的 x86 处理器相比，它们已经展现出极强的竞争力 [13]–[16]。现代 AArch64 CPU 能够实现高性能的关键因素之一是 SVE [17]–[18], , 这是一种用于单指令多数据（SIMD）处理的高级架构扩展，具有长（可扩展）向量、聚集加载和散布存储指令，以及逐通道谓词化。这些特性使 SVE 成为实现快速稀疏矩阵-向量乘法计算的理想架构。

为加速 Arm 硬件在高性能计算领域的应用，Arm 开发了 ArmPL [19]„ 这是一套面向高性能计算应用、针对 AArch64 处理器优化的核心例程。ArmPL 包含用于单线程和 OpenMP 多线程处理的 BLAS、LAPACK、快速傅里叶变换、稀疏计算、libamath（libm 的子集）和 libastring（用于字符串处理的 libc 子集）例程。ArmPL 提供的稀疏线性代数例程支持对稠密矩阵以及压缩稀疏行格式、压缩稀疏列、坐标格式和块压缩稀疏行矩阵进行高性能稀疏矩阵-向量乘法。不过，截至 23.04 版本（撰写本文时的最新版本），ArmPL 尚不支持对角线存储格式。

2) FPGA：现场可编程门阵列（FPGA）在大量可配置互连构成的网络中提供了数量极为庞大的可配置逻辑组件。现代 FPGA 还包含硬化组件，例如块随机存取存储器（BRAM），可提供类似于 CPU 一级缓存的快速片上存储器，以及用于执行浮点运算的 DSP 切片。现代 FPGA 通常还配备外部高带宽内存（第二代高带宽内存）、DDR 和高性能网络功能。长期以来，FPGA 面临的一项重大挑战是编程这项技术需要投入大量时间，且开发者必须具备详细的硬件层知识。尽管如此，近年来 FPGA 硬件和软件开发生态系统的能力已大幅提升；借助英特尔的 Quartus Prime 和赛灵思的 Vitis 等工具链，软件开发者现在可以使用 C 或 C++编写代码，通过高级综合（HLS）为 FPGA 编程并加速高性能计算工作负载。如今，FPGA 编程越来越像软件开发，而非硬件设计；因此，降低这些设备的编程门槛，使众多社区得以在各自领域研究和探索 FPGA [20]–[22]。

能够在电子层面根据代码定制硬件，为围绕内存访问和数据传输实现自定义优化技术提供了可能。然而，为了获得最佳性能，程序员必须将其算法改造为数据流风格 [23] ，而且这通常也能比传统体系结构实现高得多的能效 [24]。AMD 赛灵思最新一代产品被称为 Versal 自适应计算加速平台（ACAP）[25], ，它将 FPGA 芯片上的可编程逻辑（PL）资源与 400 多个 AI 引擎（AIE）相结合。这些 AIE 固化在芯片上，每个都代表一个超长指令字（VLIW）处理器，能够在每个时钟周期执行七条指令，并通过高速片上网络（NoC）在 AIE 之间以及与 FPGA 上的 PL 互连。凭借每个 AIE 的 8 路向量化能力，这种计算能力的升级对于 SpMV 等在这方面具有巨大潜力的内核尤其值得关注。

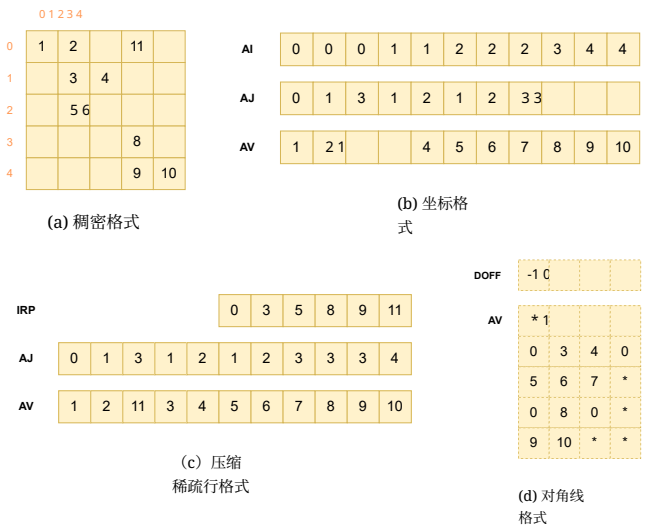


图 1：一个包含 11 个非零值的 5×5 稠密矩阵及其对应的 3 种稀疏矩阵存储格式。

B. 稀疏矩阵存储格式

稀疏矩阵利用矩阵中大多数系数为零这一特性，不显式存储这些值。换句话说，稀疏矩阵存储格式只存储非零系数，以及重建每个系数在稠密矩阵中原始位置所必需的信息。每种存储格式都以不同方式重建每个系数的原始索引，因此，如图 1 所示，不同格式可能具有截然不同的内存布局。由于各种格式所采用的底层数据结构差异显著，访问和操作每种格式中的条目可能会导致不同的内存访问模式、开销和接口。

图 1 展示了一个 5×5 稠密矩阵在本文中使用的三种稀疏存储格式中的表示：坐标格式、压缩稀疏行格式和对角线格式。最基础且广为人知的格式是坐标格式和压缩稀疏行格式。二者都被视为通用型格式，适用于具有任意稀疏模式、面向各种目标架构的广泛矩阵。下面简要介绍这些格式。更全面的说明请参阅 Saad, Y. [26]。

1) 坐标格式（图 *1b*）：使用三个数组，将每个非零元素（AV）与其列索引（AJ）和行索引（AI）显式存储，但不保证元素的排列顺序。坐标格式的 SpMV算法见算法 1。

算法 1 坐标格式稀疏矩阵-向量乘法

对于 i=0:NNZ，**执行**
  y(ai(i)) += av(i)*x(aj(i));
结束循环

2) CSR（图*1c*）：作为对坐标格式的优化而实现，其中AI数组经过压缩后生成行指针数组。因此，CSR 显式存储列索引和非零值，同时还使用指针数组（IRP）标记各行的边界，从而通过压缩行索引大幅减少该格式的内存占用。由于行指针用于表示每行中第一个非零元素的位置，而最后一个条目表示矩阵中的非零元素总数，因此 CSR 会自然地对各行进行排序，但不对每行内部的元素排序。CSR 的稀疏矩阵-向量乘法算法见算法 2。

算法 2 压缩稀疏行格式 稀疏矩阵-向量乘法

对于i=0:nrows **执行**
  $sum = 0$;
  对于 j=irp(i):irp(i+1)**执行**
    sum += av(j)*x(aj(j));
  结束 对于
  y(i) = sum;
结束循环

3) 对角线格式（图*1d*）：这是一种专用格式，最初旨在在向量架构上实现最佳性能。它适用于规则稀疏模式。DIA 使用一个二维数组，其中每列存储矩阵（AV）一条对角线上的系数，并使用整数偏移数组（DOFF）记录每条对角线的起始位置。因此，对角线存储格式适用于非零结构主要分布在对角线上的矩阵，例如通过有限差分法（FDM）等离散化方法得到的带状矩阵。DIA 的稀疏矩阵-向量乘法算法如算法 3 所示。

算法 3 DIA 稀疏矩阵-向量乘法

对于i=0:nrows **执行**
  $sum = 0$;
  对于j=0:ndiags **执行**
    k =i + DOFF(j)
    如果 k≥0**且** k<N **则**
      sum += av(i,j)*x(k);
    结束如果
  结束循环
  y(i) = sum;
结束循环

IV. AArch64 CPU 上的稀疏矩阵-向量乘法实现

针对大多数 AArch64 CPU 等通用处理器优化代码，通常涉及两种方式：在较高层次上，应用程序员可以选择使用特定目标平台的库，这些库能针对特定目标高效实现核心算法和例程；在较低层次上，应用程序员则可以自行编写针对特定 CPU（微）架构的高效代码，通常通过内置函数（或内置函数）实现。这类函数会被编译器特殊处理，使程序员能够直接使用目标架构的特性；或者也可以为目标平台编写显式汇编语言。在本研究中，我们

探索了将这两种面向 AArch64 CPU 的优化方式集成到 *Morpheus* 中，并将在本节对此进行介绍。

Arm 性能库（ArmPL） [19] 是 Arm 为面向高性能计算应用的 AArch64 目标平台（尤其是基于 Neoverse 的平台）开发的一组核心例程。它包含 BLAS、LAPACK、快速傅里叶变换、稀疏计算、libamath（libm 的子集）和 libastring（libc 中用于字符串处理的子集）例程，支持通过 C 和 Fortran 接口进行单线程和多线程处理。ArmPL 的稀疏例程支持稠密矩阵、压缩稀疏行格式、压缩稀疏列、坐标格式和块压缩稀疏行矩阵。这些例程通过类似于 FFTW 的应用程序接口提供，其中问题的描述与其执行相互独立。在此意义上，要使用 ArmPL 设置稀疏矩阵-向量乘法运算，我们首先要创建一个稀疏矩阵句柄。这通过 armpl_spmat_create_* 例程系列实现。通常，该矩阵以压缩稀疏行格式等常见格式提供。然而，在我们的案例中，句柄会针对当时使用的具体格式创建。然后，通过 armpl_spmat_hint 调用向句柄提供提示，以尝试提高后续稀疏矩阵-向量乘法调用的加速比。随后，句柄会用于一个优化阶段，该阶段类似于其他库（例如前面提到的 FFTW）中的优化阶段，库会尝试为特定矩阵和目标平台确定最佳算法和实现。此步骤通过 armpl_spmv_optimize 发起。完成这些优化后，可以反复使用该句柄，通过 armpl_*_exec_* 例程系列执行稀疏矩阵-向量乘法以及其他稀疏代数计算。不再需要该句柄后，可以调用 armpl_spmat_destroy 将其销毁。鉴于 ArmPL 在 AArch64 目标平台上的高性能和易用性，我们在本文中选择它来探索如何将目标平台专用库（ArmPL 就是其中一个示例）集成到 *Morpheus* 中，以及这些库能够以何种性能水平实现集成，从而让 *Morpheus*用户透明地享受其高性能。

