arXiv:2303.02471v2 [cs.DC] 2 Jun 2023

利用 RISC-V 向量指令优化稀疏通用矩阵乘法

Valentin Le Fèvre 巴塞罗那
超级计算中心 巴塞罗那，西班牙
valentin.lefevre@bsc.es

Marc Casas 巴塞罗那超级
计算中心 巴塞罗那，西班牙
marc.casas@bsc.es

摘要

稀疏通用矩阵乘法（SpGEMM）$C = A \times B$ 是一种基础例程，广泛应用于机器学习和图分析等领域。尽管 SpGEMM 十分重要，但在向量架构上高效执行 SpGEMM 仍是一个相对未被深入研究的课题。在此类架构上运行 SpGEMM 的最新算法基于稀疏累加器（SPA）方法；由于该方法逐列计算 C，因此对于每列包含几十个非零系数的稀疏矩阵而言，其效率相对较高。然而，当处理每列仅包含少量非零系数的矩阵时，最先进方法在计算稀疏通用矩阵乘法内核时无法充分利用长向量架构。

为解决这一问题，我们提出了带排序的 SPA 并行算法（SPARS），该算法并行计算多个 C 列，并采用其他优化措施；同时还提出了 HASH 算法，该算法使用大小可动态调整的哈希表存储中间输出值。为了兼顾 SPA 在相对稠密的矩阵块上的效率，以及 SPARS 和 HASH 在极稀疏矩阵块上的高性能，我们提出了 H-SPA(t) 和 H-HASH(t)，它们可在不同算法之间动态切换。在从 SuiteSparse 矩阵集合 [19]中选取的 40 个稀疏矩阵上，H-SPA(t) 和 H-HASH(t) 相对于 SPA 的平均加速比分别为 1.24× 和 1.57× 。对于其中最稀疏的 22 个矩阵，H-SPA(t) 和 H-HASH(t) 的平均加速比分别为 1.42× 和 1.99× 。

关键词：向量处理器，稀疏矩阵，稀疏乘法，RISC-V，稀疏通用矩阵乘法

1 引言

稀疏通用矩阵乘法（Sparse GEneral Matrix-Matrix multiplication）是稀疏线性代数中最基础的例程之一。它广泛应用于稀疏线性求解器 [10], 、图分析 [9]或机器学习 [18]。该例程使用两个稀疏矩阵 A 和 B 来计算 $C = A \times B$。由于两个输入矩阵都是稀疏矩阵，稀疏通用矩阵乘法呈现出比稀疏矩阵多向量乘法（SpMM）或稀疏矩阵-稠密矩阵乘法（SpMDM）更加不规则的内存访问模式，其中 B 是一个稠密矩阵。在众核架构 [4, 5], 、GPU [6, 39, 40], 或二者上 [20],高效执行稀疏通用矩阵乘法，已经得到了广泛研究。

支持向量指令集架构（ISA）的硬件设计，例如 ARM 可扩展向量扩展（SVE） [44] 或 RISC-V“V”向量扩展 [14], ，正变得越来越具有吸引力 [41]。采用 Aurora-SX 处理器等长向量处理器的商用产品 [48] 证实了这一趋势：面向高性能计算，使用越来越长的向量单元，这些向量单元能够在单条指令中处理多个元素。单条向量指令所管理的元素数量称为向量长度 （VL）。VL 可以动态设置为 1 到寄存器可存储的最大元素数量之间的任意值。

尽管向量处理器对于高性能计算具有重要意义，但最新的面向此类架构的稀疏通用矩阵乘法算法 [31], 采用了基于稀疏累加器 （SPA）的方法 [21], ，仍存在显著局限。该算法在处理每列含有数十个非零系数的矩阵时效率较高，但逐列计算 C，因此在处理列极为稀疏的矩阵时，无法充分利用长向量架构。

为克服这一局限，我们提出了两种面向长向量架构的稀疏通用矩阵乘法算法：首先是带排序的 *SPA* 并行算法（*SPARS*）算法，它能够在处理超稀疏矩阵时发挥大向量长度的潜力，并使用稠密数组保存输出矩阵的中间值。相较于先前研究 [31] ，SPARS 的主要创新在于采用了一种基于排序过程的全新分块算法。其次，我们提出了*HASH* 算法，它使用哈希表而非大型稠密数组来存储输出矩阵的中间值。HASH 采用与 SPARS 相同的分块机制，并使用一种新颖的方法动态调整哈希表的大小。此外，为了兼顾 SPARS 和 HASH 在处理超稀疏矩阵时的效率，以及最先进的基于 SPA 的算法在处理每列含有数十个非零系数的矩阵时的良好性能，我们提出了两种新算法：H-SPA(t)和 H-HASH(t)。它们采用混合执行模型 *i.e.*：H-SPA(t)在 SPA 和 SPARS 之间动态切换，而 H-HASH(t)则在 SPA 和 HASH 之间动态切换。总而言之，本文相较于最先进方法作出了以下贡献：

- 我们提出了 SPARS 和 HASH 算法，能够在长向量架构上高效执行稀疏通用矩阵乘法运算

以处理超稀疏矩阵。SPARS 使用大型数据结构保存输出矩阵的中间值，而 HASH 则采用一种更复杂的、基于哈希表的方法。

- 我们提出混合算法 H-SPA(t) 和 H-HASH(t)，分别将 SPARS 和 HASH 与 SPA 动态结合。该组合由每个矩阵块的稀疏结构决定。
- 我们针对具有八个向量通道、支持最大向量长度为 256 个双精度元素的向量处理器架构，评估了 SPA、SPARS、H-SPA(t)、HASH、H-HASH(t)，以及一种基于排序的算法（ESC [17,46]）。在 SuiteSparse 矩阵集合 [19] 的 40 个稀疏矩阵上，H-SPA(t) 和 H-HASH(t) 相对于 SPA 的平均加速比分别为 1.24× 和 1.57× 。对于其中最稀疏的 22 个矩阵，H-SPA(t) 和 H-HASH(t) 的平均加速比分别为 1.42× 和 1.99× 。

2 在向量架构上运行稀疏通用矩阵乘法的方法

本节介绍一些先前提出的稀疏通用矩阵乘法运算方法。第 2.1 节介绍 Gustavson 方法 [23]。第 2.2 节说明稀疏累加器（SPA）方法 [21] 及其向量化实现，该实现基于先前研究 [31]。

2.1 Gustavson 方法

Gustavson 方法 [23] 用于按列计算两个稀疏矩阵 A 和 B 的乘积，即计算 $C = A \times B$，其中A是大小为 $m_A \times n_A$的矩阵，B是大小为 $n_A \times n_B$的矩阵。我们使用压缩稀疏列（CSC）格式 [42]表示这些矩阵。例如，对于具有NNZ个非零元素的矩阵 B，CSC 格式使用 3 个数组存储该矩阵：

- 一个大小为 NNZ的 数值 数组，用于存储每个非零元素的值；
- 一个大小为 NNZ 的行_索引 数组，用于存储每个非零元素对应的行索引；。
- 一个大小为 $n_B + 1$ 的列_指针 数组，其中包含对应于前两个数组中每列起始位置的索引。第一个单元始终为 0，最后一个单元始终为 NNZ。

采用 CSC 格式时，同一列中$values$ 和$row_indices$ 中元素的存储顺序并不重要，但各列必须按一定顺序存储，以确保$column_pointers$[i+1]-$colum_pointers$[i]始终对应第 i 列中的非零元素数量。该方法将每个 C 列视为若干A 列的线性组合。更正式地说，要计算 C的第 j 列，算法将每个非零元素 B_{kj} 与 A 的第k 列相乘，并累加这些值。图 1 给出了一个

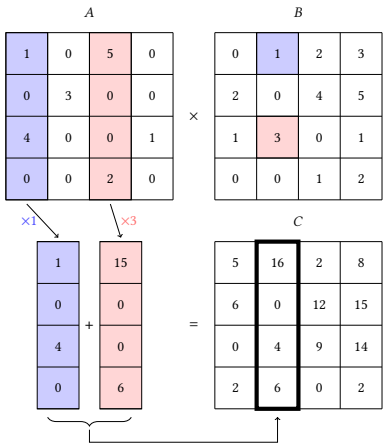


图 1. Gustavson 算法计算第二列。

采用 4×4 个矩阵的 Gustavson 方法示例。该示例通过以下步骤计算 C 的第二列：i）将 A 的第一列与数值为 1 的 B_{01} 系数相乘，并将 A的第三列与 B_{03} 系数相乘；ii）将第一步得到的两列相加，得到最终的 C 列。这是一种高度并行的方法，因为每个 C 列都可以独立于其他列进行计算。其主要问题是负载不均衡，因为不同列的计算所需执行的运算次数不一定相同。请注意，也可以对行而非列应用相同的方法，此时矩阵采用压缩稀疏行（CSR）格式表示。

