稀疏流语义寄存器：加速通用稀疏线性代数的轻量级指令集架构扩展

Paul Scheffler，*IEEE* 学生会员，Florian Zaruba、Fabian Schuiki、
Torsten Hoefler，*IEEE* 会士，Luca Benini，*IEEE* 会士

arXiv:2305.05559v2 [cs.AR] 2 Oct 2023

摘要—稀疏线性代数在许多应用领域都至关重要，但无论是在软件还是硬件中，要高效处理都颇具挑战；单侧和双侧操作数稀疏性通常需要采用不同的方法。在本研究中，我们增强了现有的内存流式 RISC-V 指令集架构扩展，通过加速流式间接寻址、交集和并集等底层操作，支持在压缩稀疏行（CSR）和压缩稀疏纤维（CSF）等常见稀疏张量格式上加速处理单侧和双侧操作数稀疏性。与经过优化的 RISC-V 基线相比，我们的扩展在稀疏-稠密乘法、稀疏-稀疏乘法和稀疏-稀疏加法中分别实现最高 7.0 倍、7.7 倍和 9.8 倍的单核加速比；在稀疏-稠密问题上，浮点运算单元利用率峰值可达 80%。在八核集群上，使用真实世界矩阵进行稀疏-稠密和稀疏-稀疏矩阵-向量乘法，速度最高分别提升 4.9 倍和 5.9 倍，能效最高分别提升 2.9 倍和 3.0 倍。我们还探讨了这些扩展在模板代码和图模式匹配等方面的更多应用。与近期的中央处理器、图形处理器和加速器方案相比，我们的扩展仅带来极小的硬件开销（计算集群面积仅增加 1.8%），同时在数据表示、稀疏度和数据流方面提供更高的灵活性。采用我们扩展的集群执行 CSR 矩阵-向量乘法时，其浮点利用率峰值分别达到近期高度优化的中央处理器和图形处理器稀疏数据结构及库的 9.9 倍和 1.7 倍，即使将片外主存（HBM）和片上互连的延迟及带宽影响考虑在内亦是如此。

索引 特e rms—计算机体系结构、硬件加速、线性代数、稀疏计算、稀疏张量 rs

1 引言

SPARSE 线性代数（LA）对针对高度规则且可向量化工作负载优化的现代处理器架构构成了严峻挑战。在稀疏张量中，相当一部分元素为零。为了减少内存占用并消除冗余计算，通常采用压缩方式，仅存储非零元素的值及其位置。然而，这会产生无法随机访问的嵌套间接数据结构、不规则内存访问模式以及显著的控制开销。即使是最基本的稀疏张量运算，加法 和 乘法，本质上也具有不规则性。更糟糕的是，其最优执行策略取决于操作数稀疏性：当只有一个稀疏操作数时，可以通过间接内存访问进行处理；而当两个操作数均为稀疏时，则需要比较两个张量中的非零位置。

考虑将稀疏矩阵 A（以广泛使用的压缩稀疏行（CSR）格式存储）与稠密向量 b 相乘，该操作可按列表 1a 所示实现。这里，A_vals 和 A_idcs 分别存储非零值及其列索引，而 A_ptrs 则界定 A 的各行。对于结果向量 c 的每个元素，我们都会执行如图所示的稀疏-稠密 点积 sV×dV* 。将该 sV×dV 编译为 RISC-V 指令集代码后，会得到一个包含九条指令的循环，其中只有一条指令——融合乘加（MAC）——执行了必要的计算 [1]。因此，顺序执行核心的浮点运算单元（FPU）利用率最高只能达到11% ；而超标量核心至少需要九个发射槽才能让一个 FPU 始终处于忙碌状态，即使不考虑数据依赖和控制瓶颈也是如此。在这两种情况下，效率的主要瓶颈都是间接寻址b [A_idcs [j]]],，它占用了半数非计算指令，且稀疏-稠密加法也需要这一操作。

现在，考虑将 A 与以常见的压缩稀疏纤维（CSF）格式存储的稀疏向量 b 相乘。每个结果元素的稀疏-稀疏 点积 sV×sV 可按列表1b所示实现。由于只有两个向量均非零的位置才会对结果产生贡献，因此我们求交 两个向量中的非零索引。与 sV×dV 相比，控制开销更高，且动态取决于两个操作数的稀疏模式。稀疏-稀疏张量

• *P. Scheffler* 和 *L. Benini* 任职于苏黎世联邦理工学院集成系统实验室（*IIS*）。电子邮件：{*paulsc,lbenini*}*@iis.ee.ethz.ch*
• *F. Zaruba* 就职于 *Axelera AI*。电子邮箱：*florian.zaruba@axelera.ai*
• *F. Schuiki* 就职于 *SiFive*。电子邮箱：*fabian@schuiki.ch*
• *T. Hoefler* 任职于苏黎世联邦理工学院可扩展并行计算实验室（*SPCL*）。电子邮箱：*htor@inf.ethz.ch*
• *L. Benini* 也任职于博洛尼亚大学电气、电子与信息工程系（*DEI*）。

• © *2023*年 *IEEE*。允许将本材料用于个人用途。所有其他用途均须获得 *IEEE* 的许可，无论是在当前还是未来的任何媒体上，包括为广告或促销目的重印或再版本材料、创作新的汇编作品、转售或再分发至服务器或列表，或在其他作品中重用本作品中任何受版权保护的部分。

• *IEEE*出版物*DOI*：*10.1109/TPDS.2023.3322029*

*下文中，我们分别使用前缀 s 和 d 表示稀疏张量和稠密张量。类似地，M 表示矩阵，V 表示向量。

```
for i in 0 to A_nrows:
  for j in A_ptr[i] to A_ptr[i+1]:
    c[i] += A_vals[j] * b[A_idcs[j]]
```

sV×dV　sM×dV

(a) 将 CSR 格式的稀疏矩阵 A 与稠密向量 b 相乘（sM×dV），通过迭代稀疏-稠密点积（sV×dV）完成。

```
while ia < len_a and ib < len_b:
  if a_idcs[ia] == b_idcs[ib]:
    c_i += a_vals[ia++] * b_vals[ib++]
  while ia < len_a and
        a_icds[ia] < b_idcs[ib]:
    ia++
  while ib < len_b and
        b_idcs[ib] < a_icds[ia]:
    ib++
```

如果索引匹配，乘加运算。

跳过非零元素 在 a 中跳过，直到索引 ≥ 在 b 中的那个

跳过非零元素 在 b 中跳过，直到索引 ≥ 在 a 中的那个

（b）CSF 格式下两个稀疏向量 a 和 b 的点积（sV×sV）。

列表 1：稀疏-稠密和稀疏-稀疏乘积内核示例。

加法需要另一种方法：我们必须形成索引的</style>并集</style>，因为任一操作数中的非零元素都会在结果中产生非零元素。

稀疏计算在许多应用中至关重要 [2]。SuiteSparse 矩阵集合 [3] 收录了来自众多现实问题的稀疏矩阵，涵盖计算物理学、数学、经济学和生物学等领域。在机器学习（ML）中，稀疏化可以在保证给定推理精度的同时，显著减少所需的运算和内存 [4]；一些方法通过仅将权重稀疏化来针对稀疏-稠密计算，而另一些方法还会利用激活稀疏性。图通常表示为高度稀疏的矩阵，是 PageRank [5] 和三角形计数 [6] 等稀疏-稠密和稀疏-稀疏工作负载中的操作数。

成熟的超标量乱序架构难以应对稀疏计算所带来的高控制开销：Fujitsu A64FX 运行 sM×dV 并采用高度优化的格式 [7] 时，性能最高仅为 130.9Gflop/s，即峰值算力的 4.7 %。尽管近期中央处理器、图形处理器和加速器领域的研究提升了稀疏工作负载的性能，但这些方案受限于通用性不足或较大的微体系结构影响。大多数通用指令集架构（ISA）扩展只考虑单侧稀疏性 [8], [9], ，而面向稀疏-稀疏工作负载的方案则会带来较大的面积影响，且无法高效处理前者 [6]。稀疏机器学习方案通常依赖定制的领域专用格式以及低稀疏度或结构化稀疏性 [10], [11], ，而更通用的稀疏线性代数加速器 [12], [13] 相较于其宿主系统和其他方案会显著增加硅面积。

为解决这些不足，我们提出稀疏流语义寄存器(SSSR)，这是一种模块化指令集架构扩展，可显著加速稀疏-稠密和稀疏-稀疏线性代数，同时不会造成重大微体系结构影响，并提升浮点运算单元利用率。SSSR 扩展了流语义寄存器（SSR） [14], ；SSR 可通过将具有规则地址模式的内存流直接映射到架构寄存器，使顺序执行核心在稠密线性代数和其他与数据无关的工作负载中实现近乎满负荷的浮点运算单元利用率† [15]。

。我们扩展 SSR，以加速一般稀疏线性代数所需的间接寻址、交集和并集 操作，同时仅小幅增加整体硬件复杂度，

†A 内存访问模式取决于输入的工作负载　　变体。

```
call    cfg_sssr_a_ft0
call    cfg_sssr_b_ft1
hwloop  1
fmadd.d %c, ft0, ft1, %c
```

SSR ft0：读取 a 的值
SSR ft1：读取 b 的值
对下一条指令循环
乘加

图 2：使用 SSSR 和硬件循环的 RISC-V 汇编代码中的 sV×dV 和 sV×sV。两个内核的差异仅在于所使用的 SSSR 配置。

显著提升稀疏运算的性能。例如，SSSR 使我们能够像清单 2 中那样，在 RISC-V 汇编代码中编写 sV×dV 和 sV×sV，其中寄存器 ft0 和 ft1 具有流语义。由于所有索引处理和比较都由硬件完成，循环体仅包含有用的 fmadd.d 指令，并通过硬件循环反复执行。

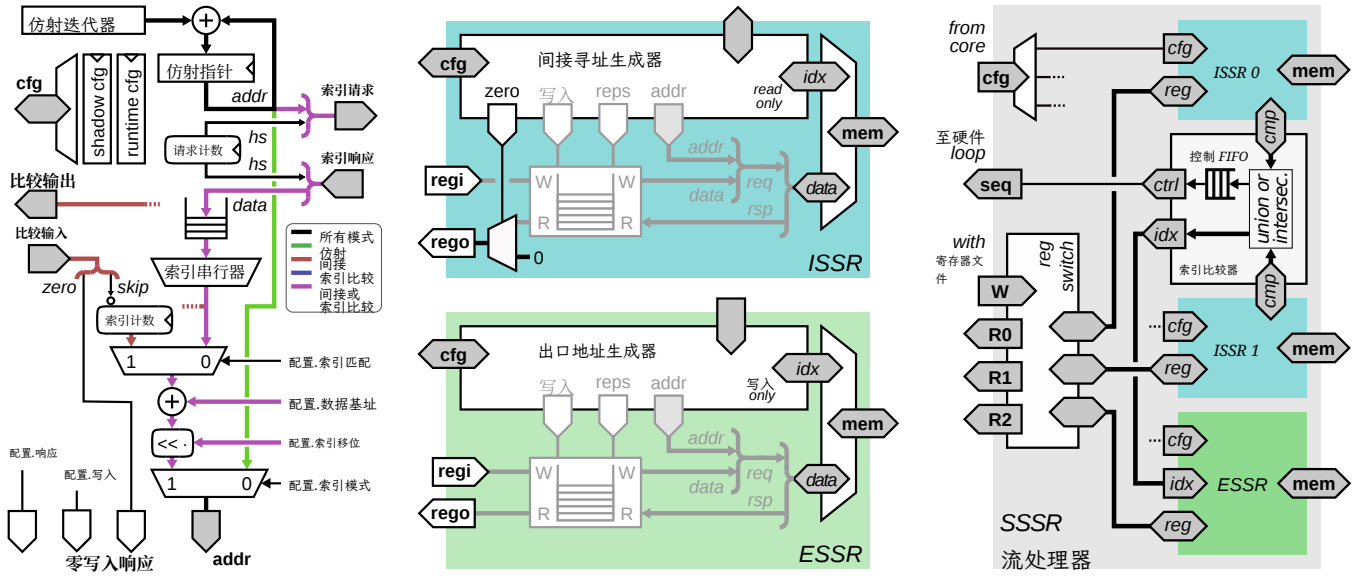
我们将 SSSR 集成到 RISC-V Snitch [16] 核心和计算集群中，对其进行评估，并使用生成的向量和真实世界矩阵，评估它们在各种稀疏线性代数工作负载上的优势。在单个核心上，SSSR 间接寻址、交集和并集分别使稀疏-稠密乘法、稀疏-稀疏乘法和稀疏-稀疏加法相较于经过优化的 RISC-V 基线实现最高 7.0×、7.7× 和 9.8× 的加速比；间接寻址的 FPU 利用率峰值最高可达 80%‡。在八核集群上，使用 SSSR 的稀疏-稠密 sMxdV 最高可快 4.9× 倍，能效提高 2.9× 倍；稀疏-稀疏 sMxsV 最高可快 5.9× 倍，能效提高 3.0× 倍。与常规 SSR 相比，SSSR 在八核集群（128KiB 数据存储器和 8KiB 指令缓存）中仅增加 1.8% 的面积。SSSR 的可综合寄存器传输级（RTL）描述及其在 Snitch 中的集成方案均可免费获取，并以开源形式提供§。