为 AArch64 CPU 开发高度优化代码的另一种方法，是利用 Arm C 语言扩展（ACLE） [27]。ACLE 是一组编译器内建函数，用于开放 Arm 架构的高级功能，旨在支持开发可在不同编译器和不同 Arm 微架构上移植的应用程序与库。Arm 架构（尤其是 AArch64）最具颠覆性的扩展之一是可扩展向量扩展（SVE） [17]。与 Arm 的 Neon 和 x86 的 AVX 扩展等当代其他单指令多数据（SIMD）扩展不同，SVE 是一种“向量长度无关”（VLA）的向量扩展。这意味着程序员无需针对特定宽度的向量寄存器编程；相反，他们采用一种略有不同的编程模型，在该模型中，向量寄存器的宽度在编译时未知。实际上，这一特性使 SVE 具有很强的可移植性，因为相同的代码（事实上，包括二进制文件）可以在向量寄存器宽度不同的硬件上透明运行。除了提供这种额外的灵活性和可移植性外，SVE 还是一套极为完整的指令集，支持非常适用于高性能计算（HPC）和机器学习（ML）应用的指令。SVE 的高性能特性包括逐通道谓词化（即针对每个向量元素进行控制）、聚集加载和散布存储、推测向量化，以及水平归约和树形归约。鉴于 SpMV 计算具有不规则的间接访问模式（如第三节B所示），SVE 的聚集加载/散布存储对其尤为有用。基于这些原因，本研究使用 ACLE 为坐标格式、压缩稀疏行格式和对角线格式矩阵开发了支持 SVE 的 SpMV 内核实现。这些实现主要是将 *Morpheus* 中现有 SpMV 内核的默认 C++版本转写为 ACLE。下文将重点介绍我们支持 SVE 的 SpMV 内核的两个主要实现细节。为便于讨论，全文将采用第三节B中介绍的算法。

COO 内核中，输出向量 y 通过行索引向量 ai 进行间接访问，这使内核循环的向量化变得复杂。这是因为 ai 中不同的元素可能指向 y 中的同一个元素。在这种情况下，必须先将对 y 的写入串行化，或在发出写入前进行累加，以确保只执行一次写入且不会丢失对 y 的更新。在我们支持 SVE 的 COO 实现中，我们选择了第二种方法：每次 i 迭代中，只处理与 ai(i) 匹配（即相等）的元素 (ai(i), ai(i+1), ai(i+2), ...)。我们利用 SVE 的谓词化功能创建这些元素的掩码，并仅对它们进行操作。这样，在将乘积 (av(i)*x(aj(i)), av(i+1)*x(aj(i+1)), ...) 写入 y 之前，可以先累加所有对应于同一个 ai(i) 的乘积，从而只对 y 执行一次累加。用 C 和 ACLE 伪代码表示如下：

```
1 vbool_t pg;
2 for(i; i< NNZ; i += vcntp(pg, pg)) { 3 //为i <NNZ中的值生成掩码 4 pg =vwhilelt(i,NNZ); 5 6 //加载ai(i,i+1,...)的值 7 vidx_tvai =vld1su(pg,ai+i); 8 9 //为元素生成掩码 10 // (ai(i), ai(i+1), ...) ==ai(i) 11 pg =svcmpeq(pg,vai,ai[i]); 12 13 //加载aj、av和x的值 14 vidx_tvaj =vld1su(pg,aj+i); 15 vtype_t vav =svld1(pg,av+i); 16 vtype_tvx =svld1_gather_index(pg,x,vaj); 17 18 //计算乘积av(i)*x(aj(i)) 19 vtype_tvr = svmul_x(pg,vav,vx); 20 21 //累加乘积 22 yval[ai[i]] += svaddv(pg,vr); 23 }
```

尽管当掩码 pg 变得过于“空洞”（即活跃元素过少）时，这种方法可能效率较低，但在实际测试中，这种情况并不常见。因此，如图 4-a 所示，该策略使我们能够获得显著高于默认实现（即编译器生成的实现）和

ArmPL 实现的加速比。

我们实现支持 SVE 的稀疏矩阵-向量乘法内核时，还有一个重要而微妙之处：DIA 存储格式的向量化方式。与更常见的做法（向量化内层循环 j）不同，我们向量化的是外层循环 i。这样做有两个原因：(i) av 中的内存访问在 j 方向上是连续的（即步长为 1），因此一次加载多个 (av(i, j)、av(i+1, j)……)，然后按顺序遍历 (j、j+1……)，可以提高缓存利用率；(ii) 这样就无需在写入 y 之前进行水平归约来累加 sum 的值。此外，我们再次利用 SVE 的谓词化功能，对每次内层迭代中有效的 k 索引进行掩码处理。伪代码如下：

```
1 vidx_t vidx =vindex(0, ndiags); 2 for(i = 0; i <rows; i += vcnt()) { 3 // 初始化 sum 4 vtype_t vsum = vdup(0); 5 6 // 为 i < nrows 中的值创建掩码 7 vbool_t pg = vwhilelt(i, nrows); 8 9 for(index_type j = 0; j <ndiags; j++) { 10 index_type k =i + doff[j]; 11 12 // 为有效的 k 值生成掩码 13 // p1 =k < 014 // p2 =k < N 15 // pm =p2 && !p1 16 vbool_t p1 =vwhilelt(k, 0); 17 vbool_t p2 =vwhilelt(k, N); 18 vbool_t pm =svbic_z(pg, p2, p1); 19 20 // 加载 av 和 x 21 vtype_t vav =22 svld1_gather_index(pm, av+i*ndiags+j, vidx); 23 vtype_t vx =svld1(pm, x+k); 24 25 // 计算乘积 av(i, j)*x(k)，并 26 // 将其累加 27 vsum =svmla_m(pm, vsum, vav, vx); 28 } 29 30 // 将结果存入 (y(i)、y(i+1)……) 31 svst1(pg, y+i,vsum); 32 }
```

同样，谓词 pm 中可能只有太少的活动元素，导致向量化无法带来收益，不过这种情况并不常见。此外，在我们的测试中（图 4-c），选择外层循环向量化而非内层循环向量化，带来了显著的加速比。不过，我们注意到，由于这种向量化涉及复杂的控制流，我们测试的编译器——GCC 11.2.0 和 LLVM 15.0.7——都无法自动执行外层循环向量化。

V. FPGA 上的稀疏矩阵-向量乘法实现

现场可编程门阵列提供了可编程逻辑，可以在电子层面进行配置，以实现针对特定算法定制的硬件。现场可编程门阵列的运行方式与传统 CPU 等冯·诺依曼体系结构的运行方式存在根本差异。因此，在移植基于 CPU 的算法时，需要将其重新设计为适用于此类设备的数据流计算风格 [23]。数据流，这一在现场可编程门阵列上追求性能的基本概念，通常围绕并发运行的阶段构建，这些阶段称为数据流阶段，彼此之间以流式方式传输数据；每个阶段都包含独立的流水线，并且每个时钟周期都会开始处理一个新的迭代。在 AMD-Xilinx 现场可编程门阵列上，本研究重点关注的正是这种情况，通常需要定义数据流区域，这些区域通过 HLS 流连接。与访问全局内存相比，使用 HLS 流通常可以减少读写所需的周期数。

将三种 *Morpheus* 存储格式——坐标格式、压缩稀疏行格式和对角线存储格式——移植到 FPGA 后，我们将每种格式分别实现为独立的内核，并生成三种不同的比特流来配置 FPGA。每种比特流都包含一个采用相应存储格式算法的稀疏矩阵-向量乘法内核，分别如算法 1（坐标格式）、算法 2（压缩稀疏行格式）和算法 3（对角线存储格式）所示。我们使用 AMD-Xilinx 的 HLS 工具链 Vitis，以 C 和 C++ 编写代码。这意味着我们需要在 CPU 上设置主机代码，并在本研究使用的 AMD-Xilinx Alveo U280 现场可编程门阵列上设置 HLS 内核。AMD-Xilinx 的主机-设备模型基于 OpenCL，因此我们需要 (i) 在主机代码中初始化设备，(ii) 创建用于输入和输出数据的 OpenCL 缓冲区，(iii) 将所需的输入矩阵和向量数据传输到设备，(iv) 数据传输完毕并存入全局设备内存后，在设备上执行内核，最后 (v) 将结果向量传回主机。输入和输出数据所需的大小因矩阵维度而异；我们以单精度浮点格式传输数据并运行设备内核。坐标格式和压缩稀疏行格式的输入元素数量及数据大小大致相近，而对角线存储格式所需的矩阵值则多得多，具体取决于填充行的数量和要存储的对角线数量：对于坐标格式，所有输入（AI、AJ、AV）都需要 n=NNZ 个元素，其中整数占 32 位，浮点数占 32 位；对于压缩稀疏行格式，IRP 需要 n=nrows+1 个 32 位整数，AJ 和 AV 均需要 n=NNZ 个 32 位浮点数；对于对角线存储格式，DOFF 需要 n=ndiags 个 32 位整数，AV 则需要 n=padded_rows*ndiags 个 32 位浮点数。与坐标格式和压缩稀疏行格式的输入数据需求相比，对角线存储格式需要的输入元素显著更多，因此增加了从主机向设备传输的数据量。

假设大多数稀疏矩阵-向量乘法计算都只是 HPCG 等大型内核中的众多例程之一，那么可以合理地认为输入数据已在设备上，因此我们仅报告设备上的内核执行时间（不包括设备初始化、下载比特流和配置硬件的准备时间，也不包括数据复制到设备和从设备复制回来的时间）。我们的实现通过单个 OpenCL 缓冲区将数据复制到设备并复制回主机；使用 Vitis 工具链时，这意味着我们受限于工具所规定的单个缓冲区大小上限 4 GB[1]，，无法运行需要的单个输入缓冲区超过该大小限制的