Gustavson 方法有多种变体，它们采用不同的方法计算最终的 C 列。当计算出一个元素 $A_{ik} \times B_{kj}$时，需要判断它是否会在C的第 j 列中产生一个新的非零元素，或者是否需要与已计算出的值$A_{il} \times B_{lj}$累加（适用于任意 $l \neq k$）。这项操作可以通过哈希表 [6, 20], 、对键进行排序与合并 [11, 15], ，或使用稠密向量来完成；稠密向量是一种用于存储所有中间累加值 [21, 40]的稠密数据结构。第 3 节提出了两种新算法，分别基于稠密向量和哈希表累加器，以便在向量处理器上高效执行稀疏通用矩阵乘法。

2.2 稀疏累加器（SPA）

本节介绍稀疏累加器算法（SPA） [21]。用于计算 $C = A \times B$ 的顺序 SPA 算法见算法 1。为简单起见，我们给出一段计算输出 C 矩阵第 j 列的伪代码，不过这段伪代码可以轻易推广到任意数量的列。第一步是创建三个数组，其大小与 C 的行数相同：SPA_值、SPA_标志和 SPA_索引。

算法 1： 顺序 SPA 算法

输入： 列的索引 j

```
1  将 SPA_值、SPA_标志和 SPA_索引初始化为零;
2  计数器 ← 0;
3  对于 列 j 中的元素 B_kj （属于 B）执行
4      对于每个 元素 A_ik 在列 k 中 A 执行
5          临时_值 ← A_ik × B_kj;
6          SPA_值[i] +=临时_值;
7          如果 SPA_标志[i] == 0 则
8              SPA_标志[i] ← 1;
9              SPA_索引[计数器] ← i;
10             递增计数器;
11 对于 i ← 0 至 计数器 − 1 执行
12     追加(C.值, SPA_值[SPA_索引[计数器]]);
13     追加(C.行_索引, SPA_索引[计数器]);
14 C.列_指针[j+1] ← C.列_指针[j]+计数器;
```

SPA_值数组存储 C 的非零系数的部分值，SPA_标志用于说明 SPA 是否已为 C 的第 j 列中的某个系数计算出非零贡献，SPA_索引则存储已获得非零贡献的 C 元素的索引。SPA 将这三个数组以及非零元素计数器初始化为零（算法 1 的第 1、2 行）。

SPA 遍历 B的 j 中的所有非零元素，并将每个非零 B_{kj} 系数与 A的第 k 列中所有与之对应的非零值 A_{ik} 相乘。将非零元素 A_{ik} 和 B_{kj} 相乘后（算法 1 第 5 行），SPA 算法会把值 $A_{ik} \times B_{kj}$ 加到 SPA_值的第 i个位置（算法 1 第 6 行）。然后，如果 SPA_标志[i] 为零，SPA 就将该值改为 1，以表示j的 C 矩阵第 i列的第 i个位置存在非零贡献。此外，SPA 会将行索引 i 存储到 SPA_索引中的计数器位置，并递增计数器的值，以表示已生成新的非零输出系数（算法 1 第 9、10 行）。反之，当 SPA_标志[i] 的值为 1 时，由于 j的 C 矩阵第i列中对应位置已经有非零贡献，SPA 不会更新数组 SPA_索引和变量计数器。最后，SPA 完成对 B j列中所有非零元素的遍历后，算法会更新 C 中第 j 列对应的非零值。变量计数器包含将要添加到 C.values 和 C.row_索引数据结构中的非零元素总数。

我们采用最新的方法对 SPA 进行向量化，以便在长向量架构 [31]上运行稀疏通用矩阵乘法。写依赖是实现高效向量化的关键因素。只有当存在两个索引k和 l，使得 $A_{ik}B_{kj}$和 $A_{il}B_{lj}$均不等于 0 时，才会出现写依赖。然而，对于一个非零 B_{kj}，与 A 的第 k 列中各元素进行的所有乘法运算都具有不同的行索引，因此在稠密向量中产生的写操作完全相互独立。因此，可以按如下方式对 SPA 进行向量化：遍历 B 某一列中的所有非零元素，并对每个元素使用向量指令加载 A的一整列。随后读取 SPA_值和 SPA_标志的当前值，再将 A 的一列与 B中的当前非零值相乘、累加并写回。最后，对标志进行测试，并将新的索引连续存储到 SPA_索引中。算法 2 展示了这种 SPA 向量化方法。所有名称以 v 开头的变量都存储在向量寄存器中。我们在每行右侧说明相应的向量指令。

算法 2： 向量化 SPA 算法

输入： 列的索引 j

```
1  使用以下内容初始化 SPA_值、SPA_标志和 SPA_索引零:
2  对于 列 j 中的每个元素 B_kj，属于 B 执行
3      向量长度 ←A.列_指针[k+1]- A.列_指针[k];   // 设置向量长度
4      vA ← 负载列 k，属于 A;                    // 连续向量加载
5      vSPA ← 加载以下内容对应的元素 SPA_值;     // 索引向量加载
6      vSPA_标志 ←加载以下内容对应的元素 SPA_标志;  // 索引向量加载
7      vSPA ←vSPA +vA · B_kj;                    // 向量融合乘加
8      将 vSPA 存回;                             // 索引向量存储
9      将 vSPA_ 标志与 0 比较以创建掩码;          // 创建向量掩码
10     使用掩码存储 vSPA_ 索引;                  // 索引向量存储
11 将列存储为稀疏格式;
```

关于算法 2，有两点需要注意：首先，尽管算法 2 假设最大向量长度大于每个 A 列中的非零元最大数量（第 3 行），但稍作修改即可处理一般情况，即 A 的一列中非零元素数量可能超过可用向量长度的最大值。为支持这种情况，算法只需对同一列迭代处理数次，只需调整

要加载的 A 元素的索引（算法 2 第 4 行）。其次，掩码向量化存储虽然可用于第 11 行存储非零值和索引，但不会将未掩码元素连续存储。可以采用多种替代方案来存储未掩码的非零元素，例如使用向量指令仅将未掩码元素存入向量寄存器，然后执行常规向量存储；或者计算未掩码元素的索引，并对这些元素应用索引存储指令。

向量化 SPA 算法仅利用跨越 A 列的 SIMD 并行性，因为它逐个处理 B 非零元素。由于稀疏矩阵通常每列的非零元素数量较少，向量化 SPA 算法可能无法充分利用具有较长向量长度的向量处理器。第 5 节展示的实验结果证实了 SPA 的这一局限性。要充分利用长向量架构，需要一种能够同时处理多个 A 和B 非零元素的新算法。

3 在向量架构上使用超稀疏矩阵执行稀疏通用矩阵乘法的三种方案 3.1 带排序的 SPA 并行算法（SPARS）

带排序的 SPA 并行算法（SPARS）计算稀疏通用矩阵乘法内核 $C = A \times B$，即使对于每列仅包含少量非零元素的高度稀疏矩阵，它也能充分利用长向量架构。SPARS 利用了不同 B 列之间的计算不存在依赖关系这一特点；这一稀疏通用矩阵乘法内核特性此前已在分布式内存架构中得到利用 [13]。SPARS 将每个向量寄存器元素用于一列，因此能够同时处理最多达 VL 所允许数量的列。SPARS 将矩阵B 划分为多个块，并按顺序处理这些块。每个块中的列数决定向量长度。本节介绍一种分块算法，用于确定每个块的大小。为避免写依赖，每个 B 列都有自己的稠密向量，即 SPA_值、SPA_标志和 SPA_索引现在都是大小为 $m_A \times$ VL 的稠密矩阵，其中 VL 表示向量长度，而 m_A 是 A 和 C的行数。