SSSR 扩展了我们先前关于间接 *SSR*加速稀疏-稠密张量积 [1] 的工作。我们在本文中纳入这些贡献，以全面、一致地介绍我们的 SSR 稀疏性扩展。有关稀疏-稀疏线性代数的所有贡献、大部分可参数化性工作以及许多次要改进均为本文原创；我们还将所有现有评估更新到了更新版本的 Snitch [16] 和新的工艺节点。具体而言，我们的贡献如下：

1) 我们提出了对 SSR 的轻量级模块化扩展，可在硬件中处理流式间接寻址、交集和并集，从而在单发射核心上高效处理稀疏通用线性代数，并在开源 RISC-V 核心和多核集群中提供完整的开源硬件实现（第 2 节）。2) 我们提出了 SSSR 的编程模型、高效的稀疏-稠密和稀疏-稀疏线性代数内核（用于处理向量和矩阵），以及该硬件的更多应用（第 3 节）。3) 在具有真实互连和主内存模型的单核（≤ 7.0，7.7，9.8×）和八核集群（≤ 4.9，5.9× 和 2.9，3.0×）上，性能和能效均显著提升，而集群面积仅增加1.8% （第 4 节）。

‡要实现浮点运算单元的完全利用，需要单独的索引存储器端口，我们不采用这种方案，以尽量减少对架构的影响（见第 2.2 节）。

§https://github.com/pulp-platform/snitch：hw/ip/ 中的 SSSR IP Snitch_SSR 以及 hw/ip/Snitch 集群中的集成。



（a）带有间接寻址地址生成器的针对每种模式区分的数据通路。

（b）间接寻址 SSR 和输出 SSR，以及其新的地址生成器和以灰色表示的保留数据移动器。

（c）稀疏 SSR 流式传输器，包含两个 ISSR、一个 ESSR，以及二者之间的索引比较器。

图 1：稀疏 SSR 扩展架构，展示地址生成、数据移动器、SSSR 流式传输器以及 SSR 间索引比较 on.

4) 与最先进的中央处理器、图形处理器和加速器方案相比，该方法通用性更高、面积影响更小，且 sM×dV 浮点运算单元的并行利用率比图形处理器高 1.7×（第 5 节）。

本文的组织结构如下：第 2 节介绍 SSSR 架构及其在 Snitch 核心和集群中的集成。第 3 节介绍 SSSR 编程模型和稀疏线性代数内核。第 4 节评估 SSSR 带来的性能、时序、面积和能效优势。最后，第 5 节讨论相关的中央处理器、图形处理器和加速器研究，并将其与我们的方法进行比较。

2 架构

我们的硬件扩展大多局限于每个 SSR 内部的地址生成器。我们将首先提出两种新型生成器设计，即间接寻址地址生成器和输出地址生成器，以及采用它们的两种扩展型、向后兼容的 SSR 硬件变体。随后，我们将介绍稀疏 SSR 流式传输器，它组合并连接三个扩展型 SSR 的索引数据通路，以支持索引交集和并集。最后，我们将讨论该流式传输器如何集成到 Snitch 核心和集群 [16] 中，并在其中评估我们的扩展。需要指出的是，SSSR 并不严格依赖特定的核心或指令集，可用于任何核心、ISA 或硬件循环。

2.1 地址生成器

我们设计了两个扩展的 SSR 地址生成器，用于处理索引流。间接寻址生成器可以从内存中读取索引流，以生成间接地址，或将读取的索引与另一个间接寻址 SSR（ISSR）的索引进行匹配，从而求交集或并集。出口生成器可以将外部提供的索引流写入内存，同时发出相应的数据地址。

第 2.3 节说明了如何组合这些功能，以实现端到端的稀疏-稀疏索引处理。

2.1.1 间接寻址地址生成器

图 1a 展示了间接寻址地址生成器，并突出显示了不同生成模式所使用的数据通路。与原始生成器设计类似，它向主机核心提供影子配置寄存器，使得在一个流仍在运行时即可设置另一个新流；同时还提供了与外围 SSR 逻辑交互的控制接口。索引流通过额外的只读内存端口获取。

所有生成模式都重用现有的、最多支持四层嵌套的仿射地址生成器，但用途各不相同。在进行 仿迭代时，它直接提供地址生成器的输出，如图所示。在间接寻址或索引匹配期间，它为内存中包含待读取索引的字生成地址；与逐个获取索引不同，这种方式能够充分利用内存总线。这些字会被获取到一个解耦先进先出（FIFO）队列中。为防止下游阻塞或队列溢出，未完成请求计数器会限制在途请求的数量。

SSSR 支持任意无符号 2^nB 整数类型的索引，只要其大小适配所使用的内存总线（即在 Snitch 上为8、16、32 和 64 位）。索引串行器会从缓冲的索引字中提取指定大小的索引。为提高灵活性和可编程性，我们支持内存中任意索引数组基地址对齐。在间接寻址模式下，串行化后的索引会再加上一个可编程的小偏移量，从而支持二次幂步长；这样，无需引入高成本的硬件乘法器，即可对二次幂步长张量的高阶轴进行间接寻址。最后，将偏移后的索引加到配置的基地址上，即可得到所需的间接地址流。

在索引匹配过程中，索引会被传递到外部匹配单元（详见第 2.3 节），该单元会将读取的索引与另一个 SSR 的 gen- 进行比较。

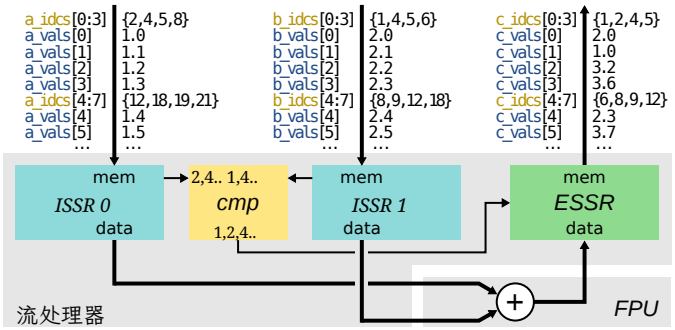


图 2：端到端 sV+sV 使用 64 位浮点数据和 16 位索引时的 SSSR 内存请求序列及索引并集数据流示例。

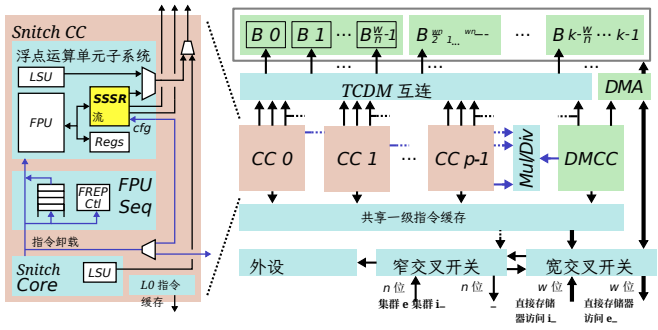


图 3：评估中使用的 Snitch 集群及 SSSR 流式传输器

另一个操作数的流由迭代器生成。与间接寻址不同，发出的地址从配置的数据基址开始以单位步长递增，因为每个索引恰好对应一个非零值。然而，取决于哪个索引流领先，匹配单元可以指示任一生成器跳过地址，或在 SSR 数据流中插入空数据元素，从而形成索引流的交集或并集。

2.1.2 出口地址生成器

出口地址生成器要简单得多。它满足了将联合流的索引与其数据一同写回的需求。为此，它通过额外端口读取由索引匹配单元生成的联合索引，将其传递给索引合并器，并通过只写存储器端口将其写回，实际上反转了间接寻址生成器的索引获取数据通路。它为每个发出的地址恰好写入一个索引，但允许索引写入在参数化数量的数据元素之前进行，从而解耦索引流和数据流，避免不必要的阻塞。

两个扩展生成器都支持完全参数化。地址和数据宽度、仿射循环次数以及索引队列深度均可调整。对于在较小存储器上运行的高面积效率设计，可以采用较小的内部索引和地址宽度。

2.2 稀疏 SSR

图1b展示了我们围绕所提出的地址生成器构建的扩展 SSR，即 ISSR 和出口流式寄存器（ESSR）。与原始 SSR 一样，两者都提供数据队列，用于解耦寄存器端口和存储器端口，并同时支持读取和写入方向。两者都公开其地址生成器的索引匹配端口；此外，ISSR 还增加了一些多路复用逻辑，使间接寻址生成器能够向读取流中注入零数据元素，以实现流并集。

在 ISSR 和 ESSR 中，我们选择使用轮询仲裁器将索引存储器端口与数据存储器端口合并，而不是要求配备两个存储器端口。在仿射流式传输期间，这不会影响数据吞吐量，因为索引端口未被使用。对于间接流式传输和出口流式传输，这种权衡的理由如下：每传输 n 个数据字，只需传输一个索引字，其中 n 是每个存储器字包含的索引数。因此，数据搬运器利用率的峰值为 $n/n+1$；在我们的评估平台上，对于64位数据，这相当于 67%、80%和88% 分别对应32位、16位和8位索引。另一种方案是在流处理器中仲裁内存访问，或为每个 SSR 配置两个核心内存端口，以稀疏流上的更高峰值利用率为代价，换取明显更大的集群互连。本文研究面积高效的单端口 SSSR 变体，可直接替换现有的单端口 SSR，并与之进行直接比较。

2.3 稀疏 SSR 流式传输器

图 1c 展示了处于默认配置下的 SSSR 流式传输器。与原始 SSR 流式传输器一样，它会合并所有可用的 SSR，并向主机处理器提供共享配置和寄存器文件接口。启用后，寄存器开关会将每个 SSR 的数据通道映射到预定义的寄存器。

SSSR 流式传输器支持完全参数化，可配置为将任意数量的 SSR、ISSR 或 ESSR 映射到任意寄存器索引。不过，如果具备可选的 SSR 间索引流合并功能，则该功能会受到一些配置限制。单个流式传输器最多提供一个索引比较器；因此，流式传输器中只有两个 ISSR 可以相互进行索引比较，并且可选的第三个 ESSR 可以使用合并后的索引流进行回写。因此，默认配置包含两个进行索引比较的 ISSR 和一个 ESSR。

索引比较器会判断两个 ISSR 索引流中哪一个（如果有）领先，并在必要时推进二者，以输出索引的并集或交集。图 2 以 sV+sV 为例，展示了索引比较器如何生成 ISSR 索引的并集，并将其转发给 ESSR 进行回写，同时由主机核心的浮点运算单元执行所需的加法运算。所需的比较模式在 ISSR 中配置，并由 ISSR 转发。由于联合流中的索引-数据对数量事先未知，比较器还提供了一个通用的流控制队列，其中的单个位用于指示是否还有下一个元素对可用，或联合数据流是否已结束。主机核心的分支单元或硬件循环单元可以解耦地读取此队列，以确定联合流的结束位置。

2.4 核心与集群集成

图 3 展示了默认 SSSR 流式传输器集成到用于评估的 Snitch 核心复合体（CC）和集群 [16] 中的情况。与此前集成的 SSR 流处理器一样，它与双精度浮点运算单元的寄存器文件复用。