[1]https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Buffer-Creation-and-Data-Transfer

矩阵。Alveo U280 FPGA 板卡上配有 8 GB 高带宽内存（HBM）和 32 GB DDR DRAM。我们利用速度更快的高带宽内存进行设备间的数据传输，因此不会运行所需累计输入数据超过 8 GB 的更大矩阵基准测试。

图 2 展示了我们使用 AMD Xilinx Vitis HLS 在现场可编程门阵列上实现的坐标格式数据流版本的结构。每个绿色方框都是一个独立的数据流区域，可并发运行；方框之间的箭头表示数据流，这些数据从一个时钟周期流向下一个时钟周期。图 2 中的紫色方框表示与外部高带宽内存的连接；所有读写操作都按 512 位的块打包，以便充分利用内存控制器。数据流区域并行运行，从第二代高带宽内存加载数据，然后将单个数据元素传递给后续阶段。图 2 中的虚线表示乒乓缓冲区。这是一种 FPGA 编程中常用的双缓冲技术：数据流阶段向一个缓冲区并发写入数据，而后续阶段则从缓冲区的先前副本中读取数据；到预定时刻，两者交换。

需要关注的一点是，构成数据流区域的流水线应确保启动间隔（II）为 1。启动间隔指的是循环开始下一次迭代前所需的时钟周期数。启动间隔为 1（通常写作 II=1）意味着数据流设计基本上可以每个时钟周期产生一个结果。考虑到现场可编程门阵列的时钟频率要低得多，通常约为 300 MHz，而 CPU 和 GPU 的时钟频率更高，因此在计算时充分利用每个时钟周期对性能至关重要。

以下伪代码描述了图 2 数据流区域中的归约阶段，该图展示了我们如何逐步实现最优 II 为 1：

```
1 voidreduce(const unsigned int A_nrows, 2 const unsigned int A_nnnz, 3 unsigned int A_rind [MAX_NROWS] ,4 hls::stream<dtype>&sum_stream, 5 dtype y_val [MAX_NROWS ]6 ) { 7 nrows_循环: 8 for (unsigned int row_index=0; row_index<A_nrows; 9 row_index++) { 10 dtype acc_part [LATENCY ] ={0,0,0,0,0,0,0,0}; 11 nnnz_循环: 12 for (unsigned int i=0; i<A_nnnz; 13 i+=LATENCY) { 14 #pragma HLS pipeline 15 acc_partial_循环: 16 for (unsigned int j=0; j<LATENCY; j++ ) { 17 #pragma HLS 循环展开 18 dtype sum =sum_stream.read(); 19 unsigned int tmp_A_rind =20 A_rind [i*LATENCY+j ]; 21 if(tmp_A_rind ==row_index) { 22 acc_part [j ] +=sum; 23 } 24 } 25 } 26 acc_final_循环: 27 for(unsigned int j=0; j<LATENCY; j++) { 28 #pragma HLS 循环展开
29 y_val [row_index ] =acc_part [j ];30 }
31 }}
```

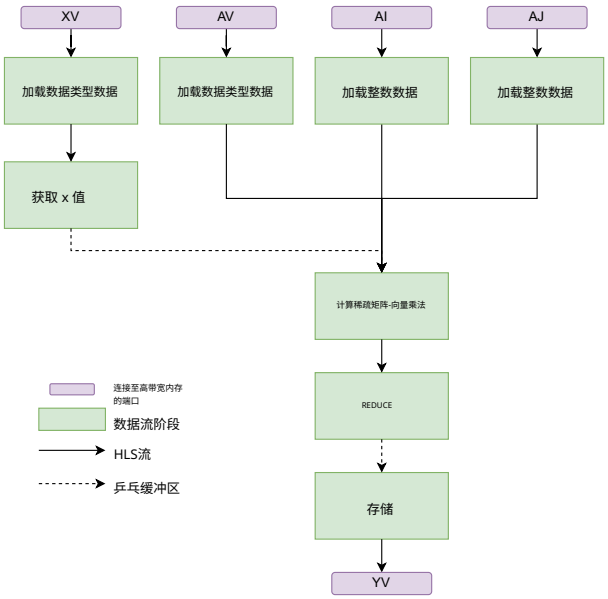


图2：采用现场可编程门阵列实现的 COO 算法的数据流区域，其中各数据流阶段并行运行，并通过 HLS 流和在 UltraRAM 中实现的乒乓缓冲区相互连接。

在 FPGA 上，COO算法1 中依据每个非零元素的行索引 实现对 YV 的朴素归约时，浮点加法需要八个时钟周期。HLS工具链检测到了空间依赖，因此没有将我们的归约函数以低于八的 II 进行流水线化。我们在伪代码第 15 行的内层循环 acc_partial_loop 中，将归约拆分为 LATENCY=8 个部分累加；工具链可以将该循环完全展开。外层 nnnz_loop 仍会由 HLS工具链以 II=8 进行流水线化，而通过完全展开内层循环，实际上可以做到每个周期产生一个结果。与 COO算法一样，我们只对具有相同行索引的元素进行归约，因此在第 21 行引入条件判断，仅累加具有相同行索引的部分和。随后，在第 26 行，我们对先前计算出的部分累加结果进行最终累加，并根据相应的行索引将归约后的值写入 YV（此处为 y_val）。所示归约以浮点加法的 LATENCY 为八为基础，即每次加法需要八个周期。为确保 SpMV内核结束时 HLS 流中没有残留数据，在主机上创建 OpenCL 缓冲区并将数据传输到 FPGA 的全局内存之前，我们会对所有输入数据结构进行填充，使其大小为 LATENCY（此处为八）的倍数。采用我们的方法后，优化后的 SpMV内核不受特定矩阵维度的限制，例如也可以处理行数和列数不是八的倍数的矩阵，但仍受单个缓冲区大小和累积输入缓冲区大小的限制。

VI. 与 Morpheus 集成

Morpheus 是一个 C++仅头文件库，主要依赖模板和元编程来实现特定的多态能力。它采用函数式设计，将数据结构（容器）与函数（算法）分离，算法作用于容器。为了支持各种硬件平台和内存层次结构，*Morpheus* 采用了两种抽象概念：1）执行空间，用于指定代码的执行位置；2）内存空间，用于指定数据在内存中的存放位置。*Morpheus* 目前支持四种执行空间：1）串行（顺序），2）OpenMP（多线程），3）CUDA（NVIDIA GPU）和 4）HIP（AMD GPU），每种执行空间也对应一个独立的后端。因此，每种受支持的算法都可以使用统一的接口。通过指定要使用的后端，也就选择了不同的执行空间。此外，*Morpheus* 还提供数据管理例程，以有效管理不同内存空间之间的数据传输。为支持异构硬件，*Morpheus*采用主机-设备 模型，从而能够在不同内存空间之间执行数据管理操作。

将第四节提出的特定优化集成到现有后端中，与集成第五节所提出的新后端相比，面临着不同的挑战，需要克服这些挑战。下面分别介绍这两种方法面临的挑战。

A. 优化

Morpheus 会对算法进行编译时内省，并检查用户作为参数提供的后端，以确定每次应分派哪种算法实现。这意味着对于每个后端，只能分派一种且仅一种算法实现。若要在单个后端中使用多种实现，只能重新编译代码。

第四节提出的优化实际上构成了 Serial 后端中稀疏矩阵-向量乘法例程的替代算法实现。鉴于*Morpheus*当前的状态，只能通过编译时标志指定要启用的版本。未来当然可以将这一选择移至运行时，这样便能启用所有版本，但每次执行算法时都会产生运行时开销。不过，只有在选择最优版本（依据矩阵的稀疏模式）所带来的性能提升足以抵消这一代价时，这样做才可行。

要将 ArmPL 提供的 COO 和 CSR 稀疏矩阵-向量乘法实现集成到各自对应的 *Morpheus* 例程中，需要将*Morpheus* 中*CooMatrix*和*CsrMatrix* 容器的数据转换为*armpl spmatt* ，以便传递给 ArmPL 专用例程。由于*CooMatrix*和*CsrMatrix* 的数据布局满足与 ArmPL 内部相同的要求，因此，*armpl spmatt* 句柄以 ARMPL_SPARSE_CREATE_NOCOPY 模式创建，这意味着传递的只是数据指针，而不是实际的数据副本。为避免每次执行稀疏矩阵-向量乘法时都为同一个矩阵创建新句柄，每种格式都会创建一个工作区作为单例 [28]。换言之，每当使用新矩阵执行稀疏矩阵-向量乘法时，工作区都会负责注册新创建的句柄，并在后续调用中复用该句柄，而不是创建新句柄。另一个需要解决的挑战是，如何将*Morpheus* 容器的多态行为适配到显式的 ArmPL 函数调用中；这些函数调用的名称中包含格式和数值类型信息。表 I 展示了一个必须实现的适配器调用示例：它将*Morpheus* 中可用的多态容器映射为显式函数调用，以使用 ArmPL 创建 COO 或 CSR 矩阵的 ArmPL 句柄。

表 I：将*Morpheus* 的多态行为适配为 ArmPL 所需的显式调用。

Morpheus 格式	适配器调用	ArmPL 调用
CooMatrix<双精度浮点数> CooMatrix<浮点数>	创建_坐标格式<T>	armpl_spmat_create_坐标格式_d armpl_spmat_创建_坐标格式_s
CsrMatrix<double> CsrMatrix<浮点数>	创建 −csr<T>	armpl_spmat_create_csr_d armpl_spmat_create_csr_s

在成功移植 *Morpheus* 中的 ArmPL 实现后，按照第四节所述的相同原则，继续移植每种格式对应的独立 SVE 实现，其过程与 ArmPL 的移植方式一致。换言之，由于 SVE 实现中未使用句柄，因此不需要工作区。不过，实现中使用的 SVE内在函数必须按照与 ArmPL 类似的方式进行适配。