算法 3 展示了 SPARS 处理一个列块的伪代码。它计算从列 j 开始的 VL C 列。对于某些向量指令，我们使用 i 索引来指定每个标量元素的计算内容。因此，提及 i 的每一行都应理解为隐式的 对所有i从 0 到 $VL-1$。与 SPA 类似，SPARS 从 B和 A矩阵中加载值（算法 3 的第 7 行和第 8 行），将它们相乘并累加到存储 SPA_ 值数据的寄存器中（第 11 行）。SPARS 使用一个向量寄存器（vIndices_B）存储 B 列非零元的索引。SPARS 用每列第一个元素的索引初始化 vIndices_B，并判断是否已到达某一列的末尾 B 列；当索引到达下一列的起始位置时，即表示到达当前列末尾。下一列的起始索引存储在 vEnd_B 寄存器中。SPARS 还使用 vCounter_A 寄存器跟踪 A 列中的位置；当 SPARS 到达相应 A 列的末尾时，该寄存器会重置（第 16 行）。此时，SPARS 通过增大 vIndices_B 中的值，移至 B 列中的下一个非零元素。最后，SPARS 使用与 SPA 相同的向量化算法存储这些列。

算法 3：向量化SPARS算法。输入：第一列的索引 j和VL，要处理的列数 1 初始化 SPA_值、SPA_标志和SPA_索引为零；2 vIndices_B[i] ← B.column_指针[j+i]；// 连续向量加载 3 vEnd_B[i] ←B.column_指针[j+i+1]；// 连续向量加载 4 vCounter_A[i] ← 0；// 向量广播 5 创建掩码 vMask，以保留满足 vIndices_B[i] < vEnd_B[i] 的元素；// 创建向量掩码 6 当 vMask 中存在未掩码元素时执行 7 vB ← 加载 vIndices_B 所指定索引处的 B 元素；// 索引向量加载 8 vA[i] ← 加载对应列中第 vCounter_A[i] 个元素，该列位于 A 中；// 索引向量加载 9 vSPA ← 加载 SPA_values 中的对应元素；// 索引向量加载 10 vSPA_flags ← 加载 SPA_flags 中的对应元素；// 索引向量加载 11 vSPA ← vSPA + vA · vB；// 向量融合乘加 12 将 vSPA 存回；// 索引向量存储 13 将 vSPA_flags 与 0 比较，以创建掩码 vFlags；// 创建向量掩码 14 使用 vFlags 存储 vSPA_indices；// 索引向量存储 15 如果 vA[i] 是该列的最后一个元素，则 16 vCounter_A[i] ← 0；// 向量广播 17 vIndices_B[i] ← vIndices_B[i]+1；// 向量加法 18 否则 19 vCounter_A[i] ←vCounter_A[i]+1；// 向量加法 20 更新 vMask；// 创建向量掩码 21 将VL列存储为稀疏列；

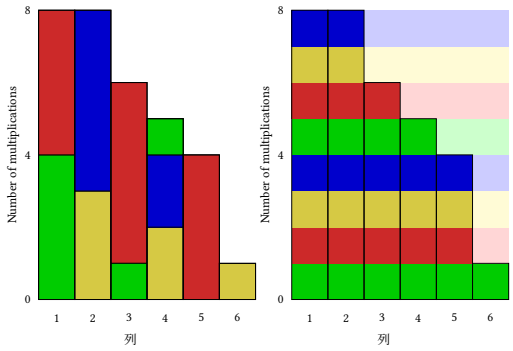


图 2. SPA（左）与 SPARS（右）的视觉对比。

图 2 展示了计算 $C = A \times B$时 SPA 和 SPARS 算法的图形表示。x 轴表示 C 矩阵的列，y 轴表示计算每个 C 列中的非零元素所需的乘法运算次数。彩色块表示由同一条向量指令执行的计算。SPA 使用向量指令垂直计算每个 C 列：加载 A 的列，并将其中的非零元素与对应的 b_{ij} 系数相乘；而 SPARS 则通过在每一步中对每列执行一次乘法，在 C 矩阵中沿水平方向推进。SPARS 的向量长度由其同时处理的列数定义。例如，在图 2 中，SPARS 同时处理 6 列，因此 VL = 6。SPARS 利用长向量指令长度的潜力取决于图 2 中彩色区域的总面积。纯色区域（左侧和底部）表示有用工作，而半透明区域（顶部和右侧）对应于 C 矩阵中具有特定稀疏模式的区域，这些模式迫使 SPARS 使用掩码，并导致向量指令长度中相当大的一部分未被使用。

使用较小的块大小会缩短 SPARS 触发的向量指令长度，并增加指令总数。不过，较小的块大小有助于减少向量指令中的掩码元素数量。相反，增大块大小会使每条指令包含更多掩码元素，但能减少指令数量并增大向量长度。例如，如果将图 2 中的矩阵划分为两个各含 3 列的块，向量长度将从 6 降至 3，而 SPARS 的指令数量将从 8 增至 13，其中第一个块需要 8 条指令，第二个块需要 5 条。此外，这样可以避免大小为 3×3 的右上角彩色方块，从而减少每条指令中的掩码元素数量。总体而言，SPARS 在处理较大的块且无需大量使用掩码时性能良好。如果相邻的矩阵C列所含非零系数数量差异很大，这可能会严重影响 SPARS 的性能，因为这些差异会迫使 SPARS 使用掩码。我们提出了两种优化方法来提升 SPARS 的性能：排序，根据矩阵列的计算负载对其排序，尽可能使连续列趋于一致；以及一种分块算法，以便使用尽可能大的块大小，同时避免块内各列之间存在较大差异。

排序：作为预处理阶段，我们按照计算负载从大到小对 B 的列进行排序。此排序的目的是减少向量通道之间的负载不均衡。为了确定计算 B 的每一列 j 所需的运算次数，我们作如下推理：每个非零元素 B_{kj} 都必须与 A中 k 列的所有非零元素相乘。设 A中 k 列的非零元素数量为 Z_k^A，则可得出列 j所需的运算次数 Op_j：

$$Op_j = \sum_{\substack{k \\ B_{kj} \neq 0}} Z_k^A$$

然后，我们按照Op_j的降序对 B 的所有列进行排序，$1 \le j \le n_B$，也就是说，我们对矩阵 B应用置换 P。现在，算法的结果是矩阵 $C \times P$，而不是 C；只需将 P 保存在内存中，就可以轻松还原结果。我们实际上并未重新排列矩阵 B，而只是调整了要访问的列的顺序。使用索引加载即可按正确顺序获取 B 的非零元素（算法 3，第 7 行）。矩阵 A 保持不变，无需额外的预处理。

分块算法：我们的算法以向量长度表示块大小，也就是说，每个块都使用与块大小相对应的向量长度进行处理。对于已处理 j 列的情况，我们提出以下算法来动态确定块大小：

1. 从 $j+1$ 到 $j_2 = j + b_{min}$ 开始处理 b_{min} 列；2. 当 Op_{j_2+1} 等于 $\max_{j+1 \le j' \le j_2} Op_{j'} = Op_{j+1}$ 时，递增 j_2；3. 如果 j_2 达到 $j + b_{max}$ 或 n_B，则停止循环。

使用该算法时，$j+1$ 到 j_2 列将在同一个块中处理，即块大小为 $j_2 - j$。该算法引入了两个新参数：b_{min} 是最小块大小，b_{max} 是最大块大小。我们的算法确保同一块中的所有列都恰好需要执行相同数量的运算，除非满足此条件的列少于 b_{min} 列。这两个参数的影响将在第 5 节中讨论。

复杂度分析：假设 A 的每一列恰好有 Z^A 个非零元素，就可以轻松分析 SPA 和 SPARS 的复杂度。 B的第 j 列有 Z_j^B 个非零元，并且我们假设 B 列已排序，*i.e.*、$Z_j^B >= Z_{j+1}^B$，$\forall j = 1, \ldots\ldots, n_B - 1$。 B共有 NNZ^B 个非零元。最后，我们假设 VL 可以整除列数 n_B。基于这些假设，可以推导出每种算法所需的总迭代次数，以及处理的矩阵系数总数，其中这些系数可能

为零或非零。零元素由掩码屏蔽；由于两种算法计算的是同一个C矩阵，因此它们处理的非零元素总数相同。SPA 执行 NNZ^B 次主循环迭代，共处理 $Z^A NNZ^B$ 个元素。SPARS 的总迭代次数为 $n_B/\mathrm{VL}\sum_{k=0}^{\quad} Z^A Z^B_{1+k}\mathrm{VL}$，其上界为 $Z^A(\ ^{NNZ^B}\mathrm{VL} + Z^B_1 - Z^B_{n_B})$；它处理 $n_B/\mathrm{VL}\sum_{k=0}^{-1}$ $\mathrm{VL}Z^A Z^B_{1+k}\mathrm{VL}$ 个元素，上界为 $Z^A(NNZ^B + \mathrm{VL}(Z^B_1 - Z^B_{n_B}))$。在简化情形下，如果 B 的所有列都具有相同数量的非零元，即 $Z^B_1 = Z^B_{n_B}$，两种算法处理的元素数量相同。在这种情况下，SPA 和 SPARS 的迭代次数之差取决于因子 $^{Z^A}\mathrm{VL}$。当 Z^A 较小时，SPARS 所需的迭代次数少于 SPA；也就是说，SPARS 更适用于每列非零元数较少的超稀疏矩阵。相反，当 Z^A 较大时，SPA 计算稀疏通用矩阵乘法内核所需的迭代次数少于 SPARS；也就是说，SPA 更适用于每列非零元数相对较多的矩阵。第 5.2 节通过实证验证了这些理论观察结果。