	参数	可能的取值	Used
p	工作核心数量	$\mathbb{N}^{>0}$	8
n	窄（核心、存储体）位宽	32、64	64
w	宽（L1指令缓存、DMA）位宽	$\{n \leq 2^j \leq 1024 \mid j \in \mathbb{N}\}$	512
k	存储体数量	w/n的倍数	32
D	紧耦合数据存储器容量（单位：KiB	$\{2^j \mid j \in \mathbb{N}\}$	128
I	L1 指令缓存容量（单位：KiB	$\{2^j \mid j \in \mathbb{N}\}$	8

表1：评估中使用的 Snitch 集群参数及取值。

CC单元保留现有的内存拓扑，将核心、浮点运算单元和第一个 ISSR 合并到一个内存端口，并为第二个 ISSR 和 ESSR 提供专用端口。我们还扩展了现有的浮点重复（FREP）硬件循环，新增一种模式，通过流控制接口针对每个联合流元素发出一次迭代。

表1列出了完全可配置集群的关键设计参数及其可能取值。所有核心共享一个 L1 指令缓存和一个整数乘除单元。连接到宽总线的直接存储器访问（DMA）引擎支持紧耦合数据存储器（TCDM）与主存之间的高带宽数据传输。该引擎由轻量级数据移动控制器（DMCC）控制，也可用于集群管理与协调。

3 编程与应用

SSSR 由主机核心通过自定义指令访问的寄存器接口进行配置。这使得作业能够快速设置：在我们的内核中，从冷启动状态为全部三个 SSSR 设置并启动新作业，最多只需 10 个周期。在我们的配置中，ISSR 0 和 1 分别映射到临时浮点运算单元寄存器 ft0 和 ft1，而 ESSR 映射到 ft2。通过通用控制寄存器 [14]，即可切换将寄存器访问重定向到 SSR。我们着重于在指令级加速稀疏线性代数，方法是构建手工优化的算子库，这也是高性能线性代数软件包的常见做法。未来的工作将着重于扩展编译器，使其能够自动推断高层代码中的间接寻址、交集和并集，并将其映射到 SSSR。

3.1 可加速的格式与操作

尽管我们会在第 3.3 节进一步探讨其他工作负载，但我们的主要目标是在一组成熟的张量格式上加速通用稀疏线性代数。与专注于低稀疏度的方法不同， [10], [11], [17], SSSR 旨在兼顾灵活性和可扩展性，从而在更广泛的稀疏度范围内实现显著的加速比（见第 4.1 节）。SSSR 支持沿任意张量格式的主轴迭代，该轴由两个数组表示：一个存储非零值的值数组，以及一个存储这些值位置的索引数组。我们将这对数组称为稀疏纤维。基于纤维的张量格式包括常见的 CSR [18] 和 CSC [19] 矩阵格式、通用的 CSF [20] 张量格式，以及许多格式变体；这些变体会对数据进行切片、将其打包成块，或进一步压缩张量的高阶维度。第 2.1 节讨论的索引移位器还支持沿更高层级的轴迭代，前提是该轴以下的所有轴的维度均为 2 的幂；实践中可通过对数据分块来确保这一点。索引可以采用任意无符号 2^nB 整数类型，只要其宽度不超过内存总线的位宽。

SSSR 的数据位宽固定为主机处理器的位宽，但借助零填充，可以轻松支持在 BCSR [21] 等分块格式上进行单指令多数据（SIMD）计算。例如，Snitch 中的 64 位浮点运算单元在分别对两个或四个元素组成的块进行计算时，可在 32 位和 16 位格式上利用其 2 路和 4 路 SIMD 能力。我们可以像处理单个双精度元素一样处理非零 SIMD 块；这种替换与 SSSR 的使用彼此独立，只是将较窄的非零元素整合为 64 位非零 SIMD 块。因此，对于给定的非零模式，SSSR 应能实现相近的峰值加速比，但加速比会有所降低，因为必须将更多非零元素按比例打包在一起，才能抵消 SSSR 的设置开销。

SSSR 不受特定稀疏度范围的限制。借助 SSR 向后兼容性，它们能够显著提升稀疏-稀疏、稀疏-稠密乃至稠密问题的效率。所支持的格式没有结构化稀疏性要求，并且能够很好地适应极端稀疏度。此外，SSSR 不会限定特定的高阶数据流：例如，稀疏-稀疏矩阵乘法（sM×sM）可以采用内积、外积或按行的方式，以更好地适应应用和数据。

为了优化面积和漏电，SSSR 流处理器的参数配置可根据特定应用需求进行调整。对于高效的稀疏-稠密乘法，一个 ISSR 和一个 SSR 即已足够；进行加法时则需要额外一个 ISSR。对于高效的稀疏-稀疏乘法，两个 ISSR 即已足够；进行加法时则需额外配置一个 ESSR。

3.2 稀疏线性代数内核

为了展示我们扩展的通用性并评估其优势，我们为扩展了 SSSR 的 RISC-V Snitch 系统创建了一组稀疏-稠密和稀疏-稀疏线性代数内核。这些内核处理 CSR 矩阵和 CSF 向量。每个内核均针对 8、16 和 32 位索引类型实现；除基于交集的内核外，每个内核还提供三种变体：

- BASE：优化后的标准 RISC-V 基线。
- SSR：扩展了 FREP 循环和 SSR 的 RISC-V
- SSSR：扩展了 FREP 循环和 SSSR 的 RISC-V

为了获得 BASE 内核，我们使用 clang -Ofast 编译高级 C 代码，并针对 SnitchCC 单元对生成的汇编代码进行手动优化，通过指令重排和循环展开，尽量减少内存或依赖停顿。SSR 内核则进一步提升性能：将稀疏值数组等规则的仿射值流映射到 SSR，但不使用我们的稀疏性扩展。需要指出的是，这不包括使用交集的内核，因为常规 SSR 无法进一步加速条件流加载。最后，我们将所有内核实现为充分利用扩展的 SSSR 变体，详情将在后文讨论。

3.2.1 稀疏-稠密内核

稀疏向量×稠密向量：清单 3 展示了我们的 SSSR 稀疏向量×稠密向量内核。不同索引大小的版本仅在 ft1 的间接寻址配置上有所不同。我们的目标是以最小开销持续发出浮点乘加指令 fmadd.d，从而最大限度地提高 FPU 利用率。我们分三个阶段实现这一目标：

```
csrsi    ssr_redir, 1                  } 将 ft0、ft1、ft2 重定向至 SSR
call     sssr_affine_setup_ft0         } SSR ft0：稀疏 a_值
call     sssr_indir_setup_ft1          } SSR ft1：a_vals 处的稠密 b
fcvt.d.w ft3, zero                     } 零初始化 ft3 ... ft[3+n]
...
frep     %len_a, 1, %n, 0b1001         } 交错累加器 n 倍
fmadd.d  ft3, ft0, ft1, ft3
...
fadd.d   fa0, fa1, ft3                 } 减少累加器
csrci    ssr_redir                     } 禁用 SSR 重定向
```

Lst. 3：sV×dV 内核的 SSSR 变体：ft0 流式传输稀疏向量*a* 的值，ft1 流式传输来自稠密向量*b*的间接值。

```
csrsi    ssr_redir, 1                  } 将 ft0、ft1、ft2 重定向至 SSR
call     sssr_union_setup_ft0          } SSR ft0：匹配到 a_个值或 0
call     sssr_union_setup_ft1          } SSR ft1：匹配到 b_个值或 0
call     sssr_egress_setup_ft2         } SSR ft2：结果 c_vals
frep.s   1, %n, 0b000                  } 持续输出，直到并集流结束
fadd.d   ft2, ft0, ft1
call     core_fpu_fence                } 等待浮点运算单元空闲（任务完成）
ssrcfg.r a0, strctl_len                } 将结果长度读入 a0
csrci    ssr_redir                     } 禁用 SSR 重定向
```

图 4：sV+sV 内核的 SSSR 变体：ft0 和 ft1 从稀疏操作数 *a* 和 *b* 中流式读取，ft2 则将稀疏结果*c*流式写出。

1) 设置：我们首先启用寄存器到 SSR 的重定向。然后设置 ISSR ft0，使其以仿射模式线性流式传输稀疏向量 a 的值；并设置 ISSR ft1，使其根据 a 的索引间接读取稠密向量 b。最后，将从 ft3 开始的一段连续累加器寄存器块初始化为零。
2）计算：我们使用 FREP 重复执行所需的 fmadd.d 指令，并通过先前配置的 ISSR 流式传输两个乘数。为防止累加和的寄存器数据依赖造成停顿，我们使用 FREP 现有的寄存器错开功能，在每次迭代时递增累加器寄存器索引，从而在连续的寄存器块中维护多个部分和；
Zaruba 等人对此进行了详细说明 [16]。由于 ISSR 的单内存端口限制了峰值吞吐量，索引类型越大，隐藏这些停顿所需的累加器就越少。3）收尾：我们将累加器归约到返回寄存器 fa0，并禁用 SSR 重定向。

由于计算和收尾阶段仅包含 FPU 指令，因此当 FREP 使 FPU 保持忙碌时，核心可以继续执行；这也使新的影子 SSR 任务能够无缝设置。必要时，可以强制核心与 FPU 之间进行同步。

sM×dV：我们不再只是针对矩阵的每一行简单地迭代 sV×dV 内核，而是进一步优化 sM×dV，以减少设置和归约开销，并最大限度地提高 FPU 利用率。我们通过单个 SSR 和 ISSR 作业流式处理整个矩阵纤维，显著降低了设置开销。此外，我们将每行开头的几次融合乘加展开，并为较短的归约添加分支；仅在必要时才发出 FREP 循环并执行完整归约，以加速较短的行。

在所有变体中，我们都使用 32 位行指针，以最大限度地扩展行规模。虽然默认情况下我们从左侧乘以压缩稀疏行矩阵，但我们的内核提供运行时参数，可使用不同的 2 的幂次向量步幅和任意结果步幅；这样便能从任一侧乘以压缩稀疏行矩阵或压缩稀疏列矩阵，并作用于具有任意 2 的幂次步幅的稠密张量轴。

sM×dM：我们将一个压缩稀疏行矩阵与一个列步幅为 2 的幂次的行主序矩阵相乘。此处我们只需迭代 sMxdV 内核，因为在第三阶循环中迭代稠密数据所增加的开销可以忽略不计（参见第 4.1.1 节）。我们再次提供自定义稠密矩阵和结果步幅的参数，从而支持 CSR 或 CSC 矩阵与行优先或列优先矩阵从任一侧进行乘法。唯一的限制是，我们的硬件索引移位器要求被间接访问的稠密轴具有 2 的幂次步幅；

在实践中，可以通过使用 clusterDMA 的跨步传输对矩阵进行分块来满足这一要求。

sV+dV：我们假设将结果累加到稠密向量中；如有必要，可以预先使用 SSR 创建一个副本。我们将 ft2 配置为流式读取稀疏向量值，将 ft0 配置为收集稠密加数，并将 ft2 配置为将求和元素散布回稠密向量；然后使用 FREP，对每个稀疏值执行一次 fadd.d ft1, ft0, ft2。通过改为迭代执行融合乘加 fmadd，也可以用相同性能将任一向量乘以一个常数。这种逐元素加法可以通过更高层的迭代扩展到任意 形状的基于纤维的张量。

sV⊙dV：由于其中一个操作数是稠密的，结果索引与稀疏操作数索引相同；如有需要，也可以使用 SSR 复制这些索引。与 sV+dV 一样，我们将每个稀疏值与收集到的稠密协操作数相乘，但改为线性写出结果，以生成相应的结果值数组。同样，这种逐元素乘法可以通过迭代来适配任意形状的张量。