B. 新后端

在任何代码中集成新后端都是一项极具挑战性的任务，因为每个新后端都会带来独特的挑战，可能需要投入大量开发工作。在本研究中，我们旨在了解如何在未来版本中支持 FPGA 所涉及的挑战。之所以选择 FPGA，是因为 FPGA 与 GPU 有许多相似之处：从开发者的角度来看，在主机-设备模型中，两者都被视为与主机（CPU）具有不同内存空间的加速器。因此，鉴于*Morpheus* 已经支持 GPU，用于管理异构硬件的高级接口可能无需更改。

Morpheus 对现场可编程门阵列的支持流程如下：

1)开发执行空间：为了保持*Morpheus*的可移植性，可以遵循*Kokkos* [29] 中提供的执行空间概念，从而实现一套通用功能，并使其与现有执行空间的接口保持一致。

完成后，*Morpheus* 将能够发现并获取运行时可用的 FPGA，与其同步，并分派为 FPGA 后端实现的任何算法。请注意，查询底层 FPGA 运行时可能需要使用 OpenCL 的辅助例程。

2) 开发内存空间：与执行空间的创建方式类似，必须为 FPGA 上可用的全局内存开发内存空间。按照 *MemorySpace* 这一概念（该概念在 *Kokkos* 中可用），可以创建两个内存空间，分别表示 DDR 和高带宽内存（第二代高带宽内存，HBM2）；这些内存通常可用于许多 FPGA。创建完成后，用户便能够使用这两个内存空间在目标 FPGA 上高效地分配和释放内存，从而创建 FPGA 上可用的任意容器。

3) 数据管理：在开发出能够获取 FPGA 内存的内存空间后，还必须为每个新的内存空间实现数据管理例程。这些例程包括 复制和镜像 操作，负责在受支持的内存空间之间传输数据，并在指定的内存空间中创建具有相同特性和内存分配大小的新容器。此外，这还支持将数据从主机（CPU）卸载到设备（FPGA），以及执行反向传输。

4) 底层实现：最后一步是为 FPGA 实现*Morpheus* 中提供的算法（包括 SpMV 乘法）。选择 FPGA 执行空间时，将启动此层级的实现。需要注意的是，使用 OpenCL 完成的准备工作以及内核启动均在此层级实现。

上述介绍的前三个步骤与*Morpheus*中的现有后端有许多相似之处。然而，最具挑战性的实现阶段是底层实现，因为FPGA的独特属性正是在这一层面体现出来的。其中一个主要挑战在于设备代码的构建流程存在巨大差异。通常，为FPGA编写的代码会经过编译，生成包含硬件配置的比特流；即使对于稀疏矩阵-向量乘法这样的小型内核，这一过程也可能需要数小时，尤其是在不断扩大的优化方案空间中，例如对大型数组进行分区。与其他体系结构相比，这种可重配置性需要付出相对较长构建流程的代价，包括在FPGA芯片上进行布线和布局，从而导致解决方案耗时更长。虽然在CPU上构建用于管理主机与设备交互的主机代码，相较于整体比特流生成过程而言相对较快，但这意味着无法进行即时编译和链接。此外，*Morpheus* 是一个仅头文件库，实际上是在应用程序层面进行编译的。因此，所有算法都以模板的形式编写，由编译器负责在应用程序层面生成，即在用户的代码中生成。结果是，输入的类型直到最后一刻才具有通用性，这意味着我们无法在此之前生成FPGA比特流。为规避这一问题，可以使用一组潜在类型，在*Morpheus*的构建和安装过程中显式实例化这些实现，从而能够*apriori*生成比特流。

另一个重要挑战是如何在同一厂商或不同厂商生产的 FPGA 设备之间实现可移植性。Klaisoongnoen 等人 [30] 研究了 AMD-Xilinx 与英特尔 FPGA 之间的高层次综合内核移植，描述了从一种 FPGA 架构迁移到另一种架构时遇到的挑战，以及不同厂商工具链之间优化技术的适用性。对于本文介绍的高层次综合 SpMV 内核，从一种 FPGA 架构迁移到另一种架构需要调整连接配置，例如将内核参数映射到内存空间。一个重要的考虑因素是目标 FPGA 卡是否提供 HBM2 内存，或者是否必须改为面向板载 DDR，以及这如何集成到可用高层次综合工具链的具体构建流程中。虽然高层次综合代码通常可以在不同 FPGA 架构之间移植，但特定于厂商的编译指示、主机代码实现以及对主机与内核直接流式传输[2] 等功能的支持各不相同。解决这一问题的一种可能方案是为每个厂商创建不同的后端，并在实现算法和优化时考虑向后兼容性。需要注意的是，通过显式实例化 FPGA 实现，并提供所支持 FPGA 设备范围的信息，我们可以预先生成一组比特流，并将其作为*Morpheus* 发行版的一部分进行分发；之后每次运行时，都可以根据相应的运行时环境加载适当的预构建实现。

聚焦于为多个后端自动生成代码，FPGA 面临的一项关键挑战在于，这类设备的运行方式与传统的基于冯·诺依曼架构的 CPU 架构存在根本差异。AMD-Xilinx 的 Vitis HLS 和英特尔的 Quartus Prime Pro 等高层次综合工具链通常可以根据基于 CPU 的代码为 FPGA 生成可用的比特流，但与经过人工调优、适配了最适合 FPGA 的数据流计算风格的 HLS 代码相比，这些比特流的性能往往显著较低。例如，研究表明，朴素的基于 CPU 的 HLS 内核与经过数据流优化的实现之间，运行时性能差异可能非常大（超过 1000 倍的情况并不少见） [31]。现有 HLS 工具虽能在一定限制下提供可移植性，但 CPU 与 FPGA 架构之间的性能可移植性仍是一项尚待解决的挑战。正如上文所述，我们的方法并非基于自动代码生成，而是允许开发者为 FPGA 编写带有显式优化的自定义代码。此外，用户还可以利用*Morpheus* 的动态格式切换功能进行额外优化；FPGA 后端同样支持这一功能。

[2]英特尔 FPGA 编程指南，通过主机管道与内核直接通信：https://www.intel.com/content/www/us/en/docs/programmable/683846/22-1/direct-communication-with-kernels-via.html

VII. 结果与评估

A. 设置

所有面向 AArch64 CPU 的实验均在 Isambard 的布里斯托尔 HPE Apollo80 分区上进行 [32]。该机柜的 72 个节点各配备一颗富士通 A64FX 处理器，具有 48 个 ARMv8.2 内核和 512 位 SVE，时钟频率为 1.8 GHz，并配有 32 GB HBM2 内存，按 4 个核心组成的内存组进行配置。

每项实验均使用 *GNU 10.2.0* 编译，并采用 -O3 -ffast-math -ftree-vectorize -funroll-loops -mcpu=native 编译器标志。对于分布式实验，还使用了 *OpenMPI 4.1.0* 。

本文报告的 FPGA 运行使用 Xilinx Alveo U280，默认时钟频率为 300MHz。该卡搭载一颗 FPGA 芯片，包含 108 万个查找表、4.5MB 片上 BRAM、30MB 片上 UltraRAM 和 9024 个 DSP 切片。此 PCIe 卡还搭载 8GB 第二代高带宽内存和 32GB DDR DRAM。

该 FPGA 卡安装在 ExCALIBUR H&ES FPGA 测试系统 [3] 中，主机配备一颗 32 核 AMD EPYC 7502 CPU 和 256GB DRAM。所有比特流均使用 Xilinx 的 Vitis 框架 2021.2 版本为 U280 构建。所有报告结果均为十次运行的平均值；FPGA 运行时间包括设备上的执行时间，但不包括设备设置时间和数据传输时间。

B. *AArch64 CPU* 上的 *SpMV* 评估

为了评估 *Morpheus* 中新增的面向 AArch64 CPU 的 SpMV 实现的性能，我们针对每种实现，在 SuiteSparse [7]collection 中提供的 2106 个稀疏矩阵上执行 100 次 SpMV 乘法。每次运行均按照第七节 A 部分所述，在富士通 A64FX 处理器上以串行方式执行。这些实现分为三个版本，如表 II 所示，同时还给出了简要描述以及每个版本支持的格式。

表 II：Morpheus 中可用的各个基于 CPU 的 SpMV 实现版本，以及每个版本支持的格式。

版本	描述	COO	CSR	DIA
普通	原始实现 未进行任何 Arm 优化	✓	✓	✓
ARMPL	使用 ArmPL 的实现	✓	✓	×
SVE	使用 SVE 扩展	✓	✓	✓

如图 3 所示，每个版本的最佳格式分布存在显著差异。对于 SuiteSparse 集合中的大多数矩阵而言，最佳格式是压缩稀疏行格式（CSR），这验证了其作为最常用存储格式的地位。然而，在 普通 版本和 *SVE* 版本中，分别有近 20% 和 40% 的矩阵使用坐标格式（COO）时效果更佳。有趣的是，

[3]ExCALIBUR H&ES FPGA 测试平台，现场可编程门阵列（FPGA）用于加速科学与数据科学代码：https://fpga.epcc. ed.ac.uk/