3.2 HASH 算法

本节提出 HASH，这是一种在向量处理器上计算稀疏通用矩阵乘法的算法，它依靠哈希函数来存储 C 输出矩阵的中间值。相比之下，SPA 和 SPARS 依靠稠密结构来累积这些中间值。HASH 将其哈希表定义为两个大小为 H 的数组：HASH_values 和 HASH_indices。前者包含给定列中非零元素的值，后者存储对应的行索引。HASH 不会直接将中间乘积存储在由行索引 i指定的位置，而是根据哈希函数将值和索引存储在以下位置：$h(i) = (i * c) \bmod H$，其中 c 为常数。然而，当 $h(i_1) = h(i_2)$ 且 $i_1 \neq i_2$ 时，可能会发生冲突。在这种情况下，HASH 使用线性探测解决冲突*i.e.*，即将值和索引存储在哈希表中下一个可用的单元；必要时从头开始查找。由于冲突会产生写依赖，且只能通过顺序访问哈希表来解决，因此，要对 HASH 进行向量化，每个 B列都需要有自己的哈希表，并在每一步累积中间乘积。HASH 的伪代码与算法 3 非常相似，只是将对 3 个 SPA数组的访问替换为哈希函数计算、冲突处理以及对哈希表的访问。B 列按负载 Op_j 递减排序，块大小也使用与第 3.1 节相同的算法确定。这两种优化的动机与 SPARS 算法相同。HASH 相较于先前研究中用于向量稀疏通用矩阵乘法的哈希函数的创新之处 [31] 体现在两个方面：i）HASH 使用与 SPARS 相同的分块和排序算法；ii）HASH 会动态调整其哈希表的大小以优化数据传输，这得益于排序预处理。

哈希表大小自适应：先前研究在稀疏通用矩阵乘法计算中使用哈希函数时，会在执行开始时将哈希表大小设为 2 的幂 [20, 31],，使其严格大于单列中间乘积的最大数量。对于向量架构，在整个稀疏通用矩阵乘法执行过程中使用相同的最大值，可能会导致性能严重下降。因此，我们在处理不同列的过程中动态调整哈希表大小，依据的是单列中间乘积的最大数量 Op_j，该数量在预处理时间内确定。我们从初始大小 $H = 2^k$ 开始，其中 k 由不等式 $2^{k-1} \leq \max_j Op_j <$ 定义

2^k 。j 索引对应第一个B 块中的所有列。对于后续的列块，可能出现 $\max_j Op_j < 2^{k-l}$ 的情况，其中 j 是该块的一个索引列，且 l
一个大于或等于 1 的整数。当这种情况发生时，H 可以设为 2^{k-l}，从而缩小内存访问范围。HASH 的整个执行过程中都会持续进行这一过程。

复杂度分析： HASH 的复杂度分析与 SPARS 类似，因为两者以相同的方式生成中间乘积。不过，这两种算法之间存在一些差异。第一个差异是，HASH 需要计算哈希函数值，而 SPARS 直接访问 SPA数组，因此会产生少量开销。第二个差异来自 HASH 运行时发生的哈希函数冲突。每一步并行计算的 VL 个中间乘积中只要发生一次冲突，就足以使所有 VL 项计算都出现一些空闲时间。如果单个元素发生冲突的概率为 p_c，那么 HASH 中发生冲突的概率为 $1 - (1 - p_c)^{\mathrm{VL}}$；当$p_c << 1$ 时，该概率接近 VL $\times p_c$。最后，第三个也是最主要的差异在于 HASH 表与 SPARS 稠密向量的大小。SPARS 稠密向量的内存占用量级为 $O(m^A \cdot \mathrm{VL})$，而对于超稀疏矩阵，HASH 可将其降至$O(\max_j Op_j \cdot \mathrm{VL})$。

3.3 混合算法

本节提出 H-SPA(t) 和 H-HASH(t) 混合算法，分别结合了 SPA 与 SPARS、HASH 的优点。正如我们在第 3.1 节所述，且第 5.2 节的结果通过实验证实，对于每列非零元数极少的稀疏矩阵，SPARS 的性能优于 SPA；而每列非零元数相对较多时，SPA 的性能则优于 SPARS。与 SPA 相比，HASH 也有同样的总体结论。H-SPA(t) 和 H-HASH(t) 需要与 SPARS 和 HASH 相同的排序预处理。通过参数t，H-SPA(t)（或 H-HASH(t）可在 SPA 与 SPARS（或 HASH）之间动态切换。由于列按负载降序排列，

H-SPA(t) 和 H-HASH(t) 只要满足 $Op_j >= t$，就会使用 SPA 处理前面的列。随后，当$Op_j < t$时，H-SPA(t)（或 H-HASH(t）会切换到 SPARS（或 HASH），以分块方式处理后面的列。块大小由第 3.1 节所述的分块算法确定。 t 的两个临界值分别为 $t = 0$ 和 $t = \infty$。当 $t = 0$时，H-SPA(0) 和 H-HASH(0) 使用 SPA 处理所有列，因此 H-SPA(0)、H-HASH(0) 和 SPA 表示相同的算法，但 H-SPA(0) 和 H-HASH(0) 的列会先进行排序。当 $t = \infty$时，所有列都按块处理，因此 H-SPA(∞) 与 SPARS 表示相同的算法，而 H-HASH(∞) 则与 HASH 相同。

4 ESC

本节将简要介绍文献中另一种常用的算法，称为 ESC [17, 46]。该算法不依赖 Gustavson 方法。ESC 代表扩展-排序-压缩，分为三个阶段。第一阶段扩展以一组 C 列为输入，并生成计算这些 C 列所需的、来自 A 和 B 的所有中间乘积，并将其存储在名为 esc_val 的数组中。除乘积的值之外，每个乘积的行索引和列索引也分别保存到两个数组 id_row 和 id_col 中。第二阶段排序首先使用 id_row 作为键对 esc_val 进行排序，然后使用 id_col 作为键进行排序。在该阶段结束时，C 中位置相同的值彼此相邻，并按列排序。最后一个阶段压缩会累加具有相同索引对的所有值，从而生成 C的非零元素。随后可将它们存储为 CSC 格式。

实现。我们按以下方式实现 ESC。在 展开阶段，我们从 B 的列生成 k 三元组（id_row,id_col,esc_val），与 SPA 的处理方式相同。对于 B 的每个非零元素，使用一条向量指令；向量长度对应于 A相应列中的非零元数量。排序阶段采用基数排序 [49], 实现，该排序由多轮向量化桶排序组成。最后，压缩阶段使用跨步内存指令完成。虚拟处理器 i（$i = 0$、……、VL -1）将三元组 ^{k}VLi压缩为 ^{k}VL（$i+1$）-1。两个不同虚拟处理器处理的三元组可能需要合并，这一步在末尾通过一个大小为 VL 的顺序循环完成。生成的非零元素按顺序存储到矩阵C 中，以避免依赖速度较慢的前缀和操作；因为每个虚拟处理器和列最终产生的元素数量可能不同，而且使用向量化算法不易设置 CSC 格式的列指针。

优化。 ESC 的执行很大程度上取决于排序阶段的执行时间。该阶段通常占总执行时间的 80% 至 95%。算法。在长向量架构上，决定基数排序性能的两个主要参数是待排序元素的数量和基数长度。

增加元素数量可以更好地利用较大的向量长度，因为该算法只包含一个大小为 VL 的顺序循环。相反，减少元素数量可以加快索引加载与存储的执行速度，尤其是在一轮结束时对元素重新排序的情况下。因此，存在一个能够实现最佳性能的中间元素数量。为此，我们通过不断对列进行分组，直到中间乘积的数量超过给定阈值，来确定 ESC 在每次迭代中处理的列数；在我们的实验中，该阈值设为 10,000。