3.2.2 稀疏-稀疏内核

sV×sV：由于间接寻址和交集都完全在 SSSR 内部处理，SSSR sV×sV 内核与其 sV×dV 对应内核完全相同，区别仅在于所使用的 SSSR 和 FREP 配置。我们将 ft0 和 ft1 配置为流式传输每个操作数的纤维，并启用交集功能，使得只有索引匹配的值对才会被流式传输至浮点运算单元。由于交集操作中产生的值对数量无法预先确定，我们使用流控 FREP 变体（frep.s），为每个值对发出一条浮点乘加指令。

sM×sV：对于每个结果元素，我们都会迭代执行 sV×sV 代码。由于我们的交集方法要求纤维中的索引按递增顺序排列，因此与 sM×sV 不同，我们会为每一行启动新的 SSSR 请求。不过，通过核心与浮点运算单元子系统的解耦，以及影子 SSSR 配置接口，我们可以隐藏部分配置开销。

sM×sM：三种稀疏-稀疏矩阵乘法数据流均可使用 SSSR 加速，通常可分为以下几类：

- 内积：针对每个行列对计算内积（sV×sV），逐元素计算输出。
- 按行：通过累加左矩阵的稀疏行（sV+sV）与右矩阵各行中的非零元素相乘后的结果，逐行计算输出。
- 外积：针对每个行列对计算外积（稠密向量缩放），然后累加所有部分稀疏结果矩阵（sM+sM）。

每种方法都有不同的迭代顺序、内存占用、通信开销和数据重用，因此

各自在不同的应用和稀疏度范围内具有优势。SSSR 可通过加速上述子内核来加速所有三种方法。本文仅实现一个 CSR×CSC 内积内核，用于迭代我们的 sM×sV 内核。

sV+sV：如代码清单 4 所示，我们从 ISSR ft0 和 ft1 中读取两个稀疏向量，并将它们的稀疏和写入 ESSR ft2。索引比较器执行索引并集，使 ft0 和 ft1 输出元素值；如果某个索引仅出现在其中一个操作数中，则输出零。因此，我们只需像示例中那样迭代执行双精度浮点加法，而由流控制的 frep.s 则确保迭代次数正确。由于 ft2 被配置为输出端，每计算出一个和，就会将一个比较后的索引写入内存，从而生成和的索引向量。任务完成后，我们从 ESSR 配置寄存器中读取结果长度；为确保任务完成，我们会同步核心和浮点运算单元流水线。在迭代执行此内核以相加更高维的稀疏张量（例如 sM+sM）时，我们可以像处理 sM×sV 一样，推迟读取长度，以便在计算与配置之间实现重叠。

sV⊙sV：此内核与 sV+sV 几乎完全相同；不同之处在于，我们将索引比较器配置为计算交集 ，并迭代执行 fmul.d。关于更高维张量以及控制与计算重叠的相同见解也适用于此处。

3.3 更多 SSSR 应用

SSSR 面向通用稀疏线性代数，可加速以此为基础构建的众多应用，包括有限元分析（FEM）模拟、稀疏神经网络和图神经网络，以及 PageRank [5] 等图工作负载。不过，SSSR 加速的间接寻址、交集和并集都是通用操作，支撑着更多不规则工作负载；下面我们将探讨其中的一些：

码本解码：ISSR 可以流式读取压缩数据，这些数据以索引数组的形式存储，索引指向较小数组中重复出现的值。这类码本可用于高效存储量化后的深度学习参数 [22], 图像通道 [23], 以及稀疏张量的非零元素。流处理器中的每个 ISSR 都可以独立地流式读取经码本编码的向量。两个 ISSR 可以处理含有码本编码非零元素的稀疏-稠密线性代数（一个收集稠密操作数，另一个收集非零元素），性能与我们的稀疏-稠密内核相近，同时显著减小可压缩张量的体积。

模板代码：迭代模板代码在 n维网格上处理数据，相对于每个网格点，按照固定但不规则的模式访问数组。例如，偏微分方程和图像处理 [24]。ISSR 可以将模板存储为索引数组，并以每个网格点的偏移量作为基地址进行流式传输，从而加速模板计算。同样的方法也可以加速稀疏卷积，且无需进行类似 im2col 的预处理。

图模式匹配：利用图节点稀疏邻接向量之间的交集，可以识别并统计子图嵌入，并评估图的相似性 [6]。因此，SSSR 可用于加速计算机视觉和药物发现等领域的图模式匹配工作负载。

散布-聚集数据变换：ISSR 支持流式散布-聚集变体。因此，它们可以加速任何现有的基于散布-聚集的数据重排变换，in-包括稀疏矩阵转置 [25], 并行基数排序 [26],，或通过散射其非零元素来稠密化稀疏张量。

4 评估

我们首先将在单个 Snitch CC 上评估 SSSR 对线性代数内核的性能优势，并借助逼真的动态随机存取存储器（DRAM）模拟器，评估并行集群 sM×dV 和 sM×sV 实现。接下来，我们将讨论流处理器不同硬件配置对面积和时序的影响。最后，我们将研究这些扩展带来的能效优势。

与所集成的基准 Snitch 平台一样，我们的硬件扩展被设计并建模为参数化、完全可综合的 SystemVerilog RTL 硬件描述。这样，我们便能够以完整的周期精度和可见性对基准系统及扩展系统进行仿真，从而实现对所有组件的精确跟踪和利用率监测。虽然我们可以使用任何支持 SystemVerilog 的仿真器，但这里使用 *QuestaAdvanced Simulator*。用于评估面积、时序和功耗的实现，采用与仿真输入相同的 RTL 描述。模拟 CC 或集群外部的硬件资源采用行为级方式建模。除非另有说明，所评估的集群和 CC 使用默认的 SSSR 流式传输器，并按照表1中的规定进行参数化。

对于所有实验，稠密测试张量通过采样正态分布值获得；稀疏向量则针对给定的非零元素数量和维度生成，其值服从正态分布，索引服从均匀分布。所使用的稀疏矩阵均来自 SuiteSparse Matrix Collection [3] 的真实问题矩阵；它们涵盖各种问题领域和长宽比，包含 2k 至 3.2k 列，以及 2.8k 至 543k 个非零元素。

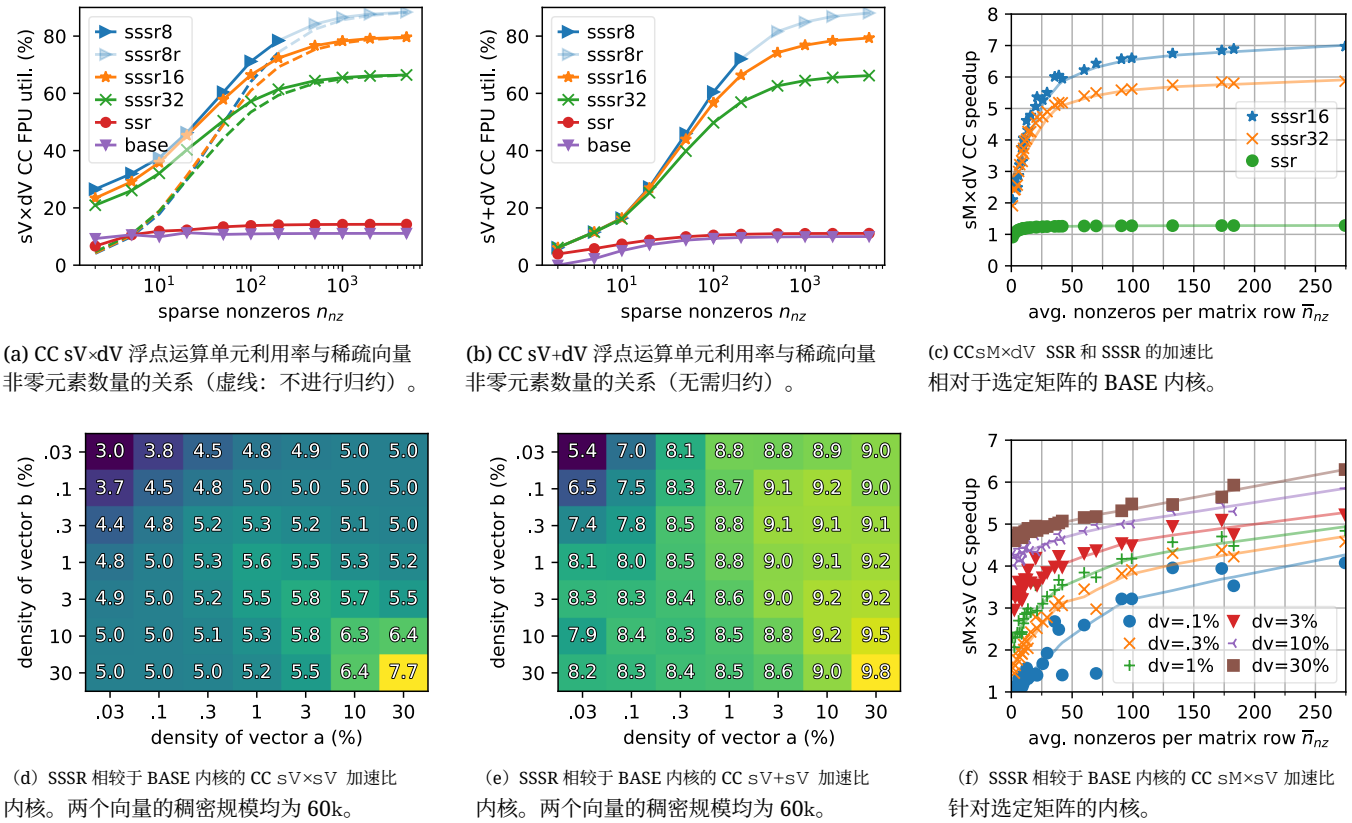
4.1 单核性能

为了评估 SSSR 带来的性能优势，同时排除其他系统组件的影响，我们首先在 RTL 仿真中评估单个 CC 的线性代数内核：将其连接到专用指令缓存和专用三端口数据存储器。这两者分别与集群中的共享指令缓存和 TCDM 通道类似，但还会产生额外的缓存未命中和存储体冲突；我们将在第 4.2 节中纳入并讨论这些因素的影响。

4.1.1 稀疏-稠密内核

稀疏向量×稠密向量：图 4a 展示了稀疏向量×稠密向量内核在稀疏向量非零元素数量 n_{nz}不同时的 FPU 利用率。只要稠密向量的长度未超出 TCDM 的容量，内核的运行时间就不会受其影响。由于任何长度的 RISC-V 加载指令都只占用一条指令，非 SSSR 变体在所有索引位宽下的表现都相同。图中还展示了 SSSR 内核在使用和不使用累加器归约（虚线）时的 FPU 利用率。由于采用 8 位索引的 CSF 向量最多可容纳 256 个非零元素，我们还展示了使用 8 位索引间接寻址并重复索引的结果（sssr8r）。

三种 SSSR 索引变体的性能均显著优于 BASE 和 SSR 内核，其中 16 位和 32 位变体接近其仲裁施加的 67% 和 80% 利用率上限，并实现了比 SSR 高出 4.7× 和 5.6× 的浮点运算单元利用率



(a) CC sV×dV 浮点运算单元利用率与稀疏向量非零元素数量的关系（虚线：不进行归约）。

(b) CC sV+dV 浮点运算单元利用率与稀疏向量非零元素数量的关系（无需归约）。

(c) CCsM×dV SSR 和 SSSR 的加速比相对于选定矩阵的 BASE 内核。

(d) SSSR 相较于 BASE 内核的 CC sV×sV 加速比内核。两个向量的稠密规模均为 60k。

(e) SSSR 相较于 BASE 内核的 CC sV+sV 加速比内核。两个向量的稠密规模均为 60k。

(f) SSSR 相较于 BASE 内核的 CC sM×sV 加速比针对选定矩阵的内核。

图 4：利用间接寻址、交集和并集的部分内核的单 CC 性能结果。图 4a 和 4b 包含假设 8 位索引多次出现（sssr8r）的实验结果。图 4c 和 4f 中叠加的趋势线采用局部估计散点图平滑法（LOESS）拟合。

。8 位索引内核的利用率最高可提升 5.8×，而采用重复索引的 8 位间接寻址接近其 88% 的仲裁上限。我们注意到，BASE 和 SSR 也分别接近其受发射限制的利用率上限，即每九个周期和每七个周期执行一次乘加运算。