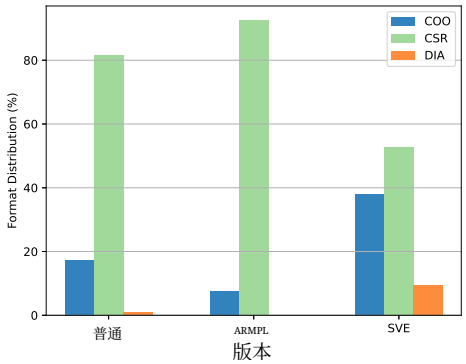
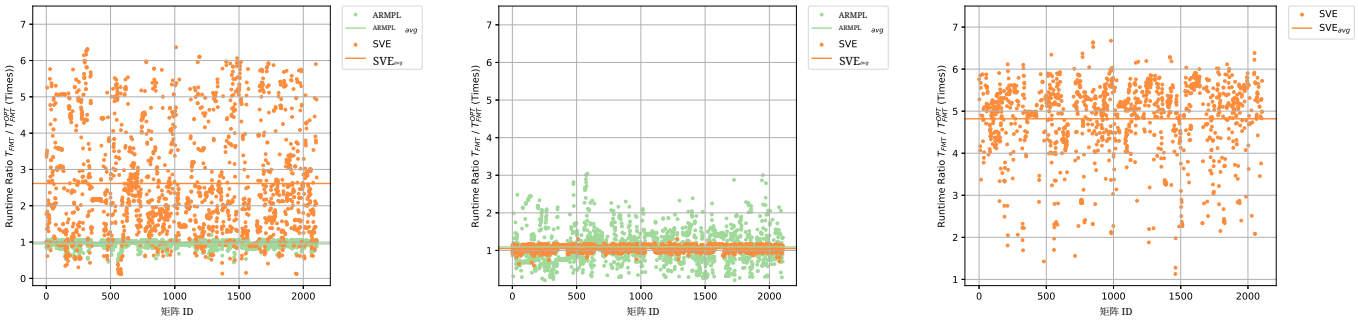


图 3：在 A64FX 上，对 SuiteSparse 集合中 2100 多个稀疏矩阵进行串行稀疏矩阵-向量乘法运算时最优格式的分布。图中展示了该算法各个版本的分布情况。

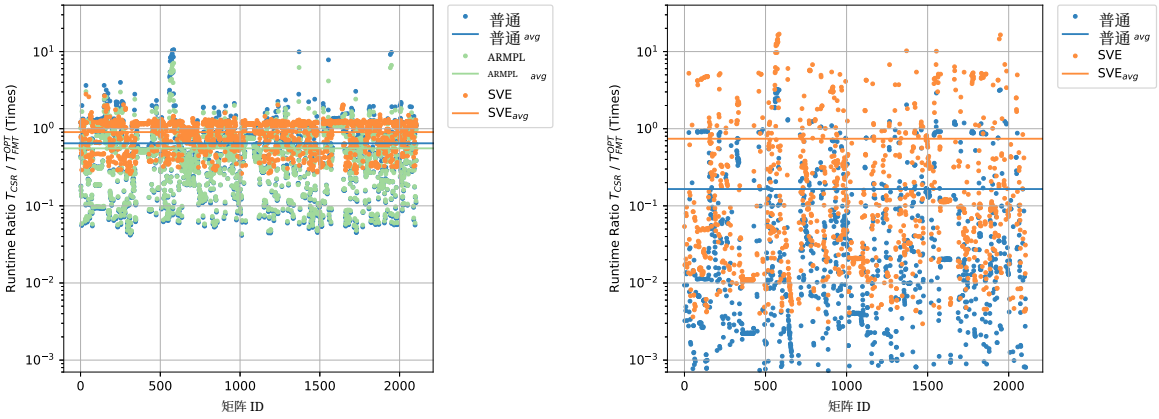
尽管对 普通 版本而言，对角线存储格式几乎毫无用处，但 *SVE* 版本执行的向量化使对角线存储格式成为 10% 矩阵的最佳格式。这表明，普通版本中编译器针对对角线存储格式执行的向量化，可能不如 *SVE* 版本中使用自定义 SVE 扩展有效。这里的主要结论是，对于相同的硬件、操作和矩阵集合，在大多数情况下，压缩稀疏行格式（CSR）都能提供最佳性能；不过，如果采用不同的实现方式或应用不同的优化，最佳格式的分布可能会发生显著变化。

图 4 展示了 A64FX 处理器上，SuiteSparse 集合中 2100 多个稀疏矩阵的稀疏矩阵-向量乘法单核性能。对于每种格式，均将普通版 SpMV 的运行时与采用相同格式的各优化版 SpMV（*ARMPL* 和 *SVE*）的运行时进行比较。对于 COO（图 4a），*ARMPL* SpMV 实现的性能与普通 COO SpMV 实现相当，而 *SVE* 实现始终优于前者，平均加速比分别为 $1\times$ 和 $3.6\times$。*SVE* 版本取得的性能提升，可归因于 SpMV 算法实现中所作的假设，这些假设使我们能够利用不同的内置指令。例如，假设矩阵已排序（*Morpheus* 会在执行任何稀疏矩阵-向量乘法运算前确保矩阵已排序），便可以使用树形归约代替传统的从左到右归约，将结果累加到输出向量 y 中。值得强调的是，尽管对于 COO 中的大多数矩阵，*SVE* 版本的性能显著优于 *Plain* 版本，但仍有相当数量的矩阵性能明显较差。对于极稀疏且结构无规则的矩阵，*SVE* 版本似乎在向量化过程中引入了更多开销，实际降低了 SpMV 的性能。对于 CSR（图 4b），*ARMPL* 和*SVE* 版本的平均运行时性能均与*Plain*相当。有趣的是，对于大量矩阵



(a) 坐标格式　(b) 压缩稀疏行格式　（c）对角线格式

图 4：在 A64FX 上，对 SuiteSparse 集合中的 2100 个稀疏矩阵进行 SpMV 乘法的串行性能。对于每种格式，将普通 *Morpheus* SpMV 的原始性能与优化后的（ARM）SpMV 进行比较。优化版本包括*ArmPL* 和 *SVE* 实现，考察的格式包括坐标格式、压缩稀疏行格式和对角线格式。比值大于 1 表示性能优于采用相同格式的原始实现。直线表示各版本在所有矩阵上的平均加速比。



（a）坐标格式　（b）对角线格式

图 5：在 A64FX 上，对 SuiteSparse 集合中的 2100 个稀疏矩阵进行 SpMV 乘法的串行性能。对于每种格式，将普通 *Morpheus* 压缩稀疏行格式 SpMV 的原始性能与优化后的（ARM）SpMV 进行比较。优化版本包括*ArmPL* 和 *SVE* 实现，考察的格式包括坐标格式和对角线格式。比值大于 1 表示性能优于采用原始压缩稀疏行格式实现时的性能。直线表示各版本在所有矩阵上的平均加速比。

ARMPL 版本相比*Plain*的加速比超过 $1\times$ ，最高可达 $3\times$。与此同时，对于大量矩阵，其性能似乎明显较差，而 *SVE* 版本的性能表现则更稳定。利用 SVE 内置函数带来的最大收益出现在 DIA（图 4c）中：*SVE* 版本相比 *Plain* 实现的平均加速比为 $\approx 5\times$ 。*SVE* 实现对集合中的所有矩阵都优于*Plain*实现，这表明编译器难以对*Plain*版本进行有效向量化。

由于对于大多数矩阵，执行稀疏矩阵-向量乘法运算的最佳格式是压缩稀疏行格式，因此在图 5 中，我们将三种版本（普通、*ARMPL* 和*SVE*）下坐标格式和对角线格式的运行时性能，与相同配置下普通 版本的压缩稀疏行格式运行时性能进行比较与之前相同。对于坐标格式（图 5a），平均而言，普通版本和*ARMPL* 版本的表现大致相同，且不如普通 CSR实现。另一方面，*SVE* 版本所采用的优化，使其平均性能达到普通CSR实现的水平。值得注意的是，对于大多数矩阵，无论使用哪种版本，坐标格式相较于压缩稀疏行格式都会导致显著的性能下降。不过，对于少数矩阵，这三个版本都能带来明显的加速比，最高加速比可达 $10\times$。对角线格式（图 5b）也呈现出类似的情况。显然，对角线存储格式只适用于少数具有特定特征的矩阵。不过，对于这些矩阵，采用 SVE 内在函数进行性能优化可以显著提升性能；与压缩稀疏行格式相比，最高加速比可达 $\approx 20\times$ 。

值得指出的是，尽管原本预计*ARMPL* 版本在坐标格式和压缩稀疏行格式下会达到最佳表现，但其平均表现与对应的普通 版本不相上下。不过，对于大量矩阵，压缩稀疏行格式的*ARMPL*实现表现最佳。此外，采用 SVE 内在函数对对角线格式产生了显著影响，使其适用范围扩大了一个数量级，并且与对应的普通 实现相比，持续带来明显的加速比。这项实验清楚地表明，正如没有一种格式能够在不同稀疏模式下始终表现最佳，也没有一种实现能够做到这一点。因此，这些发现促使我们扩展*Morpheus* ，使其也支持一种高效机制，以便在运行时选择最佳实现。

C. FPGA 上稀疏矩阵-向量乘法的评估

在评估我们针对现场可编程门阵列的稀疏矩阵-向量乘法实现时，我们对 SuiteSparse 集合中的每个矩阵，分别运行了三种稀疏矩阵-向量乘法内核，各迭代 10 次。实验设置的更多细节见第七节 A。图 7 展示了在串行模式下，对 2100 多个稀疏矩阵执行稀疏矩阵-向量乘法运算时，最佳格式的分布。就性能或最短运行时间而言，压缩稀疏行格式占主导地位：与坐标格式和对角线格式相比，在 80% 以上的矩阵上，它都能取得最佳性能。SuiteSparse 矩阵中，次优格式是坐标格式，占比超过 10%；对角线格式在 5% 以上的矩阵上仍是最佳选择。值得强调的是，在 A64FX 处理器上，普通 稀疏矩阵-向量乘法版本也呈现出类似的分布，如图 3 所示。

图 6a 和 6b 展示了我们的 FPGA 原型采用坐标格式和对角线格式时，相对于压缩稀疏行格式的串行性能。这些结果证实了最佳格式比较所得的结论：对于大多数矩阵，采用压缩稀疏行格式的稀疏矩阵-向量乘法算法在 FPGA 上优于坐标格式和对角线格式的实现。请注意，尽管坐标格式和对角线格式相对于压缩稀疏行格式的平均加速比都远低于 1，但在极少数矩阵上，性能确有提升（例如，有一个案例中，坐标格式的加速比为 $2\times$）。进一步优化内核，使其充分利用硬件特性，有望让最佳格式的分布更加多样化。不过，目前坐标格式和对角线格式相对于压缩稀疏行格式均未实现加速，因此我们可以得出结论：压缩稀疏行格式在传统体系结构上的优势，在我们面向 FPGA 的基线版本中同样成立。