基数长度（记为 r）也会影响基数排序的性能。较大的基数长度可以减少元素排序所需的轮数。如果索引的最大值为 N，则所需轮数为 $\lceil \log_{2^r} N \rceil$。在稀疏通用矩阵乘法中，最大值要么是列数，要么是 C的行数。然而，在向量化算法中，每个虚拟处理器持有 2^r 个桶，这意味着用于桶排序的数组总大小为 VL $\cdot$ 2^r。同样，为了减小索引加载与存储期间内存地址的范围，较小的 r 更为理想；当 VL 较大时，这一点尤为重要。我们通过实验确定，5 或 6 是在使用较大向量长度时最合适的基数长度，因此将其设为 $r = 5$，除非 $r = 6$ 能够减少排序轮数。

5 实验评估

本节对我们的算法进行评估。我们在第 5.1 节介绍实验设置。在第 5.2 节中，我们使用合成矩阵评估 SPA、SPARS 和 HASH 的性能。在第 5.3 节中，我们使用 SuiteSparse MatrixCollection [19] 对 SPA、SPARS、HASH、H-SPA(t)、H-HASH(t) 和 ESC 进行全面评估。我们在第 5.4 节中分析 H-SPA(t) 和 H-HASH(t) 对参数t, b_{min}, 以及 b_{max} 的敏感性。源代码和评估文件可在 [1] 找到。

5.1 实验设置

硬件平台：我们在由一个内核和一个向量处理单元（VPU） [36]组成的向量处理器架构上开展实验。该内核是一个六级单发射RISC-V内核，支持在VPU上执行向量指令。向量指令由内核解码后发送至VPU执行。标量指令则在内核上运行。VPU拥有8个向量通道，支持的最大向量长度（VL）为256个双精度（DP）元素。每个通道配备一个融合乘加（FMA）单元，每周期可执行2次双精度浮点（FP）运算，因此整个VPU每周期最多可执行16次双精度浮点运算。我们使用FPGA实现该向量处理器架构。所用FPGA为一块VCU128开发板

运行频率为50 MHz，用于实现标量核心和VPU加速器。该开发板配备4 GB DDR4 动态随机存取存储器和1 MB L2 缓存。

稀疏矩阵： 我们的实验研究涉及两组矩阵。第 5.2 节介绍合成矩阵实验的结果：我们指定每列的非零元素数量，并使用均匀分布随机变量将它们分布到各列中。合成矩阵实验使我们能够轻松分析算法对输入矩阵稀疏性的敏感程度。第 5.3 节针对从 SuiteSparse 矩阵集合 [19] 中获取的 40 个稀疏矩阵组成的异构集合，对 SPA 、 SPARS 、 HASH 、 H- SPA(t) 、 H-HASH(t) 和 ESC 进行了全面评估。第 5.2 节和第 5.3 节均进一步介绍了所用矩阵和算法的详细信息。

执行与测量详情 对于 $A = B$ 的情况，我们计算 $C = A \times B$ 。性能结果报告算法的总执行时间。我们的测量不计入内存分配等执行前步骤，因为这些步骤与我们所考察的并行算法性能正交。

5.2 合成矩阵的性能

本节使用合成矩阵评估 SPA、SPARS 和 HASH 的性能。这些矩阵能够清晰展示这些算法对某些矩阵输入参数的敏感性。我们使用两个参数生成合成矩阵： n（行列数）和 Z（每列的非零元素数）。均匀分布随机变量决定每列中 Z 元素的具体位置。由于每列中的非零元素数保持不变，因此每列需要执行的乘法运算数也保持不变，且等于 Z^2。因此，在这些实验中无需对列进行排序。此外，参数 b_{min} 也无关紧要，因为每个块的大小始终达到最大值 b_{max}。

图3比较 SPA 和 SPARS 的性能，图4比较 SPA 和 HASH 的性能。x 轴表示 b_{max} 参数的值。我们取 b_{max} 从 8 到 256。矩阵规模 n 设置为 2560，因此在所有场景中至少有 10 个块。y 轴表示执行时间。实线表示 SPARS 或 HASH 的执行时间，而虚线表示 SPA。我们考虑以下取值：

每列非零元素的数量， Z：2，4，5，6，8 和 10。我们使用不同的颜色表示这些不同的值，以展示 SPARS（或 HASH）和 SPA 的性能。图表右侧的图例显示了我们用于表示考虑不同 Z 值时结果的颜色。

图3中的结果表明，SPARS 在 $b_{max} = 40$ 时达到最佳性能，而当 b_{max} 值更大时，性能会下降。

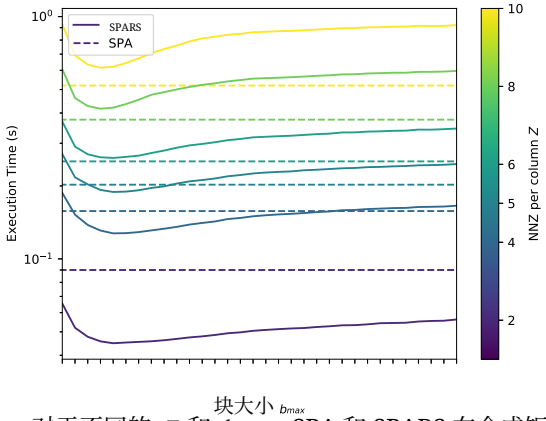


图3。 对于不同的 Z 和 b_{max}，SPA 和 SPARS 在合成矩阵上的性能。

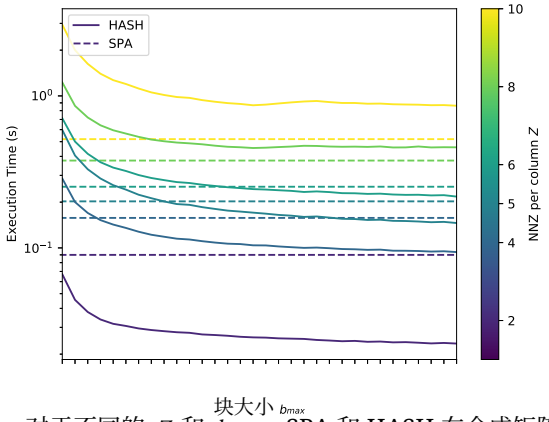


图4。对于不同的 Z 和 b_{max}，SPA 和 HASH 在合成矩阵上的性能。

这种现象由索引加载和存储指令的性能决定。较大的 b_{max} 值会生成向量长度（VL）较大的向量指令，因此，设置较大的 b_{max} 值会迫使索引加载与存储向量指令在较短时间内对大范围地址发起大量内存请求，从而使内存子系统不堪重负，并导致 b_{max} 值较大时性能下降。相反，图4中的结果显示，HASH 不受这一性能问题影响，并且在 b_{max} 最大时比 SPARS 更快。这证实，缩小包含 C 值部分累加值的数据结构的大小至关重要，因为这样可以缩小索引加载和存储所访问的地址范围。值得注意的是，较大的 b_{max} 值使 HASH 能够以较大的 VL 运行，从而显著提升性能。尽管哈希表冲突的数量随 VL 线性增加，如第3.2节所示，HASH 在 VL 较大时性能仍会提升。SPA 的结果始终呈水平线，因为参数 b_{max} 不会影响 SPA 的性能。

图 3 和图 4 中的结果还表明，对于较大的 Z，SPA 的性能优于 SPARS 和 HASH；而对于较小的 Z，SPARS 和 HASH 的速度快于 SPA。

第 3.1 节所述的复杂度分析也显示了相同的趋势。具体而言，当我们将 $b_{max} = 40$ 设置为使 SPARS 的性能达到最大值时，对于每列非零系数数量恒定的稀疏矩阵，当且仅当 $Z \leq 5$ 时，SPARS 的速度才快于 SPA。类似地，当我们将 $b_{max} = 256$ 设置为使 HASH 的性能达到最大值时，当且仅当 $Z \leq 6$ 时，HASH 的速度才快于 SPA。对于一般情况，这一特定条件并不严格成立，因为非零元素在矩阵各列中的分布会显著影响这三种算法的性能。最后可以看到，当 Z 非常小时，HASH 的效率确实很高，因为它能够利用非常小的哈希表，实现快速的索引加载与存储。当 Z 增大时，SPARS 的速度会快于 HASH，但仍不如 SPA；这是因为与稠密向量中的直接索引相比，索引计算和冲突增加了该算法的开销。