由于处理非零元素的速度快得多，SSSR 内核需要显著更高的 n_{nz}，才能完全抵消其设置和归约开销。当非零元素较少时，它们在不进行归约时的利用率甚至低于 BASE 和 SSR，这促使我们在 SSSR sM×dV 内核中采用行展开。由于使用较小索引的 SSSR 内核需要更多累加器才能维持峰值利用率，因此只有在 n_{nz}较高时，它们才优于使用较大索引的对应内核。

sV+dV：图 4b 显示了我们的 sV+dV 内核的 FPU 利用率。这里，结果累加到稠密向量中。我们观察到与 sV×dV 类似的趋势，但有两个关键差异。由于每个非零元素都会写回一个结果值，BASE 和 SSR 的峰值利用率进一步下降，分别为每十个周期和每九个周期执行一次乘加运算。出于相同原因，SSSR 内核无需进一步归约，因为结果会通过使用第二个 ISSR 的流式间接寻址立即写回。使用 32 位索引、16 位索引以及带重用的 8 位索引的 SSSR 内核，仍分别接近其 67%、80% 和 88% 的仲裁限制。

sM×dV：图 4c 展示了 SSSR sM×dV 内核相对于 BASE 的加速比与每个矩阵行的平均非零元素数量 n_{nz} 之间的关系，这反映了每行执行的乘加运算次数。这里我们假设紧耦合数据存储器足够大，可以存储完整矩阵。我们不评估 8 位索引，因为它们不足以索引所选矩阵的列。

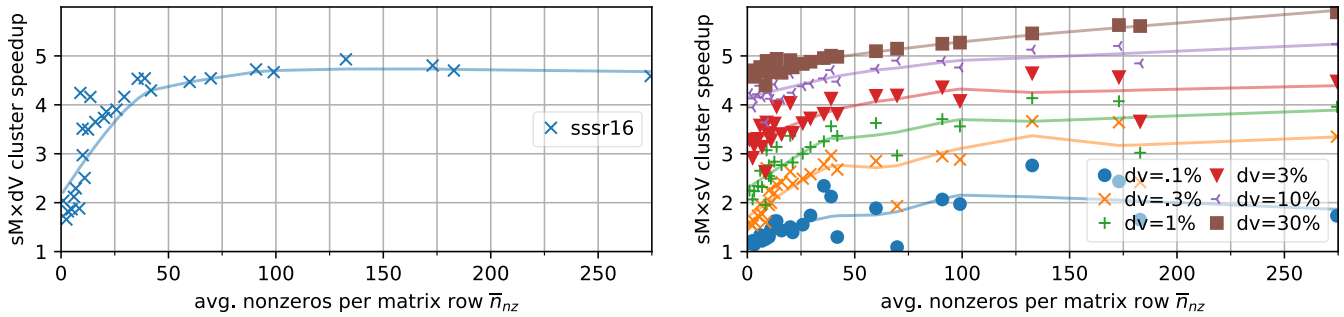
对于 sV×dV，我们的 16 位和 32 位 SSSR 内核相较于 BASE 的加速比接近理论极限，最高分别达到 5.9× 和 7.0×；为弥补行迭代和归约开销，其 n_{nz} 必须高于 BASE 和 SSR。浮点运算单元利用率最高分别达到66% 和79%。同样，16 位 SSSR 内核仅在 $n_{nz} \gtrsim 20$ 时优于 32 位变体，这是因为它的归约序列更长。

sM×dM：即使在边缘情况下，加速比和浮点运算单元利用率也几乎与我们考察的 sM×dV 内核相同：对于仅含 64 个非零元素的微小稀疏矩阵*Ragusa18*，与双列稠密矩阵相乘而非与稠密向量相乘时，浮点运算单元利用率仅变化0.12%。

4.1.2 稀疏-稀疏内核

sV×sV：图 4d 展示了我们的 16 位 SSSR sV×sV 内核相较于对应 BASE 内核的加速比，向量密度为0.03-30%，向量长度为 60k。我们观察到加速比为3.0-7.7×，并随着两个操作数的密度增加而近乎对称地上升。密度相近时，加速比通常更高；随着密度差异增大，加速比趋近于 5.0×。

为理解这些趋势，我们首先考察两种稳态边缘情况。当遇到没有交集、并扫描一个向量的非零元素时，BASE 每个非零元素需要五个周期，而 SSSR 需要一个周期，这解释了密度差异情况下 5.0× 的加速比上限。当只有交集需要处理时，BASE 每个非零元素对需要 18 个周期，而 SSSR 需要 1.25 个周期，因此可实现的峰值加速比为 14.4×；若非零位置完全重合，FPU 利用率则为80%。因此，



(a) 对于 16 位索引和选定矩阵，集群 sM×dV 中 SSSR 相对于 BASE 的加速比。(b) 对于 16 位索引、不同向量密度和选定矩阵，集群 sM×sV 中 SSSR 相对于 BASE 的加速比。

图 5：选定线性代数内核扩展到八核集群后的性能结果。趋势线使用 LOESS 拟合。

随着两种密度下索引匹配的可能性增加，加速比也会随之提高。最后，当两个向量都变得非常稀疏、只包含少量元素时，交集运算会很快结束，而 SSSR 的设置开销会抵消加速效果。随机向量的非零位置和序列也会对结果产生影响，造成轻微的不对称。

sV+sV：图 4e 展示了我们的 16 位 SSSR sV+sV 内核相对于 BASE 的加速比，所用向量与 sV×sV 相同。加速比介于 5.4 到 9.8× 之间，并随着两个操作数的密度增加而提高，且操作数之间存在轻微的不对称性。

为理解这些趋势，我们再次考察边缘情况。假设索引比较连续进行，仅读取加数 a 或 b时，SSSR 的处理速度最高可分别快 9.6× 和 8.8× 倍。实际上，ESSR 索引写入时偶尔出现的背压会使加速比分别降至观测到的 9.0×和 8.2×倍。在 a 和b 情况下，加速比并不对称，因为 BASE 中的三元分支代码会产生一个或两个分支周期。索引连续匹配时，峰值加速比再次达到 14.4×倍。因此，随着两个操作数的密度增加，且两者密度相近，加速比也会提高。

sM×sV：图 4f 展示了我们的 16 位 SSSR sM×sV 内核相对于 BASE 的加速比，并以不同向量密度下每个矩阵行的平均非零元素数 n_{nz} 为基准。我们再次假设紧耦合数据存储器中存放着完整矩阵。

我们观察到与迭代的 sV×dV 内核相似的趋势：两个参数的密度越高，加速比越大；在对应密度下，加速比接近 sV×dV 内核，最高可达 6.3×。然而，稠密向量维度较小，意味着随着密度降低，抵消效应和调制效应会更早显现：当向量密度为 0.1% 或非零元素数为 2-3 时，加速比会显著降低，且会随非零元素模式大幅波动。不过，加速比仍高于 1，这意味着即使两个参数中的非零元素很少，我们依然能够摊销 SSSR 设置开销。

4.2 集群性能

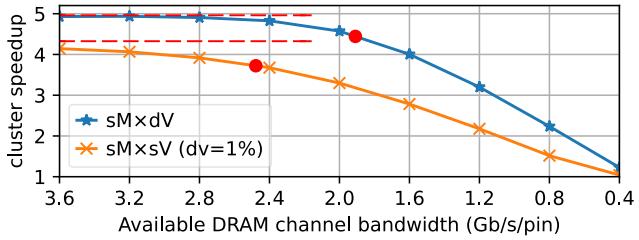
为评估多核心和系统级性能，我们在 Snitch 集群上实现了 sM×dV 和 sM×dV 的并行扩展。我们重用了针对架构优化的单核心内核，在各核心之间动态分配大小可变的行块，并针对矩阵实现了基于 clusterDMA 的双缓冲数据搬移方案。

为确保内存吞吐量、延迟和调度符合实际情况，我们将集群连接到八个通道之一—Micron MT54A16G808A00AC-36 HBM2E DRAM 的通道（3.6 Gb/s/引脚 或 57.6 GB/s 峰值带宽，88ns 平均往返延迟¶），我们使用*DRAM-Sys* [27]对其进行模拟。所有输入数据和指令最初都驻留在该 DRAM 中，所有结果也都会写回其中。一个 16KiB 四路组相联 L2 指令缓存用于缓存指令取指，DMA 数据读取和写入会绕过该缓存。除了*DRAMSys*所建模的 DRAM、PHY 和控制器延迟外，我们还将片上互连的延迟建模为额外的 16 个周期的正向和反向延迟。我们还在第 4.2.1 节中进一步探讨互连延迟和 DRAM通道带宽共享的影响。

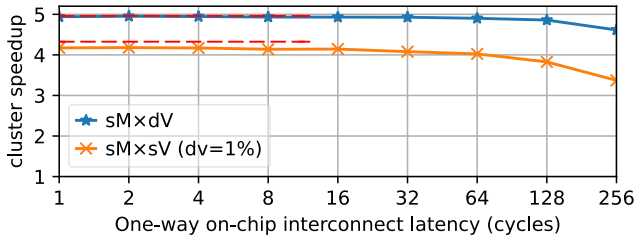
sM×dV：图 5a 展示了在集群上并行运行的 sM×dV 中，16 位 SSSR 内核相较于 BASE 内核的加速比。即使对于 $n_{nz} = 1$，加速比也很显著，达到 1.7×，最高可达 4.9×，并且在 $n_{nz} > 30$ 上持续超过 4×。我们观察到的趋势与单核 sM×dV 相同，但加速比有所降低，波动也更明显，这可归因于多种原因。最重要的是，存储体冲突会因间接访问的伪随机访问模式而加剧，降低可实现的 TCDM 带宽，进而降低 ISSR 吞吐量。此外，初始向量传输无法与有效计算重叠，因此加速比会随向量长度变化。我们的矩阵双缓冲和行分配方案也会造成一定的计算负载不均衡和开销。*DRAM* 模型的高延迟、有限吞吐量和调度虽然大多可通过双缓冲 DMA 传输较长的连续矩阵块来隐藏，但仍使加速比中位数降低1.8%，最高降低 6.9%。最后，我们观察到指令缓存缺失偶尔会导致停顿。尽管存在这些固有的并行化和扩展开销，SSSR 仍能实现高达46.8% 的整体 FPU 利用率，并以极小的面积成本带来显著的效率提升：配备 SSSR 的八 个核心，其计算吞吐量可与运行 BASE 的*39* 个核心相当。

sM×sV：图 5b 展示了不同向量密度下 16 位 SSSR sM×sV 的加速比，峰值为 5.9×。与 sM×dV 一样，我们观察到其趋势与所用的单核内核相似，但由于初始向量复制、指令缓存和 TCDM 停顿、工作分配不均以及 DRAM效率低下，加速比有所降低。对于较高的向量密度，

¶根据 SSSR sM×dV 中使用峰值加速比矩阵 mycielskian12 的 DMA 矩阵分块读取计算平均值。不包括片上互连。



(a) 集群 sM×∗V 在 DRAM通道带宽受限时的加速比。红点表示加速比已下降10%的情况。



(b) 集群 sM×∗V 的加速比随片上互连延迟的变化。

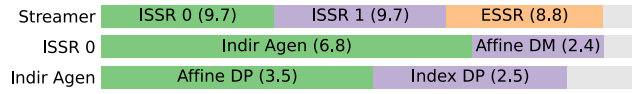
图6：SSSR Snitch集群的加速比对 DRAM通道带宽受限及片上互连延迟变化的敏感性。红色虚线表示带宽无限且延迟为零时的加速比。

性能主要受TCDM停顿和设置开销影响：峰值加速比仅比单核情况低6.9% 。由于每个核心只处理矩阵行的一个子集，并且DRAM响应时间存在差异，少非零调制总体上更加明显。随着向量密度 d_v 降低以及计算不平衡程度增加，DRAM效率低下问题也随之加剧：对于 $d_v = 30\,\%$，加速比平均仅比采用理想存储系统低0.40%，而对于 $d_v = 0.1\%$，则低16% 。观测到的最低加速比 1.1× 比单核情况更高，我们将其归因于代码大小显著更小，因此SSSR内核产生的指令缓存停顿更少。

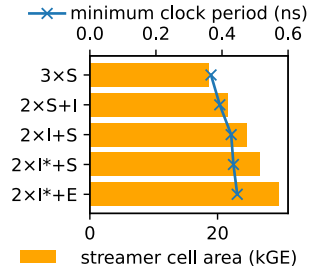
4.2.1 带宽和延迟敏感性

为评估 SSSR 加速比对带宽共享和互连扩展的敏感性，我们在DRAM吞吐量受限且互连延迟变化的条件下，重新运行集群 sM×dV 和 sM×sV。我们采用峰值 sM×dV 加速比矩阵 mycielskian12（$n_{nz} = 133$，4.3% 密度），以确保 DRAM 压力较高；对于 sM×sV，则采用1% 向量密度。在这两种情况下，我们还给出了理想加速比，其假设存储系统带宽无限且延迟为零。