在图 6c 中，将第五节所述的坐标格式 SpMV内核优化与朴素版本进行了比较。我们观察到，对于大多数矩阵，使用第二代高带宽内存带来了小幅性能提升，但将高带宽内存与片上 BRAM 结合使用并未产生进一步的明显效果。如图 2 所示，归约运算对于 FPGA 而言并非最优，因为迭代之间的浮点累加需要超过一个时钟周期，从而在循环迭代之间引入空间依赖，并因此使启动间隔大于一个，这意味着周期被浪费了。在 FPGA 上，通常的做法是通过对独立迭代进行分块归约来优化这一点，图 6c 中以*REDUCE*表示了这种方法的性能数据。然而可以看出，在本例中这种方法并无益处，这是因为行索引条件判断意味着不满足该条件的周期本来就无所事事，因此抵消了归约优化所增加的复杂性。不过值得强调的是，在其中一个矩阵上，它在三种优化中实现了最高的最大加速比；我们观察到，随着行数和非零元素数 NNZ 增加，其开销也会增加，因此归约优化对小矩阵效果更好。这一结果再次证明，这次是在 FPGA 上，同一种稀疏矩阵-向量乘法运算采用多种实现的重要性，以及根据输入的稀疏模式动态切换至最佳格式的能力。

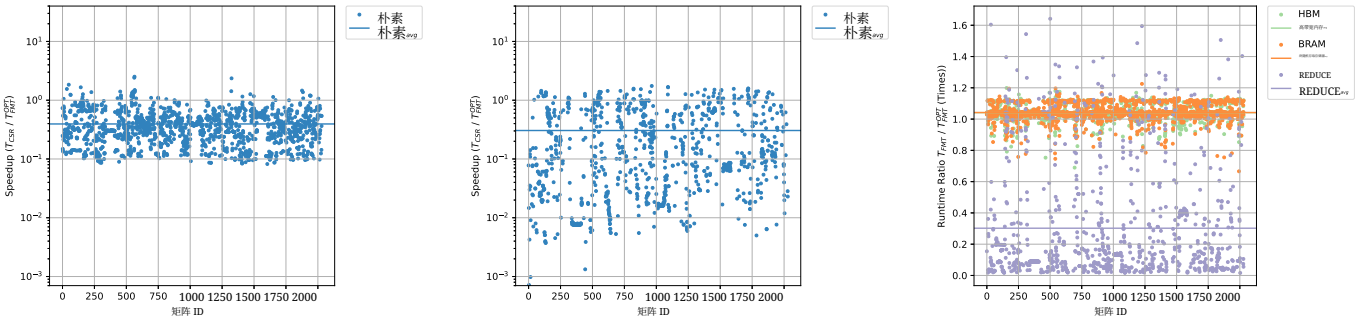
D. HPCG 实验

HPCG 基准测试在规则的三维网格上求解泊松微分方程，并使用 27 点模板进行离散化。它采用带有对称高斯-赛德尔 [26]预条件器的预条件共轭梯度（PCG）算法，并包含以下计算：稀疏矩阵-向量乘法（SpMV）；向量更新；全局点积；局部对称高斯-赛德尔平滑器（包括稀疏三角求解）；以及多重网格（MG）预条件求解器。HPCG 的性能瓶颈源于迭代求解器每一步都会执行的稀疏操作，即 SpMV 和高斯 - 赛德尔平滑器。

HPCG 是一种广泛认可且广为人知的基准测试，用于衡量高性能计算系统的性能。因此，存在多个针对特定厂商的实现（ [10]，[33], [34]）。在此前的工作中，我们也实现了支持*Morpheus*的 HPCG [9]。该基准测试按以下阶段进行：

1) 问题设置：通过创建几何结构和线性系统来构造合成问题。2) 参考计时：测量运行稀疏矩阵-向量乘法和多重网格（MG）参考实现所需的时间，以及参考共轭梯度（CG）求解器的求解时间。3) 问题优化设置：配置将在优化问题中使用的用户定义数据结构。4) 验证与校验：检查优化问题是否返回了预期结果。5) 优化问题计时：测量优化版 CG 求解器的求解时间。

在以下实验中，我们对以下实现的性能进行基准测试：1）启用了*Morpheus*的 HPCG，以及新增的*ARMPL* 和 *SVE* 版本的 SpMV；2）供应商（Arm）提供的 HPCG 实现与原始 HPCG 的对比。我们重点关注第五阶段。不过，由于本研究关注的是 SpMV 乘法，因此禁用了所有实现中的预条件器。



(a) 普通坐标格式稀疏矩阵-向量乘法的性能，与CSR实现相比。

(b) 普通对角线格式稀疏矩阵-向量乘法的性能，与CSR实现

(c) 优化后的坐标格式 SpMV 版本相对于普通坐标格式版本。

图 6：在 AlveoU280 上，对 SuiteSparse 集合中的 2100 个稀疏矩阵进行 SpMV 乘法的串行性能。比率高于 1 表示相对于参考实现所达到的性能有所提升。直线表示各版本在所有矩阵上的平均加速比。

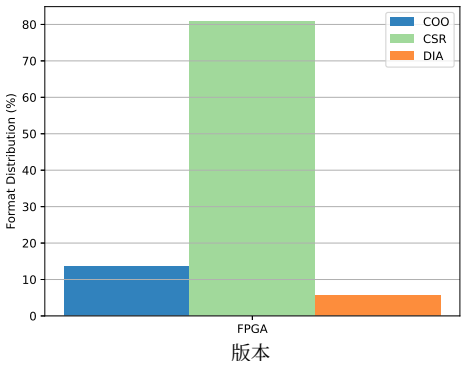


图 7：在 Alveo U280 上，对 SuiteSparse 集合中的 2100 多个稀疏矩阵进行串行 SpMV 乘法时的最优格式分布。图中展示了各算法版本对应的分布。

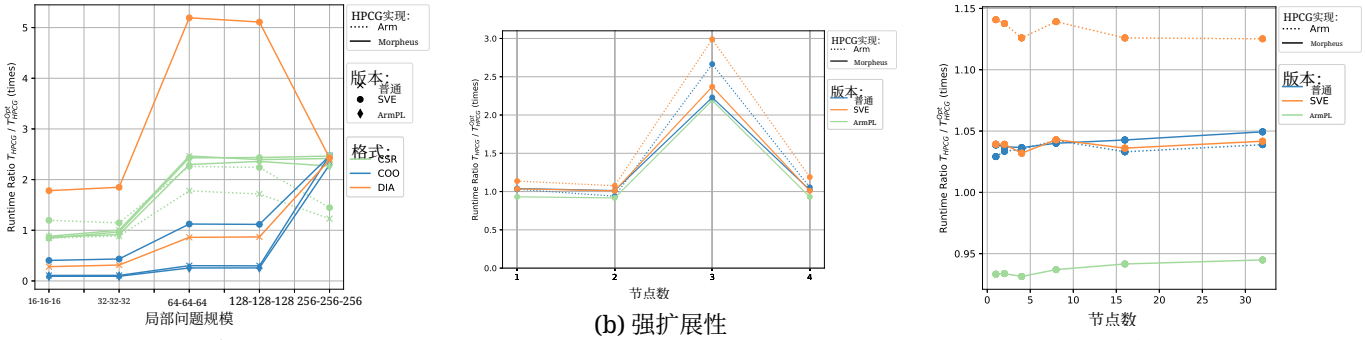
实验按照第七节 A 部分所述，在 A64FX 处理器上进行配置。

图 8a 展示了在不同问题规模下，两个 HPCG 实现（*Morpheus*和 *Arm*）的单核 SpMV 运行时性能，并与原始 HPCG 进行比较。对于启用 *Morpheus* 的实现中的每种格式，我们测量了 *Plain*、*ARMPL*和 *SVE* 版本的 SpMV 乘法例程的运行时间。同样，对于 Arm 实现，我们测量了使用（*SVE*）和不使用（*Plain*）SVE 内在函数时 SpMV 的运行时间。

HPCG 中的系统矩阵使用 FDM 生成。因此，该矩阵具有高度规则的结构，非零元素分布在对角线附近。因此可以预期，与其他格式相比，对角线存储格式的性能应当最优，图 8a 证实了这一假设。两个版本的 DIA 稀疏矩阵-向量乘法，即 普通版和 *SVE*版之间的最佳性能差异，与图 5b 中观察到的平均性能一致，相对于参考 HPCG 的最大加速比为 $5\times$。需要注意的是，对于较小的问题规模，两个版本的性能都会受到零填充所引入的额外操作影响。Arm HPCG 的 *SVE* 版本与使用 CSR 的 *Morpheus*启用版 HPCG 的性能非常接近。然而，在问题规模为 64^3 时，不支持 *SVE* 的 Arm HPCG 版本性能会下降。这可以归因于 Arm HPCG 中的矩阵经过了重新排序，以及缺少 SVE 扩展。有趣的是，当问题规模为 256^3 时，之前性能优于参考实现的每个实现现在要么性能下降，要么保持相当的水平。不过，使用坐标格式的所有 *Morpheus*启用版 HPCG，以及使用对角线存储格式的 普通版，现在都出现了性能提升，其中使用坐标格式的 *SVE* 版本取得了所有实现中最优但幅度有限的性能。该结果进一步说明，有必要针对同一格式的不同稀疏矩阵-向量乘法实现进行运行时切换。