5.3 真实矩阵上的评估

本节评估 SPA、SPARS、HASH、H-SPA(t)、H-HASH(t) 和 ESC 算法在一组 40 个方阵上的性能，这些矩阵属于 SuiteSparse 矩阵集合 [19]。这些矩阵的选择标准为：(i) 能够放入我们实验平台的可用内存；(ii) 代表不同的稀疏模式。这些矩阵的每列非零元素的平均比率各不相同，这是决定 SPA、SPARS 和 HASH 性能的一个重要因素。各矩阵中元素在列间分布的方差也不尽相同，这也凸显了混合算法相较于 SPARS 和 HASH 的性能优势。表1列出了这 40 个矩阵。第二列和第三列分别显示每个矩阵的行列数（列 *Size*）以及每个矩阵的非零元素总数（列 $\#NNZ$）。第四、第五和第六列分别显示每列非零系数的最小值、最大值和平均值，第七列则显示每个矩阵每列非零系数分布的方差。第 8 至第 11 列按每列乘法次数列出相同的统计数据。

我们考虑以下十种算法：

- Spa，即 H-SPA(0)。
- Spars-16/64，*i.e.*H-SPA(∞)，块大小在 $b_{min} = 16$ 和 $b_{max} = 64$ 之间可变。
- Spars-40/40，*i.e.*H-SPA(∞)，块大小为 40，*i.e.*，$b_{min} = b_{max} = 40$。
- H-Spa-16/64，*i.e.* H-SPA(40)，$b_{min} = 16$，$b_{max} = 64$。
- H-Spa-40/40, *i.e.* H-SPA（40）, $b_{min} = b_{max} = 40$.
- Hash-32/256，*i.e.* H-HASH（∞），$b_{min} = 32$，$b_{max} = 256$。
- Hash-256/256，*i.e.* H-HASH（∞），$b_{min} = b_{max} = 256$。
- H-哈希-32/256，*i.e.* H-HASH（40），$b_{min} = 32$，$b_{max} = 256$。
- H-Hash-256/256, *i.e.* H-HASH(40), $b_{min} = b_{max} = 256$。
- ESC，如第4节所述。

表1在第 12 列以秒为单位报告了 SPA 的执行时间，并在最右侧的 9 列中列出了其他算法相对于 SPA 的加速比。高亮单元格表示每个矩阵上表现最佳的算法。表1中的矩阵按每列平均乘法次数递增排序。实验结果与第 5.2 节中的合成矩阵实验以及第 3.1 节和第 3.2 节的复杂度分析一致：基于 SPARS 和 HASH 的算法（Spars-40/40、Spars-16/64、Hash-256/256、Hash-32/256）在每列非零元数较少的矩阵上表现良好。相反，在处理较稠密的矩阵时，SPA 比 Spars-40/40、Spars-16/64、Hash-256/256 和 Hash-32/256 更快。对于全部矩阵，Spars-40/40 相较于 SPA 的平均加速比为 1.13× 。在集合中最稀疏的 22 个矩阵上，其加速比达到 1.38× ；这些矩阵的每列非零元数平均少于 6，每列乘法次数少于 39。对于 40 个矩阵，Spars-16/64 的平均加速比为 1.08× ，在其中最稀疏的 22 个矩阵上则达到 1.34× 。对于 40 个矩阵，Hash-256/256 和 Hash-32/256 的平均加速比分别为 1.41× 和 1.44×。对于其中最稀疏的 22 个矩阵，两者的平均加速比分别为 1.85× 和 1.88× 。对于超稀疏矩阵，基于 HASH 的算法通常比基于 SPARS 的算法更快。而在 SPA 是最快算法的矩阵上，情况则恰好相反，这印证了第 5.2 节中的观察。

Spars-40/40 通常比 Spars-16/64 提供更好的性能。Spars-40/40 将块大小固定为 40，以获得最佳性能；与 Spars-16/64 那样在 $b_{min} = 16$和 $b_{max} = 64$之间形成负载相近的块相比，这是一种更好的策略。Spars-40/40 比 Spars-16/64 更快的原因有两个：(i) 由于许多列共享相同的值 Op_j，Spars-16/64 通常会形成大小为 64 的块（b_{max}值），而这并非最佳块大小，正如第 5.2 节所示；(ii) 尽管 Spars-40/40 会使向量处理器的部分单元在一段时间内处于空闲状态，但它通过使用更少的指令处理比 Spars-16/64 更多的列来弥补这段空闲时间。平均而言，Hash-32/256 的加速比略高于 Hash-256/256，但在这两种算法中，Hash-256/256 更经常是最佳算法。将块大小固定为 256，可以通过一条指令计算尽可能多的中间乘积；当每列的非零元素数量变化不大时，这是性能最佳的取值，正如我们在第 5.2 节中所展示的。

与 Hash-256/256 相比，稀疏通用矩阵乘法中最先进的向量化 HASH 算法在使用 256 的块大小时，平均耗时多 52%。对于方差非常小的矩阵 olm1000、rdb1250 和 Chebyshev3，Hash-256/256 相对于这一先前方法所获得的最低加速比为 1.01× 。最高