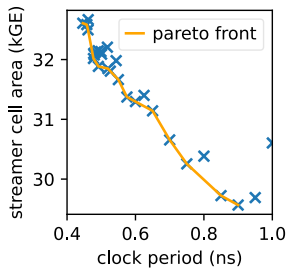
图6a展示了当可用DRAM通道带宽从其完整吞吐量3. 6 限制到0. 4Gb/s/pin时，加速比如何下降，这模拟了与其他总线代理共享通道的情况。我们保持单向互连延迟为16个周期。对于具有完整DRAM带宽的SSSR加速sM×dV，集群的平均吞吐量为 $R_T = 1.6\mathrm{Gb/s/pin}$；因此，我们预计加速后的sM×dV在 R_T 以上时为计算受限，在其以下时为存储器受限。事实上，当低于 R_T 时，加速比开始线性下降，直到达到 1× ，略低于0. 4Gb/s/pin；超过该点后，加速版本和基线变体均受存储器限制，性能相同。然而，我们还



(a) 默认流处理器及其主要子组件的面积分解（面积单位为 千门等效）。最右侧浅灰色部分表示剩余面积。



(b) 不同流式处理器配置的面积和最小时钟周期不同的流式处理器配置。



(c) 不同目标时钟下流处理器面积与时钟周期的关系。

图 7：GF12LP+ 综合得到的 SSSR 流式传输器面积和时序结果。S：SSR，I：ISSR，E：ESSR，I*：采用共享比较器的 ISSR。

观察到在高于 R_T时加速比有所下降；这是因为高于 R_T 的DRAM带宽可以掩盖由计算不平衡和DRAM调度导致的轻微吞吐量变化。对于sM×sV，我们观察到类似的趋势：在这种情况下，由于矩阵纤维处理速度更快， R_T 更高，而计算受限区域的加速比则因计算不平衡加剧而下降得略快。

图6b展示了在假设 DRAM 通道吞吐量充足的情况下，片上互连延迟增加时加速比如何下降。对于 sM×dV，单向延迟超过 64 个周期之前，加速比始终近乎不变；这一数值与下游 DRAM 子系统的延迟相当。这是因为集群的 DMA 引擎以双缓冲块的形式传输矩阵数据，每块长达数十KiB ，可使总线连续数百个周期处于饱和状态；只有当总访问延迟与这些传输所需的时间相当时，性能才会受到负面影响。对于 sM×sV，趋势相同，因为矩阵数据仍以较大的双缓冲块形式传输。然而，计算速率的波动和负载不均衡更大，因此延迟增加时，加速比损失略为明显。

总体而言，SSSR 在消除指令瓶颈方面的效率使稀疏工作负载更接近存储器受限状态；即使进入存储器受限区间后，它们仍能带来显著的加速比，只有当 DRAM 吞吐量降至常规通道带宽的十分之一以下时，其优势才会完全消失。尽管 sM×dV 和 sM×sV 对存储器的需求都很高，但能够重用矩阵数据的稀疏工作负载（例如矩阵乘法和迭代求解器）受存储器限制的程度要低得多。此外，我们的双缓冲数据传输方法表现出很强的延迟耐受性；除了 DRAM 延迟外，它还能弥补数百个周期的互连延迟，而加速比损失极小。

4.3 面积与时序

我们使用 Synopsys*Design Compiler* ，针对 GlobalFoundries 的 12LP+ FinFET 技术，在多种配置下综合 SSSR 流式传输器。

我们针对 25 ℃ 下的典型（TT）工艺角，供电电压为0.8 V。除非另有说明，时钟设为1 GHz，核心和紧耦合数据存储器接口的 IO 延迟分别约束为35% 和60%。在默认流处理器配置中，两个 ISSR 和一个 ESSR 各有四级数据 FIFO、17 位地址（覆盖 128KiB 紧耦合数据存储器），以及四个仿射循环层级。

图 7c 展示了默认流处理器的分层面积分布。每个 ISSR 占用9.7 千门等效，ESSR 占用8.8 千门等效，整个流处理器共30 千门等效 。ISSR 的面积主要由其地址生成器占据，其中只有37% 完全专用于索引流式传输。图 7b 对比了不同流处理器配置的面积和可实现的最小时钟周期。如果只需要间接寻址能力，每个 ISSR 仅需额外增加3.0 千门等效 （16 %）的面积。两个 SSR 之间的交集功能还需额外增加2.1 千门等效。增加并集能力后的完整 SSSR 流处理器相比基线方案增加11 千门等效（60%）的面积开销，且最小时钟周期仅从367ps 适度增加至446 ps，仍能轻松满足 Snitch 的 1GHz 时钟目标。图 7a 展示了不同周期目标下实现的最小周期和面积，表明随着时序压力增加，SSSR 流处理器的面积仍能平稳扩展。在集群级，为全部八个工作核心添加 SSSR 流处理器，相比仅提供常规 SSR，单元面积开销仅1.8 % 。

4.4 能量与功耗

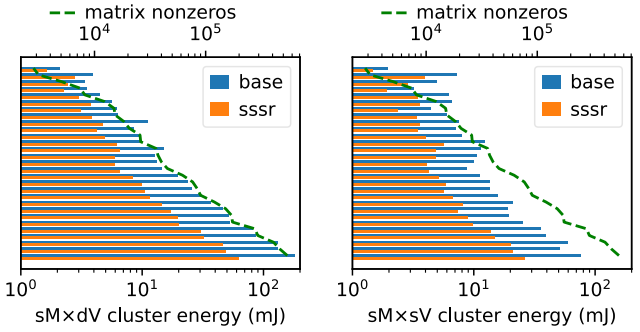
我们估算了采用 16 位索引 BASE 和 SSSR 内核、运行第 4.2 节工作负载的 Snitch 集群的功耗和总能量消耗。我们使用与第 4.2 节相同的互连和主存模型。目标工艺为 GlobalFoundries 的 12LP+Fin-FET 技术的 TT 工艺角，频率为 1GHz。我们使用 Synopsys *Fusion Compiler* 实现该集群，并使用 Synopsys *PrimeTime*估算低效能和高效能矩阵 cryg2500 与 cavity12 的功耗，随后根据 RTL 仿真中测得的组件利用率对动态功耗进行缩放。

图 8a 显示了每个矩阵使用两种内核变体时 sM×dV 的总能量。虽然 BASE 内核的集群中位功耗可预期地更低（195mW 对比285mW），但 SSSR 将每次融合乘加的最低能量从282 pJ 降至103pJ，能效最高提升 2.9× 倍。图 8b 显示了向量密度为1%时 sM×sV 的总能量。同样，SSSR 的中位功耗较高，但将每个矩阵非零元素的最低能量从107pJ 降至 43pJ ，能效最高提升 3.0× 倍。

5 相关工作

本节将我们的方法与众多先进的中央处理器、图形处理器和加速器解决方案进行比较，涵盖硬件和软件方案。我们在两张表中总结研究结果。表 2 汇总了所有讨论过的双精度浮点 sM×dV 软件实现所用的评估平台、矩阵格式，以及达到的最高峰值计算性能比例。表 3 比较了所有讨论过的硬件设计的功能、灵活性和架构成本。

标量处理器架构：配备 SSSR 的处理器可以视为一种扩展且专用的解耦处理器。



(a) sM×dV　　(b) sM×sV（1% 向量密度）

图 8：GF12LP+ 中 Snitch 集群工作负载的能耗估算。基准测试和数据与第 4.2 节中的结果相同。

访问/执行（解耦访问/执行）架构 [28]。解耦访问/执行架构结合了两个解耦处理器，一个负责内存访问，另一个负责执行，二者通过寄存器映射先进先出队列进行通信。SSSR 提供了一种轻量级访问处理器设计，并具备多项性能增强功能。它们利用多元素流作为抽象，从而改善解耦效果，最大限度地减少从主指令流中卸载的操作，并实现近乎理想的计算吞吐量和内存吞吐量。其作业类型定义了一种紧凑且正交的指令集，可最大限度地降低访问代码的复杂性，并支持轻量级实现。不同于最初的解耦访问/执行概念，SSSR 支持在不同寄存器上运行多个并发流。我们的低成本稀疏性扩展在间接流和联合流上保持高吞吐量，同时无需更传统访问处理器架构所需的硬件和簿记开销，从而无需大幅修改执行处理器即可高效实现通用稀疏计算。

向量处理器架构：ISSR 中的间接寻址类似于向量架构中的散布-聚集，这项技术近来已应用于超标量乱序处理器。Arm 的可扩展向量扩展 [29] 和英特尔 Knights Landing （KNL） [30] 支持单指令多数据散布-聚集。不过，与 ISSR 相比，单指令多数据向量长度较短且有对齐要求，这限制了散布-聚集在稀疏工作负载上的优势。无限向量扩展（UVE）[31] 放宽了向量长度限制，并引入了类似 ISSR 的向量寄存器绑定间接流。不过，它要求工作负载可向量化，且无法在标量粒度下高效运行。Gong 等人 [32] 扩展了向量乘加单元，以跳过无效计算并合并有效计算。虽然这与 SSSR 正交，但它只对未压缩数据中稀疏度较低且规律的情况有效。

面向标量和向量处理器的软件优化：众多软件解决方案旨在加速标量和向量处理器上的稀疏线性代数。CVR 矩阵格式 [33]旨在提高 x86 中央处理器上 sM×dV 的单指令多数据通道利用率和缓存效率。不过，它在 Xeon Phi 7250 上的双精度浮点计算利用率最高仅为0.69% 。Zhang 等人 [34]针对英特尔 AVX-512 扩展优化 sM×dV，提出了一种基于切片 ELLPACK (SELL) 的格式，使 Xeon Phi 7230 的双精度浮点计算利用率最高可达1.5% 。Regu2D [35] 利用自适应二维分区方案，将行排列为大小相近的组，并在必要时填充零以填满单指令多数据通道，

	Work	平台	格式	峰值浮点利用率
CPU	CVR [33]	Xeon Phi 7250	CVR	0。69 %
	Zhang 等人 [34]	Xeon Phi 7230	类似 SELL	1.5 %
	Regu2D [35]	Xeon Gold 6132	Regu2D	3.1 %
	Alappat 等人 [7]	A64FX	SELL-C-σ	4.7 %
GPU	Tsai 等人 [37]	V100	CSR	1。6 %
	Merrill 等人 [38]	K40	CSR	2.0 %
	TileSpMV [39]	A100	tile-adapt.	2。9%
	Tsai 等人 [37]	Radeon VII	CSR	3。2%
	cuSPARSE [40]	GTX 1080 Ti	CSR	17 %
	TileSpMV [39]	Titan RTX	tile-adapt.	27 %
	SSSR（我们的方法）	Snitch +SSSR	CSR	47 %

表 2：所讨论的 FP64 sM×dV 软件实现概览。

从而将其平台的双精度浮点峰值利用率提高至3.1%。Alappat 等人 [7] 使用 SELL-C-σ 矩阵格式对富士通 A64FX 上 sM×dV 的性能进行建模，并报告其性能最高可达130.9Gflop/s，即双精度浮点峰值计算能力的4.7 %。尽管如此，这仍比我们在配备 SSSR 的 Snitch 集群上取得的性能低 9.9× 。Z. Xie 等人[36] 为多核和众核处理器提出了一个 CSR 接口的 sM×sM 库，该库基于非零元素模式，通过深度学习模型选择最佳内部格式和算法。

流式指令集架构扩展：通用处理器扩展越来越多地利用流作为抽象来加速计算。Prodigy [8] 使用编译时推断的元数据来预取嵌套间接流。不过，它无法消除加载-存储或记账指令。SpZip [41] 使用解耦的流式单元读写压缩的不规则数据。这与 SSSR 读写基于纤维的格式类似，但 SpZip 不支持交集或并集，且一对取数器-压缩器会比 SSSR 流式传输器多占用116 千门等效 或 3.9× 的面积。Z. Wang 等人 [9] 提出了一种解耦的寄存器映射扩展，可加速与 ISSR 类似的仿射和间接内存流。不过，流需要显式步进，这严重限制了顺序执行核心可能获得的加速比，而且不支持交集和并集。SparseCore [6] 可加速索引流的交集、并集和减法，并提供专用功能单元。不过，它无法加速间接寻址，因此处理单边稀疏性时效率本质上较低。此外，在 Open-Cell 15nm 中，SparseCore 每个流单元占用$0.183\mathrm{mm}^2$ ，相当于两个配备 SSSR 的完整 Snitch CC。