Morpheus 和 Arm 版本的分布式 HPCG 性能均以原始 HPCG 实现为基准进行测量。强扩展性实验选择的全局问题规模为 $192 \times 256 \times 192$，弱扩展性实验选择的局部问题规模为$48 \times 64 \times 64$。对于分布式实现，由于矩阵右侧添加了远程元素，因此每个进程上的矩阵稀疏模式与串行情形不同。矩阵起初具有结构化形式，但其远程部分高度非结构化。因此，在启用*Morpheus*的 HPCG 中，我们将该矩阵在物理上拆分为局部 部分和远程 部分，以便为两者分别选择不同的存储格式。具体做法是使用一种先运行后调优的自动调优器，为每个进程找出最优格式。我们每种实现版本在各进程上选择的格式见表 III。请注意，在*SVE* 版本中，强扩展性和弱扩展性实验选择的最优格式均为对角线格式和坐标格式。

图 8b 和图 8c 展示了启用 *Morpheus* 和 Arm 的 HPCG 实现相对于基准 HPCG 的各版本稀疏矩阵-向量乘法扩展性能。请注意，对于这两种



(a) 串行性能　(b) 强扩展性　(c) 弱扩展性

图 8：Morpheus 和 Arm 加速的 HPCG 实现的性能。性能以参考 HPCG 相对于 A64FX 上各最优 HPCG 实现及版本的 SpMV 运行时比率进行衡量。对于串行版本，高性能共轭梯度（HPCG）针对一组问题规模运行。对于分布式的 Morpheus 加速的 HPCG，使用先运行后调优的自动调优器，以确定每个进程所用的最优格式。比率大于 1 表示性能优于使用原始 HPCG 时达到的性能。

表 III：强扩展与弱扩展下，Morpheus 加速的 HPCG 在不同进程中各版本所用的最优格式。

版本	本地格式	远程格式
普通	CSR	CSR
ARMPL	CSR	CSR
SVE	DIA	COO

强扩展与弱扩展，启用 *Morpheus* 的 HPCG 性能都与最优的 Arm 启用 *SVE* 版本 HPCG 相近。在图 8b 中，3 个节点上的性能提升源于系统矩阵规模恰好在向量化带来的收益与相关开销之间取得了平衡。此外，图 8c 的弱扩展性结果虽然相比原始 HPCG 仅有小幅提升，但仍表明我们的改进达到了最优 Arm 实现的性能。不过，如果局部问题规模更接近我们此前观察到性能提升的范围（即图 8a 所示的 64^3 到 128^3 之间），预期会有更显著的加速；但由于内存限制，无法在分布式层面进行此类运行。

VIII. 相关工作

针对稀疏计算优化的研究工作大致分为两类，即：（i）创建能够更好地捕捉特定体系结构或稀疏性模式特征的新型存储格式，例如压缩稀疏行格式（CSR）5 [3]和 Sell-C-σ [5]；以及（ii）使用 Morpheus-Oracle [35] 和 SMAT [36]等自动调优器，自动确定计算所使用的最佳格式。

近期有一些研究探讨了不同稀疏格式在 AArch64 目标平台上的性能。尤其是在 A64FX 处理器上，研究表明，SELL-C-σ 矩阵存储的性能和内存带宽饱和度优于标准压缩稀疏行格式，在大型、受内存带宽限制的稀疏矩阵-向量乘法数据集上，其性能可达到与英伟达 V100 图形处理器相当的水平 [37]。此外，研究人员还为基于 A64FX 的富岳超级计算机提出了一种经过优化的压缩稀疏行格式变体——“基于位图的压缩稀疏行（BCSR）”。与经典压缩稀疏行格式相比，它能更高效地提取边信息，且内存占用更小，使富岳在 2020 [14] 年 11 月的 Graph500 基准测试中跻身第一。压缩稀疏行格式和 ELLPACK 格式（ELL）的其他变体，即对齐压缩稀疏行（ACSR）和对齐 ELL（AELL），也已被提出，并使用 Neon 指令在 AArch64 目标平台上实现 [38]。

对于稀疏矩阵乘法，FPGA 厂商（如英特尔）在英特尔 oneAPI 数学核心函数库的稀疏 BLAS 库中实现了坐标格式、压缩稀疏行格式和对角线存储格式等 [39]，；而赛灵思则在其 Vitis 稀疏库中实现了压缩稀疏列格式 [40]，并在其通用矩阵运算（GEMX）引擎库中实现了坐标格式 [41] 。该引擎库提供了用于在 FPGA 上构建矩阵运算加速器的构件。其他研究则着重于在高层次综合中为 SpMV 乘法实现改进的压缩稀疏行（MCSR）格式 [42]，或设计定制的稀疏矩阵格式，以利用新一代 FPGA 上可用的第二代高带宽内存 [43]。

九、结论与后续工作

本文阐明了在两种新兴体系结构 AArch64 CPU 和 FPGA 上，为 COO、CSR 和 DIA 三种稀疏存储格式实现稀疏矩阵-向量乘法内核原型所面临的挑战。通过针对各自的体系结构优化这三种内核，我们突显了不同方法的性能优势，并表明 Morpheus 的实现具有竞争力。此外，我们还介绍了这些原型在 Morpheus 库中的潜在集成目标。虽然我们的结果证明，尤其是在配备 SVE 的 AArch64 CPU 上，相较于 ARMPL，可以实现高性能，但现场可编程门阵列等加速器

与不采用主机-设备模型的体系结构相比，现场可编程门阵列表现出更大的性能可移植性差距。

关于未来工作，将 FPGA 原型作为 Morpheus 的新型后端进行完整集成颇具价值，但由于动态内存和数据类型支持受限，这将需要进一步完善内存管理抽象、构建流程集成以及智能容器/层转换。鉴于英特尔和 AMD-Xilinx 等厂商已经针对特定架构实现了 HPCG，因此可以将这些实现与 Morpheus-HPCG 版本进行评估，尤其是 AMD-Xilinx 在 FPGA 上与 HPCG 问题规模生成紧密相关的 CSR 实现。此外，这些存储格式的 FPGA 原型还可受益于在最新一代 AMD-Xilinx FPGA（即 Versal ACAP）上的实现。

虽然本文重点关注 Morpheus 支持的三种核心稀疏矩阵存储格式，即坐标格式、压缩稀疏行格式和对角线格式，但还存在大量其他存储格式，例如 ELL、HYB 或 HDC，它们被广泛使用；Morpheus 在 AArch64 CPU 和 FPGA 上的扩展可以进一步使高性能计算用户受益。

在理论评估方面，未来的工作包括应用屋顶线模型，以了解和验证我们实现的性能，并将目标系统上的理论性能与实际达到的性能进行比较。

此外，使用 SVE 内在函数实现当前格式的多线程（OpenMP）版本，并在其他 Arm系统上进行基准测试，也值得关注。最后，扩展 *Morpheus*，使其支持动态选择和分派机制，能够根据稀疏模式选取最优算法，这可能会带来益处。

致谢

本研究是 EPSRC 项目 ASiMoV（EP/S005072/1）的一部分。我们使用了由 GW4 和英国气象局运营、由 EPSRC（EP/T022078/1）资助的 Isambard 2英国国家二级高性能计算服务（http://gw4.ac.uk/isambard）。我们还感谢 ExCALIBUR H&ES FPGA 测试平台和 AMD 赛灵思 HACC计划为本研究提供了计算资源访问权限。

参考文献

[1] S. Filippone、V. Cardellini、D. Barbieri 和 A. Fanfarillo，“通用图形处理器上的稀疏矩阵-向量乘法”，*ACM*数学软件汇刊，第 43 卷，第 4 期，2017 年 1 月。 [在线]。获取地址：https: //doi.org/ 10 1145 3017994 。/

[2] Y. Zhao 、 J. Li 、 C. Liao 和 X. Shen，“ 弥合深度学习与稀疏矩阵格式选择之间的差距”，载于第 *23* 届 *ACM SIGPLAN* 并行编程原理与实践研讨会论文集，PPoPP ’18 系列。美国纽约州纽约：计算机协会，2018 年，第 94–108 页。 [在线]。获取地址：https://doi.org/ 10 1145 3178487 3178495 。/。

[3] W. Liu 和 B. Vinter，“CSR：一种适用于跨平台稀疏矩阵-向量乘法的高效存储格式”，载于第 *29* 届 *ACM* 国际超级计算会议论文集，ICS ’15 系列。美国纽约州纽约：计算机协会，2015 年，第 339–350 页。 [在线]。获取地址：https://doi.org/10.1145/2751205.2751209

[4] E. Coronado-Barrientos、M. Antonioletti 和 A. Garcia-Loureiro，《一种用于高效 SpMV 运算的新型 AXT 格式：使用 AVX-512 指令和 CUDA》，《工程软件进展》，第 156 卷，文章编号 102997，2021 年。 [在线]。获取地址：https://www.sciencedirect.com/science/article/pii/S 0965997821000260

[5] M. Kreutzer、G. Hager、G. Wellein、H. Fehske 和 A. Bishop，《一种统一的稀疏矩阵数据格式，可在配备宽 SIMD 单元的现代处理器上高效执行通用稀疏矩阵-向量乘法》，《SIAM 科学计算杂志》，第 36 卷，2013 年 7 月。[6] C. Stylianou 和 M. Weiland，《利用动态稀疏矩阵实现具有性能可移植性的线性代数运算》，载于《2022 年 IEEE/ACM 高性能计算性能、可移植性与生产力国际研讨会（P3HPC）》。美国加利福尼亚州洛斯阿拉米托斯：IEEE 计算机协会，2022 年 11 月，第 47–57 页。 [在线]。获取地址：https://doi.ieeecomputersociety. org/ 10 1109 3 56579 2022 00010 。/P

[7] HPC。T. A. Davis 和 Y. Hu，《佛罗里达大学稀疏矩阵集合》，《ACM数学软件汇刊》，第 38 卷，第 1 期，2011 年 12 月。 [ ] 10 1145 2049662 2049663 在线。获取地址：

[8] https://doi.org/。/。J. Dongarra、M. A. Heroux 和 P. Luszczek，《高性能共轭梯度基准测试：一种用于高性能计算系统排名的新指标》，《国际高性能计算应用杂志》，第 30 卷，第 1 期，第 3–10 页，2016 年。 [ ] 10 1177 1094342015593158 在线。获取地址：