Name	Size	#NNZ	#每列非零元数量				#每列乘法次数				稀疏度（秒）	Spars	Spars	H-稀疏	H-稀疏	Hash	Hash	H-哈希	H-哈希	ESC
			Min	最大值	平均值	方差	最小值	最大值	平均值▲	方差		16/64	40/40	16/64	40/40	32/256	256/256	32/256	256/256	
poli	4008	8188	1	15	2.04	0.46	1	38	3.92	5.83	1.50e-1	2.10×	2.22×	2.10×	2.21×	4.21×	3.83×	4.20×	3.83×	0.95×
S40PI_n1	2028	5007	0	8	2.47	0.30	0	25	6.39	1.50	8.69e-2	2.05×	2.05×	2.05×	2.04×	3.63×	3.30×	3.61×	3.28×	0.70×
科霍宁	4470	12731	0	51	2.85	10.20	0	221	11.88	238.58	2.32e-1	1.17×	1.21×	1.19×	1.26×	1.22×	1.27×	1.37×	1.69×	0.54×
Hamrle2	5952	22162	2	8	3.72	3.42	4	40	14.07	82.28	3.78e-1	1.29×	1.42×	1.29×	1.42×	2.26×	2.31×	2.25×	2.32×	0.59×
bp_0	822	3276	1	20	3.99	10.43	1	107	14.18	272.39	4.97e-2	1.33×	1.46×	1.41×	1.49×	1.26×	1.05×	1.43×	1.43×	0.54×
barth4	6019	23492	2	10	3.90	0.68	4	51	14.91	22.04	3.79e-1	1.36×	1.48×	1.36×	1.48×	2.27×	2.29×	2.28×	2.33×	0.57×
振荡_直流_30	430	1544	1	13	3.59	2.33	1	60	15.00	65.90	2.43e-2	1.33×	1.45×	1.35×	1.51×	1.23×	1.13×	1.32×	1.42×	0.50×
rw5151	5151	20199	1	4	3.92	0.11	2	16	15.49	3.1480	3.09e-1	1.32×	1.40×	1.32×	1.40×	2.20×	2.21×	2.19×	2.21×	0.53×
olm1000	1000	3996	3	4	4.00	0.00	10	16	15.97	0.15	5.39e-2	1.55×	1.48×	1.55×	1.48×	2.15×	2.18×	2.12×	2.16×	0.51×
tub1000	1000	3996	3	4	4.00	0.00	10	16	15.97	0.15	5.80e-2	1.68×	1.60×	1.68×	1.60×	2.29×	2.33×	2.28×	2.32×	0.56×
bcspwr09	1723	6511	2	15	3.78	3.02	5	80	17.30	102.80	1.10e-1	1.30×	1.38×	1.30×	1.37×	1.39×	1.57×	1.42×	1.77×	0.48×
saylr3	1000	3750	1	7	3.75	4.06	1	42	18.13	166.59	6.00e-2	1.25×	1.38×	1.26×	1.36×	1.66×	2.03×	1.63×	1.92×	0.53×
sherman4	1104	3786	1	7	3.43	6.40	1	47	18.16	332.27	5.77e-2	1.17×	1.23×	1.17×	1.20×	1.33×	1.53×	1.30×	1.42×	0.35×
qh1484	1484	6110	2	13	4.12	2.56	5	68	19.51	94.54	9.71e-2	1.28×	1.34×	1.28×	1.33×	1.38×	1.49×	1.40×	1.67×	0.43×
shyy41	4720	20042	1	6	4.25	1.63	2	36	19.62	129.92	3.12e-1	1.26×	1.38×	1.26×	1.38×	2.16×	2.23×	2.16×	2.23×	0.48×
rajat03	7602	32653	1	52	4.29	1.26	3	303	19.71	51.70	5.15e-1	1.19×	1.27×	1.22×	1.33×	1.98×	1.40×	2.16×	2.18×	0.48×
young3c	841	4089	3	5	4.74	0.21	11	25	22.51	11.03	5.85e-2	1.40×	1.38×	1.40×	1.38×	1.99×	2.12×	2.00×	2.12×	0.49×
sherman3	5005	20033	1	7	4.00	7.09	1	49	23.11	411.19	3.36e-1	1.00×	1.10×	1.09×	1.12×	1.64×	1.83×	1.34×	1.40×	0.42×
dw1024	2048	10114	3	8	4.94	0.26	11	49	24.54	17.05	1.52e-1	1.26×	1.23×	1.25×	1.22×	1.79×	1.84×	1.82×	1.84×	0.41×
rdb1250	1250	7300	4	6	5.84	0.15	18	36	34.25	14.17	1.07e-1	1.21×	1.17×	1.21×	1.17×	1.64×	1.63×	1.63×	1.63×	0.33×
tols1090	663	1712	1	22	3.25	25.97	1	471	38.00	13361.58	7.30e-2	0.92×	0.79×	1.36×	1.36×	0.70×	0.35×	1.52×	1.52×	0.25×
fpga_dcop_05	1220	5852	1	36	4.80	20.44	7	164	38.12	427.76	1.09e-1	0.95×	1.00×	1.03×	1.06×	0.90×	0.84×	1.05×	1.15×	0.32×
watt_1	1856	11360	2	7	6.12	1.67	6	49	39.37	125.89	1.72e-1	1.08×	1.08×	1.05×	1.05×	1.36×	1.39×	1.14×	1.13×	0.35×
saylr4	3564	22316	3	7	6.26	0.56	13	49	39.76	52.96	3.55e-1	0.93×	1.02×	0.98×	1.02×	1.48×	1.61×	1.16×	1.20×	0.37×
orsreg_1	2205	14133	4	7	6.41	0.41	19	49	41.49	49.78	2.06e-1	1.04×	1.04×	1.00×	1.00×	1.55×	1.59×	1.19×	1.20×	0.33×
wang1	2903	19093	4	7	6.58	0.37	19	49	43.62	46.98	2.93e-1	1.01×	1.07×	1.01×	1.03×	1.52×	1.56×	1.11×	1.12×	0.35×
gemat12	4929	33044	1	28	6.70	11.56	1	206	45.27	735.35	6.12e-1	0.85×	0.93×	0.99×	1.02×	0.79×	0.95×	1.06×	1.10×	0.37×
lshp3466	3466	23896	4	7	6.89	0.20	21	49	47.74	20.56	3.44e-1	0.94×	1.01×	0.98×	0.98×	1.46×	1.48×	1.00×	1.00×	0.31×
LeGresley_4908	4908	30482	2	34	6.21	9.39	8	324	48.25	1065.07	5.03e-1	0.79×	0.86×	0.99×	1.02×	1.04×	1.00×	1.17×	1.20×	0.32×
lns_3937	3937	25407	1	13	6.45	10.39	1	113	48.44	866.46	4.00e-1	0.82×	0.89×	0.99×	1.01×	1.22×	1.23×	1.06×	1.07×	0.32×
孔隙 2	1224	9613	2	30	7.85	29.53	10	298	63.62	2199.05	1.50e-1	0.78×	0.89×	1.01×	1.01×	0.77×	0.59×	1.03×	1.01×	0.29×
Chebyshev3	6435	51480	3	9	8.99	0.02	15	65	64.92	2.12	5.23e-1	0.94×	1.01×	1.01×	1.01×	1.36×	1.36×	1.00×	1.00×	0.31×
str_200	363	3068	1	26	8.45	84.35	1	449	70.61	12314.86	4.93e-2	0.83×	0.91×	1.02×	1.05×	0.65×	0.25×	0.99×	0.93×	0.32×
dwt_2680	2680	25026	4	19	9.34	3.44	27	228	90.65	623.75	4.01e-1	0.70×	0.76×	1.00×	1.01×	0.77×	0.91×	1.00×	0.99×	0.26×
cage9	3534	41594	3	23	11.77	14.08	15	474	152.60	7046.5951	8.00e-1	0.65×	0.73×	1.00×	1.00×	0.57×	0.59×	1.00×	1.00×	0.25×
nasa1824	1824	39208	6	42	21.50	49.58	65	1197	511.64	59420.46	8.14e-1	0.41×	0.47×	1.00×	0.99×	0.36×	0.31×	0.99×	0.99×	0.16×
ex22	839	22460	7	62	26.77	190.67	176	2270	907.22	220428.89	5.50e-1	0.33×	0.41×	1.00×	1.01×	0.29×	0.20×	1.02×	1.00×	0.17×
加法器_直流工作点_01	1813	11156	1	1332	6.15	1076.11	2	9439	1014.45	396265.13	2.25	0.61×	0.64×	1.00×	1.00×	0.34×	0.18×	1.00×	1.00×	0.20×
Goodwin_013	1965	56059	5	62	28.53	224.66	138	2359	1048.69	316412.44	1.47	0.31×	0.38×	1.00×	1.00×	0.27×	0.24×	1.00×	0.99×	0.14×
iprob	3001	9000	2	3000	3.00	2994.00	3002	6000	3003.00	2994.00	10.33	0.77×	0.72×	1.00×	1.00×	0.34×	0.31×	1.00×	0.99×	0.18×
									平均加速比		1.00×	1.079×	1.131×	1.204×	1.235×	1.436×	1.413×	1.535×	1.569×	0.399×
Name	Size	#NNZ	Min	Max	Avg.	Var.	Min	Max	平均值▲	Var.	稀疏度（秒）	SPARS	SPARS	H-稀疏	H-稀疏	Hash	Hash	H-哈希	H-哈希	ESC
			每列非零元素数				每列乘法次数					16/64	40/40	16/64	40/40	32/256	256/256	32/256	256/256	

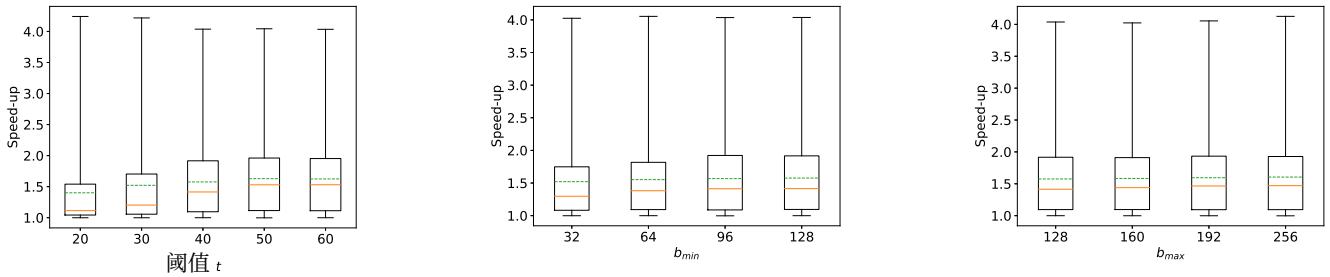
表1。40 个稀疏矩阵的统计数据和性能结果。

加速比为 4.49×，对应 rajat03，其中各列包含 3 到 303 个中间乘积。

此外，预处理时间通常不到 HASH 算法执行时间的 40%，对于最后 6 个矩阵，甚至不到 5%。只有 5 个矩阵的相对预处理时间超过 40%：poli、S40PI_n1、Hamrle2、barth4 和 rw5151。对于这 5 个矩阵，Hash-256/256 和 Hash-32/256 的加速比都不错。

与整个由40个矩阵组成的集合中的 SPA 相比，H-Spa-40/40 达到了 1.24× 平均加速比。对于其中最稀疏的22个矩阵，H-Spa-40/40 达到了 1.42× 平均加速比。表1显示，H-Spa-40/40 能够大幅超越 SPA、Spars-40/40 和 Spars-16/64。对于每列计算负载较大的矩阵块，H-Spa-40/40 能够动态调整其策略并采用 SPA，因此能够超越 Hash-256/256 和 Hash-32/256。对于计算负载较低的矩阵块，它能够切换至 SPARS，但其性能仍低于基于 HASH的算法。对于列间非零值数量差异较大的矩阵，H-Spa-40/40 相较于 SPA、Spars-40/40 和 Spars-16/64 可获得特别大的加速比。在这些矩阵上，SPA 和基于 SPARS 的方法都无法实现良好的性能。例如，对于 tols1090，与 SPA 相比，H-Spa-40/40 达到了 1.36× 的较大加速比。在该矩阵上，它还大幅超越了 Spars-40/40 和 Spars-16/64，以及 Hash-256/256 和 Hash-32/256。表1表明，H-Spa-16/64 能够实现与 H-Spa-40/40 相近的性能。在整个由40个矩阵组成的集合中，H-Spa-16/64 相对于 SPA 达到了平均 1.20× 加速比；与 H-Spa-40/40 一样，对于其中最稀疏的22个矩阵，它相对于 SPA 实现了最大的加速比。H-Spa-16/64 的性能略差于 H-Spa-40/40，原因与 Spars-16/64 差于 Spars-40/40 相同： *i.e.*，将 b_{min} 和 b_{max} 设置为40会生成比将 b_{min} 设置为16、将 b_{max} 设置为64更好的矩阵块。