图形处理器软件：从通用处理器转向更专用的架构以来，图形处理器一直是密集研究的对象，旨在提升其处理稀疏数据计算的效率。图形处理器上的单侧稀疏性通常通过专用算法和张量格式在软件 层面加速。Merrill 等人 [38] 采用基于归并的分解方法，提高了图形处理器上压缩稀疏行 sM×dV 的性能一致性，在 K40 图形处理器上达到峰值双精度浮点计算能力的2.0% 。英伟达通过*cuSPARSE* [40]为其图形处理器提供经过优化的稀疏线性代数内核。我们在 Jetson AGX Xavier（仅支持 FP32）和 GTX 1080 Ti 图形处理器（支持 FP32 和 FP64）上评估其 CUDA工具包10.0 中的压缩稀疏行 sM×dV 内核，使用*nvprof*对与 SSSR 相同的矩阵进行了 100 次调用的性能分析。对于 FP32，两款图形处理器的峰值流式多处理器（SM）占用率都很高，分别达到96 % 和87%，但峰值浮点利用率较低，

	Work						
CPU	SVE S/G [29]		✓		M	H	-
	KNL S/G [30]		✓		M	H	-
	UVE [31]	✓**	✓		H	H	72¶
	Gong 等人 [32]		✓	✓	M	L	31¶
	Prodigy [8]		✓		M	H	10¶
	SpZip [41]		✓		M	H	116
	Z. Wang等人 [9]		✓		H	H	-
	SparseCore [6]			✓	H	H	619
GPU	A100 [17]		✓		M	L	-
	Zhu等人 [42]		✓		L	M	12§
	Y. Wang等人 [10]			✓	L	M	157§
Accelerators	MatRaptor [43]			✓	L	H	-
	OuterSPACE [44]			✓	L	H	-
	Sadi等人 [45]		✓		L	H	-
	EIE [46]		✓	✓	L	H	-
	ExTensor [12]			✓	M	H	-
	Dadu 等人 [13]		✓	✓	M	H	-
	SISA [47]			✓	M	H	-
	SSSR（本文方案）	✓	✓	✓	H	H	30

L/M/H = 低/中/高。* 定性评估，依据操作数粒度、使用约束和开销。† 定性评估，依据可高效处理的范围和结构。‡ 根据作者提供的数据，不同工作使用的核心不同。** 仅为 gem5 模型，没有 RTL。¶ 仅为架构状态，按每位 6T=1.5GE 计算。§ 仅给出了制程节点；我们分别假设为 GlobalFoundries 的 22nm 和 12nm。

表 3：所讨论硬件设计概览。

活跃 SM 上的占比分别为2.1% 和0.75 %，这表明各个 warp 的线程并行度较低。GTX 1080 Ti 上的双精度浮点也显示出类似的 SM 占用率，但峰值浮点利用率明显更高，达到17%，这可能是因为每个 SM 上的双精度浮点核心较少。Tsai 等人 [37] 为英伟达和 AMD 的图形处理器以及多种格式创建了经过优化的 sM×dV 库内核。他们的压缩稀疏行内核通常优于厂商代码，在英伟达 V100 和 AMD Radeon VII 图形处理器上的峰值双精度浮点利用率分别达到1.6% 和3.2% 。Tile-SpMV [39] 在稀疏分块上使用针对 warp 优化的内核，分块采用给定格式或自适应选定的格式。它在 A100 和 Titan RTX 上的峰值双精度浮点利用率分别达到2.9% 和27 %；Titan RTX 同样是每个 SM 上双精度浮点核心少于单精度浮点核心。不过，这一数值比配备 SSSR 的 Snitch 集群低 1.7× 。Shi 等人 [48] 通过将矩阵编码为坐标编码分块的数组来加速 sM×dM，从而改善访问连续性和数据重用。然而，在 SuiteSparse 矩阵上，相较于 cuSPARSE，其 FP32 平均加速比仅为 1.7× ，不足以达到与配备 SSSR 的 Snitch 相当的浮点利用率。Karimi 等人 [49] 设计了一组内存感知矩阵格式，通过改善缓存利用率和线程级并行度来加速 sM×dV。尽管性能有所提升，其最佳格式在 SuiteSparse 矩阵上相较于 cuSPARSE 压缩稀疏行的 FP32 中位加速比仅为 1.8×（K40）和 3.5× （V100），仍不足以达到我们的利用率。

图形处理器上也存在一些稀疏-稀疏软件方法：Zachariadis 等人 [50]使用基于张量核心的外积方法和稠密 16×16 分块来加速 sM×sV，但灵活性较低，并且会在分块内产生不必要的计算损耗。Li 等人 [51] 使用机器学习模型，根据张量属性动态选择八种 sM×dV/sM×sV 方法之一，从而优化通用稀疏矩阵-向量乘法。该解决方案从根本上无法超过

所选现有方法的峰值性能，但也可进行调整，以提升其他系统的整体性能，包括配备 SSSRs 的系统。

图形处理器硬件：也有人提出了用于加速稀疏计算的图形处理器硬件 扩展。英伟达 A100 架构 [17]通过高效处理每四个值中最多两个零，引入了对单边结构化稀疏性的支持。SSSR 能够高效处理范围广得多的稀疏度（$\gg 50\%$），无需任何结构要求，并支持对完整紧耦合数据存储器进行随机访问。Zhu 等人 [42]提出了一种算法，并协同优化了张量核心的修改，以实现高效的稀疏-稠密神经网络推理。Y. Wang 等人 [10] 采用外积方法扩展 V100 张量核心，从而高效处理稀疏-稀疏矩阵乘法和卷积。然而，所使用的位图编码妨碍了在低稀疏度以上实现高效性能和内存扩展；而稠密输出则使整体方法能够通过散射式 ISSR 加以复现。

硬件加速器：稀疏加速器通常面向单一应用领域，例如深度学习（DL） [22]，或特定的线性代数操作和数据流。大多数加速器采用针对具体用途定制的数据格式和精度，因而无法胜任通用计算。值得注意的是，配备 SSSR 的通用处理器能够实现许多已采用的专用计算方案。MatRaptor [43]和 OuterSPACE [44] 分别采用按行和外积数据流加速 sM×sM；SSSR 能够高效处理这两种数据流。Sadi等人 [45] 通过按列切分矩阵并累加稀疏结果，加速大规模 sM×dV，从而改善向量局部性和重用；可由主机核心遍历各切片，同时由 SSSR 负责乘法和结果累加。Han 等人的 EIE [46] 通过将完整的共享权重模型保存在片上 SRAM 中，降低深度学习推理的动态随机存取存储器访问能量；SSSR 同样能够在采用码本编码值的稀疏数据上实现高利用率，但不受单一应用、调度或数据流的限制。ExTensor [12] 通过分层求取 CSF 张量的交集，避免冗余计算和数据传输；配备 SSSR 的处理器可在软件中处理较高阶轴，同时由流处理器对性能关键的主轴求交集，从而复现这一方法。Dadu 等人 [13] 提出了一种同时加速流连接和流式间接寻址的计算结构。然而，这种松耦合的空间计算方法在面积和可编程性方面代价显著。SISA [47] 引入了一套以集合为中心的 ISA，用于加速图挖掘算法。除了集合交集和并集，它还加速集合差、成员关系与大小查询，以及元素插入或删除。它同时支持索引集和位图集，并支持基于二分搜索的“跳跃”模式，以便对大小悬殊的集合进行索引连接。尽管 SISA 高度专用于图挖掘，且不面向单边稀疏性或“低复杂度”算法，但其许多额外功能都可加入 SSSR。不过，SISA 的架构成本仅通过与其他设计进行宽泛类比来讨论，因此难以评估此类权衡。

6 结论

在本工作中，我们扩展 SSR，以加速间接寻址、交集和并集，从而创建一种模块化、向后兼容的指令集架构扩展，可在基于稀疏纤维的格式（包括广泛使用的压缩稀疏行和 CSF）上高效执行通用稀疏线性代数。我们在 RISC-V Snitch 系统中评估了稀疏 SSR：展示了高效的单核稀疏-稠密和稀疏-稀疏线性代数内核，并在八核集群上并行化矩阵-向量乘积。

在使用 16 位索引的单个核心中，SSSR 的流式间接寻址使 sV×dV FPU 利用率峰值达到80% ，与经过优化的基线相比，sM×dV 加速比最高可达 $7.0\times$ 。SSSR 的交集和并集操作使稀疏-稀疏点积和向量加法在广泛的稀疏度范围内分别实现$3.0\text{-}7.7\times$ 和$5.4\text{-}9.8\times$ 的加速比，从而使 sM×sV 加速比最高达到 $6.3\times$。扩展至配备互连和动态随机存取存储器模型的 Snitch 集群后，在体现真实延迟、吞吐量和调度的情况下，使用 SSSR 的 sM×dV 整体最高可快 $4.9\times$ 倍且能效提高 $2.9\times$ 倍；对于向量密度为1%的情况，sM×sV 整体最高可快 $5.9\times$ 倍且能效提高 $3.0\times$ 倍。这些改进仅使每个核心的额外面积增加11kGE，即整个集群增加1.8% ；得益于流处理器的模块化设计，该开销还有望进一步降低。

与近期的中央处理器、图形处理器和加速器方案不同，SSSR 能够加速单侧和双侧稀疏计算，同时在数据流和操作数稀疏性方面保持灵活，并将面积影响降至最低。运行 sM×dV 的 Snitch 集群搭载 SSSR，其峰值浮点利用率比近期中央处理器和图形处理器软件分别高出 $9.9\times$ 和 $1.7\times$ 。

致谢

感谢审稿人提供的宝贵意见。本研究得到 ETH Future Computing Laboratory（EFCL）的支持，该实验室由华为技术有限公司捐资设立。此外，本研究还获得了欧洲高性能计算联合事业的资助，资助协议编号为 101034126（The EuropeanPilot）及特定资助协议编号 101036168（EPI SGA2）。

参考文献

[1] P. Scheffler、F. Zaruba、F. Schuiki、T. Hoefler 和 L. Benini，“用于高效稀疏-稠密线性代数的间接流语义寄存器架构”，载于*2021*年欧洲设计、自动化与测试会议及展览会（*DATE*），2021年，第1787–1792页。[2] Z. Zhang、Y. Xu、J. Yang、X. Li 和 D. D. Zhang，“稀疏表示综述：算法与应用”，*IEEE Access*，第3卷，第490–530页，2015年。[3] 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1:1–1:25页，2011年。[4] T. Hoefler、D. Alistarh、T. Ben-Nun、N. Dryden 和 A. Peste，“深度学习中的稀疏性：神经网络高效推理与训练的剪枝和增长”，机器学习研究杂志，第22卷，第241:1–241:124页，2021年。[5] L. Page、S. Brin、R. Motwani 和 T. Winograd，“PageRank引用排名：为Web带来秩序”，载于*WWW 1999*，1999年。[6] G. Rao、J. Chen、J. Yik 和 X. Qian，“SparseCore：用于稀疏计算的流式指令集架构与处理器专用化”，载于第*27*届美国计算机协会程序设计语言与操作系统架构支持国际会议论文集，2022年，第186–199页。[7] C. L. Alappat、J. Laukemann、T. Gruber、G. Hager、G. Wellein、N. Meyer 和 T. Wettig，“A64FX上的流式内核和稀疏矩阵-向量乘法性能建模”，*2020*年*IEEE/ACM*高性能计算系统性能建模、基准测试与仿真会议（*PMBS*），第1–7页，2020年。