[9] https://doi.org/。/。《Morpheus-HPCG 基准测试》，Morpheus，11 月。在线]。获取地址：https://github.com/morpheus-org/morpheus-hpcg [10] 《面向 Arm 的 HPCG》，Arm 软件，2022 年 11 月。 [在线]。获取地址：https://github.com/ARM-software/HPCG for Arm [11] S. Balay、S. Abhyankar、M. F. Adams、S. Benson、J. Brown、P. Brune、K. Buschelman、E. M. Constantinescu、L. Dalcin、A. Dener、V. Eijkhout、J. Faibussowitsch、W. D. Gropp、V. Hapla、T. Isaac、P. Jolivet、D. Karpeev、D. Kaushik、M. G. Knepley、F. Kong、S. Kruger、D. A. May、L. C. McInnes、R. T. Mills、L. Mitchell、T. Munson、J. E. Roman、K. Rupp、P. Sanan、J. Sarich、B. F. S mith、S. Zampini、H. Zhang、H. Zhang 和 J. Zhang，《PETSc 网页》，https://petsc.org/，2023 年。 [在线]。获取地址：https://petsc.org/ [12] H. Anzt、T. Cojean、G. Flegar、F. G¨obel、T. Gr¨utzmacher、P. Nayak、T. Ribizel、Y. M. Tsai 和 E. S. Quintana-Ort´ı，《Ginkgo：面向高性能计算的现代线性算子代数框架》，《ACM 数学软件汇刊》，第 48 卷，第 1 期，第 2:1–2 33， 2022 [ ] 10 1145 3480935 页，2 月。在线。

[13] 获取地址：https://doi.org/。/。A. Jackson、M. Weiland、N. Brown、A. Turner 和 M. Parsons，《A64FX 上的应用研究》，载于《2020 年 IEEE 国际集群计算会议（CLUSTER）》。日本神户：IEEE，2020 年 9 月，第 549–558 页。[14] M. Nakao、K. Ueno、K. Fujisawa、Y. Kodama 和 M. Sato，《超级计算机富岳在 Graph500 基准测试广度优先搜索中的性能》，载于《高性能计算》，B. L. Chamberlain、A.-L. Varbanescu、H. Ltaief 和 P. Luszczek 编。瑞士沙姆：Springer International Publishing，2021 年，第 12728 卷，第 372–390 页。[15] R. Jesus、T. Oliveira e Silva 和 M. Weiland，《在基于 Arm 的超级计算机上向量化并分发 NTT 以计算 Goldbach 分拆数》，2021 年 5 月，Cray 用户组（CUG）2021。[16]Masoud Koleini，《Graviton3 的机器学习性能优于 x86》，2022 年 10 月。[在线]。获取地址：https://community.arm.com/arm-community-blogs/b/infrastructure-solutions-blog/posts/xgboost-lightgbm-aws-graviton3[17] 《Arm 架构参考手册补编：可扩展向量扩展》，Arm Limited，DDI 0584B.a，2021 年 5 月。[18] N. Stephens、S. Biles、M. Boettcher、J. Eapen、M. Eyole、G. Gabrielli、M. Horsnell、G. Magklis、A. Martinez、N. Premillieu、A. Reid、A. Rico 和 P. Walker，《Arm 可扩展向量扩展》，《IEEE Micro》，第 37 卷，第 2 期，第 26–39 页，2017 年。[19] 《Arm 性能库参考指南》，Arm Limited，版本 2210-00，2022 年 9 月。[20] N. Brown，《将不可压缩流矩阵组装移植到 FPGA，以加速 HPC 工程模拟》，载于《2021 年 IEEE/ACM 异构高性能可重构计算国际研讨会（H2RC）》。IEEE，2021 年，第 9–20 页。[21] N. Brown，《利用赛灵思和英特尔 FPGA 加速大气建模中的平流计算》，载于《2021 年 IEEE 国际集群计算会议（CLUSTER）》。IEEE，2021 年，第 767–774 页。[22] N. Brown、M. Klaisoongnoen 和 O. Brown，《采用数据流技术优化 FPGA 信用违约互换引擎》，载于《2021 年 IEEE 国际集群计算会议（CLUSTER）。

美国：电气与电子工程师协会（IEEE），2021 年 10 月。[23] M. Klaisoongnoen、N. Brown 和 O. Brown，《我渴望速度：利用最新一代 FPGA 为高频交易提供新能力》，载于《第 11 届高效加速器与可重构技术国际研讨会论文集》，丛书 HEART '21。美国纽约州纽约：计算机协会（ACM），2021 年。[24] M. Klaisoongnoen、B. Nick 和 O. Brown，《低功耗期权希腊值：利用 FPGA 进行以效率为导向的市场风险分析》，载于《HEART2022：高效加速器与可重构技术国际研讨会》。美国：计算机协会（ACM），2022 年 6 月。[25] N. Brown，《探索 Versal AI 引擎，以加速基于模板的大气平流模拟》，载于《第 31 届 ACM/SIGDA 现场可编程门阵列国际研讨会（FPGA）》，2023 年。[26] Y. Saad，《稀疏线性系统迭代方法》，第 2 版，丛书《应用数学其他著作》。工业与应用数学学会，2003 年。 [在线]。获取地址：https://www-users.cs. umn.edu/~saad/IterMethBook 2ndEd.pdf [27] 《Arm C 语言扩展》，Arm Limited，版本 2022Q4，2022 年 11 月。[28] E. Gamma、R. Helm、R. Johnson 和 J. Vlissides，《设计模式：可复用面向对象软件的基础》。美国：Addison-Wesley Longman 出版公司，1995 年。[29] H. C. Edwards、C. R. Trott 和 D. Sunderland，《Kokkos：通过多态内存访问模式实现多核性能可移植性》，《并行与分布式计算杂志》，第 74 卷，第 12 期，第 3202–3216 页，2014 年，专题：高性能计算的领域特定语言与高级框架。 [在线]。获取地址：http://www.sciencedirect.com/science/article/pii/S0743731514001257 [30] M. Klaisoongnoen、N. Brown 和 O. Brown，《快速且节能的导数风险分析：在赛灵思和英特尔 FPGA 上流式处理期权希腊值》，IEEE/ACM 异构高性能可重构计算国际研讨会（H2RC），2022 年 12 月。[31] N. Brown 和 D. Dolman，《一切皆关乎数据移动：优化 FPGA 数据访问以提升性能》，载于《2019 年 IEEE/ACM 异构高性能可重构计算国际研讨会（H2RC）》，2019 年，第 1–10 页。[32] GW4 和英国气象局，《Isambard 2英国国家二级高性能计算服务》，2022 年。 [在线]。获取地址：http://gw4.ac.uk/isambard/ [33] E. Phillips 和 M. Fatica，《高性能共轭梯度基准测试的 CUDA 实现》，载于《高性能计算系统：性能建模、基准测试与仿真》，S. A. Jarvis、S. A. Wright 和 S. D. Hammond 编。瑞士沙姆：Springer International Publishing，2015 年，第 68–84 页。[34] A. Zeni、K. O'Brien、M. Blott 和 M. D. Santambrogio，《在可重构硬件上优化实现 HPCG 基准测试》，载于《欧洲并行处理会议》。Springer，2021 年，第 616–630 页。[35] C. Stylianou 和 M. Weiland，《通过自动格式选择与机器学习优化稀疏线性代数》，载于《第十八届自动性能调优国际研讨会（iWAPT2023）》。美国佛罗里达州圣彼得堡：IEEE 计算机协会，即将发表。[36] J. Li、G. Tan、M. Chen 和 N. Sun，《SMAT：一种面向稀疏矩阵-向量乘法的输入自适应自动调优器》，载于《第 34 届 ACM SIGPLAN 编程语言设计与实现会议论文集》，丛书 PLDI '13。美国纽约州纽约：计算机协会，2013 年，第 117–126 页。 [在线]。获取地址：https://doi.org/10 1145 2491956 2462181 。/。 [37] C. Alappat、N. Meyer、J. Laukemann、T. Gruber、G. Hager、G. Wellein 和 T. Wettig，《A64FX 上稀疏矩阵-向量乘法与格点量子色动力学的执行-缓存-内存建模和性能调优》，《并发与计算：实践与经验》，第 34 卷，第 20 期，2022 年 9 月。[38] Y. Zhang、W. Yang、K. Li、D. Tang 和 K. Li，《基于对齐存储格式的 ARM 处理器上 SpMV 性能分析与优化》，《并行与分布式计算杂志》，第 158 卷，第 126–137 页，2021 年 12 月。[39] 英特尔。《稀疏矩阵存储格式》。[在线]。获取地址：https://www.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top/appendix-a-linear-solvers-basics/sparse-matrix-storage-formats.html [40] 赛灵思。（2019）稀疏库，Vitis 加速库。 [在线]。链接：https://github.com/Xilinx/Vitis Libraries/tree/main/sparse [41] 赛灵思。通用矩阵运算（GEMX）。 [在线]。链接：https://github.com/Xilinx/gemx [42] M. Hosseinabady 和 J. L. Nunez-Yanez，"使用高级综合实现稀疏矩阵-向量乘法的流式数据流引擎"，*IEEE* 集成电路与系统计算机辅助设计汇刊，第 39 卷，第 6 期，第 1272–1285 页，2020 年。[43] Y. Du、Y. Hu、Z. Zhou 和 Z. Zhang，"使用高层次综合在配备高带宽内存的 FPGA 上实现高性能稀疏线性代数：以稀疏矩阵-向量乘法为例"，载于*2022*年 *ACM/SIGDA* 国际现场可编程门阵列研讨会论文集，系列：FPGA '22。美国纽约：计算机协会（ACM），2022年，第 54–64 页。 [在线]。链接：https://doi.org/10.1145/3490422.3502368