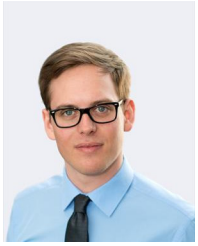
[8] N. Talati、K. May、A. Behroozi、Y. Yang、K. Kaszyk、C. Vasiladiotis、T. Verma、L. Li、B. Nguyen、J. Sun、J. M. Morton、A. Ahmadi、T. M. Austin、M. F. P. O'Boyle、S. A. Mahlke、T. N. Mudge 和 R. G. Dreslinski，“Prodigy：利用软硬件协同设计改善数据间接非规则工作负载的内存延迟”，载于2021年IEEE高性能计算机体系结构国际研讨会，2021年，第654–667页。[9] Z. Wang 和 T. Nowatzki，“面向通用处理器的基于流的内存访问专用化”，载于2019年第46届ACM/IEEE年度国际计算机体系结构研讨会，2019年，第736–749页。[10] Y. Wang、C. Zhang、Z. Y. Xie、C. Guo、Y. Liu 和 J. Leng，“双侧稀疏张量核心”，载于2021年第48届ACM/IEEE国际计算机体系结构研讨会，2021年，第1083–1095页。[11] A. Gondimalla、N. Chesnut、M. Thottethodi 和 T. N. Vijaykumar，“Sparten：用于卷积神经网络的稀疏张量加速器”，载于第52届IEEE/ACM国际微体系结构研讨会论文集，2019年，第151–165页。[12] K. Hegde、H. A. Moghaddam、M. Pellauer、N. C. Crago、A. Jaleel、E. Solomonik、J. S. Emer 和 C. W. Fletcher，“ExTensor：稀疏张量代数加速器”，载于第52届IEEE/ACM国际微体系结构研讨会论文集，2019年，第319–333页。[13] V. Dadu、J. Weng、S. Liu 和 T. Nowatzki，“通过利用常见的数据依赖形式实现通用加速”，载于第52届IEEE/ACM国际微体系结构研讨会论文集，2019年。[14] F. Schuiki、F. Zaruba、T. Hoefler 和 L. Benini，“流语义寄存器：在单发射核心中实现完全计算利用率的轻量级RISC-V指令集架构扩展”，IEEE计算机汇刊，第70卷，第212–227页，2021年。[15] V. Ramachandran 和 E. Shi，“面向多核处理器的数据无关算法”，载于第33届美国计算机协会算法与体系结构并行性研讨会论文集，2020年，第373–384页。[16] F. Zaruba、F. Schuiki、T. Hoefler 和 L. Benini，“Snitch：用于高效面积和能量执行浮点密集型工作负载的微型伪双发射处理器”，IEEE计算机汇刊，第70卷，第1845–1860页，2021年。[17] Nvidia 公司，“NVIDIA A100张量核心图形处理器架构。”[在线]。可访问：
h
t
t
p
s://
w
w
w.
n
v
i
d
i
a.
c
o
m/
c
o
n
tent/da
m/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf [18] S. C. Eisenstat、M. C. Gursky、M. H. Schultz 和 A. H. Sherman，“耶鲁稀疏矩阵软件包”，耶鲁大学计算机科学系，技术报告，1977年。[19] I. S. Duff、R. G. Grimes 和 J. G. Lewis，“稀疏矩阵测试问题”，ACM数学软件汇刊，第15卷，第1期，第1–14页，1989年3月。[20] S. Smith 和 G. Karypis，“采用压缩稀疏张量的张量-矩阵乘积”，载于第5届非规则应用：架构与算法研讨会论文集，2015年，论文5，第1–7页。[21] A. Pinar 和 M. T. Heath，“改善稀疏矩阵-向量乘法的性能”，ACM/IEEE SC 1999会议（SC'99），第30–30页，1999年。[22] S. Dave、R. Baghdadi、T. Nowatzki、S. Avancha、A. Shrivastava 和 B. Li，“机器学习模型稀疏与非规则张量计算的硬件加速：综述与见解”，IEEE论文集，第109卷，第1706–1752页，2021年。[23] M. W. Marcellin、M. A. Lepley、A. Bilgin、T. J. Flohr、T. T. Chinen 和 J. H. Kasner，“JPEG 2000中的量化概述”，信号处理：图像通信，第17卷，第84、2002年1月，73页，无。[24] G. Roth、J. M. Mellor-Crummey、K. Kennedy 和 R. G. Brickner，“在高性能Fortran中编译模板”，载于ACM/IEEE SC 1997会议（SC'97），1997年，第12–12页。[25] P. Stathis、D. Cheresiz、S. Vassiliadis 和 B. H. H. Juurlink，“稀疏矩阵转置单元”，载于第18届国际并行与分布式处理研讨会，2004年，论文集，2004年，第90–99页。[26] M. Zagha 和 G. E. Blelloch，“向量多处理器的基数排序”，载于1991年ACM/IEEE超级计算会议论文集，1991年，第712–721页。[27] M. Jung、C. Weis 和 N. Wehn，“DRAMSys：灵活的DRAM子系统设计空间探索框架”，IPSJ系统与LSI设计方法学汇刊，第8卷，第63–74页，2015年。[28] J. E. Smith，“访问/执行解耦计算机架构”，ACM计算机系统汇刊，第2卷，第289–308页，1984年。[29] 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。

[30] A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, 和 Y. Liu，“Knights Landing：第二代英特尔至强 Phi 产品”，IEEE Micro，第36卷，第2期，第34–46页，2016年。[31] J. M. Domingos, N. Neves, N. Roma, 和 P. Tom´as，“支持数据流传输的无限向量扩展”，载于2021年第48届ACM/IEEE国际计算机体系结构研讨会，2021年，第209–222页。[32] Z. Gong, H. Ji, C. W. Fletcher, C. J. Hughes, S. S. Baghsorkhi, 和 J. Torrellas，“Save：用于加速中央处理器上深度神经网络训练与推理的稀疏性感知向量引擎”，载于2020年第53届IEEE/ACM国际微体系结构研讨会，2020年，第796–810页。[33] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, 和 L. Zhang，“CVR：x86处理器上稀疏矩阵向量乘法的高效向量化”，载于2018年国际代码生成与优化研讨会论文集，2018年，第149–162页。[34] H. Zhang, R. T. Mills, K. Rupp, 和 B. F. Smith，“在PETSc中使用AVX-512实现向量化并行稀疏矩阵向量乘法”，第47届国际并行处理会议论文集，2018年。[35] X. Fei 和 Y. Zhang，“Regu2D：通过二维分区和规则排列加速英特尔处理器上的稀疏矩阵向量乘法向量化”，第50届国际并行处理会议论文集，2021年。[36] Z. Xie, G. Tan, W. Liu, 和 N. Sun，“面向多核与众核架构的基于模式的稀疏矩阵乘法库”，IEEE并行与分布式系统汇刊，第33卷，第159–175页，2022年。[37] Y. M. Tsai, T. Cojean, 和 H. Anzt，“AMD和英伟达图形处理器上的稀疏线性代数——竞赛开始了”，High Performance Computing，第12151卷，第309–327页，2020年。[38] D. Merrill 和 M. Garland，“基于归并的并行稀疏矩阵向量乘法”，载于SC16：国际高性能计算、网络、存储与分析会议，2016年，第678–689页。[39] Y. Niu, Z. Lu, M. Dong, Z. Jin, W. Liu, 和 G. Tan，“TileSpMV：一种用于图形处理器上稀疏矩阵向量乘法的分块算法”，2021年IEEE国际并行与分布式处理研讨会（IPDPS），第68–78页，2021年。[40] Nvidia 公司，“CUDA工具包10.0”。 [在线]。网址：https://developer.nvidia.com/cuda-toolkit [41] Y. Yang, J. Emer, 和 D. S´anchez，“SpZip：为非规则应用中的高效数据压缩提供架构支持”，载于2021年第48届ACM/IEEE国际计算机体系结构研讨会，2021年，第1069–1082页。[42] M. Zhu, T. Zhang, Z. Gu, 和 Y. Xie，“稀疏张量核心：面向现代图形处理器上向量级稀疏神经网络的算法与硬件协同设计”，载于第52届国际微体系结构研讨会。美国计算机协会，2019年，第359–371页。[43] N. Srivastava, H. Jin, J. Liu, D. H. Albonesi, 和 Z. Zhang，“MatRaptor：一种基于逐行乘积的稀疏-稀疏矩阵乘法加速器”，载于2020年第53届IEEE/ACM国际微体系结构研讨会，2020年，第766–780页。[44] S. Pal, J. Beaumont, D. hyeon Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. N. Mudge, 和 R. G. Dreslinski，“OuterSPACE：一种基于外积的稀疏矩阵乘法加速器”，载于2018年IEEE国际高性能计算体系结构研讨会，2018年，第724–736页。[45] F. Sadi, J. Sweeney, T. M. Low, J. C. Hoe, L. T. Pileggi, 和 F. Franchetti，“利用可扩展多路归并并行化高效处理大型高稀疏矩阵的稀疏矩阵向量乘法”，载于第52届IEEE/ACM国际微体系结构研讨会论文集，2019年。[46] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. Horowitz, 和 W. J. Dally，“EIE：压缩深度神经网络上的高效推理引擎”，载于2016年第43届ACM/IEEE国际计算机体系结构研讨会，2016年，第243–254页。[47] M. Besta, R. Kanakagiri, G. Kwasniewski, R. Ausavarungnirun, J. Ber´anek, K. Kanellopoulos, K. Janda, Z. Vonarburg-Shmaria, L. Gianinazzi, I. Stefan, J. G´omez-Luna, M. Copik, L. Kapp-Schwoerer, S. D. Girolamo, M. Konieczny, O. Mutlu, 和 T. Hoefler，“SISA：面向存内计算系统图挖掘的以集合为中心的指令集架构”，MICRO-54：第54届IEEE/ACM国际微体系结构研讨会，2021年。[48] S. Shi, Q. Wang, 和 X. Chu，“使用定制稀疏存储格式在图形处理器上高效进行稀疏矩阵与稠密矩阵乘法”，载于2020年第26届IEEE国际并行与分布式系统会议（ICPADS），2020年，第19–26页。[49] E. Karimi, N. B. Agostini, S. Dong, 和 D. R. Kaeli，“Vcsr：一种高效的图形处理器内存感知稀疏格式”，IEEE并行与分布式系统汇刊，第PP卷，第1–1页，2022年。[50] O. Zachariadis, N. Satpute, J. G´omez-Luna, 和 J. Olivares，“利用图形处理器张量核心加速稀疏矩阵乘法”，Computers & Electrical Engineering，第88卷，文章编号106848，2020年。

[51] M. Li, Y. Ao, 和 C. Yang，“用于不同稀疏度输入向量的图形处理器自适应稀疏矩阵向量乘法/稀疏矩阵稀疏向量乘法”，*IEEE*并行与分布式系统汇刊，第32卷，第1842–1853页，2020年。



Paul Scheffler 于2018年和2020年分别获得苏黎世联邦理工学院电气工程与信息技术理学学士和理学硕士学位。目前，他正在Benini教授的数字电路与系统组攻读博士学位。他的研究兴趣包括稀疏和不规则工作负载的硬件加速、片上互连、众核架构以及高性能计算。



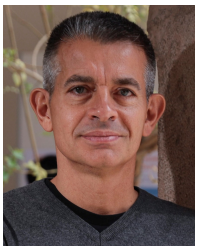
Florian Zaruba 于2014年获得维也纳工业大学理学学士学位，并于2017年和2021年分别获得苏黎世联邦理工学院的理学硕士和博士学位。他是Axelera AI的系统架构师。他的研究兴趣包括超大规模集成电路设计和高性能计算机体系结构。



Fabian Schuiki 于2014年、2016年和2021年分别获得苏黎世联邦理工学院（ETHZ）电气工程理学学士、理学硕士和博士学位。他是SiFive公司的硬件编译器工程师。他的研究兴趣包括跨精度计算，以及近存和内存内处理。



Torsten Hoefler是瑞士苏黎世联邦理工学院的计算机科学教授。他在印第安纳大学获得博士学位。Hoefler博士的研究兴趣主要涉及以性能为中心的软件与硬件开发。他是美国计算机协会会士，也是欧洲科学院成员。



Luca Benini 担任苏黎世联邦理工学院数字电路与系统讲席教授，同时也是博洛尼亚大学的正教授。他获得了斯坦福大学的博士学位。Benini博士的研究兴趣包括高能效并行计算系统、智能传感微系统和机器学习硬件。他是美国计算机协会会士，也是欧洲科学院成员。