对于 H-Hash-256/256 和 H-Hash-32/256，同样可以得出上述结论，因为相较于 SPA，它们分别实现了 1.57× 和 1.54× 的加速比。对于密度最高的矩阵，它们能够切换到 SPA，从而避免 Hash-256/256 和 Hash-32/256 在这些矩阵上出现的性能下降。由于基于 HASH 的算法在超稀疏矩阵上的性能优于基于 SPARS 的算法，因此它们的平均加速比高于 H-Spa-40/40 和 H-Spa-16/64。特别是，H-Hash-256/256 在最稀疏的 22 个矩阵上实现了 1.99× 的加速比。



(a) 变化 t (b) 变化 b_{min} (c) 变化 b_{max}

图 5. H-HASH(t) 相对于 SPA 的加速比，比较参数 t (左)、 b_{min} (中) 和 b_{max} (右) 的不同取值。

H-Hash-256/256 的性能略优于 H-Hash-32/256，因为对于最稀疏的矩阵，Hash-256/256 通常比 Hash-32/256 更快。

ESC 是一种适用于向量处理器的慢速算法。尽管该算法不存在负载不均衡问题，但与累加值相比，对部分输出进行排序的开销很大。在 40 个矩阵上，ESC 相较于 SPA 的平均加速比仅为 0.40× 。

5.4 H-HASH(t) 的敏感性分析

本节针对参数 b_{min}、 b_{max} 和 t，对 H- 哈希(t)算法的性能进行敏感性分析。第 5.3 节考察了两组参数（b_{min}，b_{max}）、（32，256）和（256，256），以及 H-HASH(t) 的单个阈值$t = 40$；而图 5 展示了在不同参数设置下，针对 SPA，H-HASH(t) 在表1所列的 40 个矩阵上取得的加速比分布。橙色实线表示中位数，绿色虚线表示平均值，箱体边缘表示第一和第三四分位数，须线则表示所取得加速比的最小值和最大值。H-稀疏(t)也有类似结果，但由于其对参数 b_{min}、 b_{max} 和 t 的敏感性模式与 H-HASH(t) 非常相似，因此这里不再赘述。

图 5(a) 展示了当 $b_{max} = b_{min} = 128$ 和 $t = 20$、30、40、50、60 时，H-HASH(t) 的加速比分布。提高阈值 t 会使最大加速比略有下降。然而，较小的阈值并不利于大多数矩阵：例如，采用 $t = 20$时，有 24 个矩阵的加速比低于 1.20×；而采用$t = 50$、60 时，这样的矩阵只有 12 个。$t = 20$的平均加速比为 1.40× ， $t = 30$为 1.52×， $t = 40$为 1.57× ， $t = 50$ 为 1.63× ，$t = 60$为 1.62× 。中位数也呈现相同趋势，区间$[40, 60]$ 的整体性能最佳。

图 5(b) 展示了当 $b_{max} = 128$、 $b_{min} = 32$,64,96, 128 和 $t = 40$ 时，H-HASH(t) 的加速比分布。所有 b_{min} 取值对应的分布都非常相似。 $b_{min} = 32$、$b_{min} = 64$、 $b_{min} = 96$ 和 $b_{min} = 128$ 的平均加速比分别为 1.52× 、 1.55× 、 1.57× 和 1.58× 。总的来说，性能整体略有提升，随着 b_{min} 增大，这种提升会更加明显，因为更大的向量长度有利于 HASH 算法。

图 5(c) 展示了当 $b_{max} = 128$、160、192、256、$b_{min} = 128$ 和$t = 40$时，H-HASH(t) 的加速比分布。$b_{max} = 128$的平均加速比为 1.58× ， $b_{max} = 160$为 1.58× ， $b_{max} = 192$ 为 1.59× ，$b_{max} = 256$为 1.61× 。与 b_{min} 的研究类似，提高 b_{max}可以同时处理更大的列块。其影响较小，因为只有在每列乘法次数完全相同时，才会使用 b_{min}到 b_{max} 之间的块大小；因此，提高最小块大小 b_{min} 比提高 b_{max} 影响更大。

6 相关工作

负载不均衡 负载不均衡是 Gustavson 方法中经常被提及的问题，在 GPU 上尤其如此。Parger 等人 [40] 对矩阵进行低复杂度的预处理分析，其复杂度与非零元数量呈线性关系。根据分析结果，可以采用分箱方法来减少 GPU 上算法的负载不均衡。Liu 等人 [34] 计算每行的乘法运算次数，并将其作为该行最终非零元数量的上界。他们根据此上界将各行划分为 38 个箱，以此指导算法和分配空间方面的执行选择。Nagasaka 等人 [37] 将矩阵行映射到不同线程，同时确保每个线程执行的浮点运算总数相同。我们提出的 SPARS、HASH 和混合算法通过应用第 3.1 节所述的排序技术，缓解由矩阵列之间的差异导致的负载不均衡。

稠密向量的替代方案Deveci 等人 [20]将哈希映射和稠密向量都用作稀疏累加器。Nagasaka 等人 [37] 提出了两种基于哈希映射的替代方案。第一种是经典算法 [6], ，其中列索引用作键，发生冲突时采用线性探测。第二种方案使用向量寄存器来增加哈希映射的每个单元中存储的元素数量。这样可以减少冲突，但代价是更高的开销。H-哈希（t）包含三个

与这些先前方法相比，具有根本性创新的方面：i）通过对列进行排序 B来减少负载不均衡；ii）动态调整哈希表大小；iii）根据基于稠密向量累加器（*i.e.* SPA）和基于哈希映射（*i.e.* HASH）的技术，在不同算法之间动态切换。我们的实验表明，如果没有这些创新，在我们的长向量平台上，基于HASH的算法性能会逊于SPA（相较SPA性能下降0.95×）。值得注意的是，我们的算法（*e.g.* H-哈希（t））可以像先前研究 [37], 那样应用于配备SIMD单元 [43]的多核CPU设备，并带来性能提升。

另一种累积 A 列中非零元的方法是扩展-排序-压缩（*ESC*） [11]。该方法扩展部分乘积列表及其对应的行索引和列索引，对它们进行排序，最后压缩所有行索引和列索引相同的条目。Cusp [16]采用了这种方法。Winter等人 [46] 使用ESC算法。他们采用两级负载均衡方案，将负载分配给GPU线程。Dalton等人 [17] 在其细粒度ESC实现中，采用与我们相同的排序方法来减少负载不均衡。Kunchum等人 [29] 提出了ESC的一种变体，旨在优化非零元数量远高于对应行的 A 行的性能。第 5.3 节评估了ESC算法，并表明该算法在长向量架构下性能较差。

非基于Gustavson的方法Lee等人 [30] 采用外积公式，并设计了一种块重组算法，以改善GPU线程之间的负载均衡。Niu等人 [39] 最近提出了一种用于稀疏通用矩阵乘法的分块算法，以避免Gustavson 方法在GPU上的常见问题。他们还提出了一个阈值，用于在每个分块的稠密和稀疏表示之间切换，并设计了一种用于稀疏矩阵-向量乘法的分块算法 [38]。最后，Xie等人提出了一种用于稀疏通用矩阵乘法的新型基于人工智能的方法 [47]。尽管其所依赖的算法都采用Gustavson 方法，但他们研究了不同的矩阵存储格式和算法，并训练了一个定制的深度学习网络，为给定的输入矩阵和目标架构选择最佳方案。本文采用基于 Gustavson 方法的方法计算稀疏通用矩阵乘法，因为它能为向量架构提供高度并行性。

SpMV 和 SpMM 的并行执行多年来，人们一直在探索 SpMV（稀疏矩阵-向量乘法）等其他稀疏线性代数内核的并行执行。SpMV 运算$y = A \times x$与稀疏通用矩阵乘法的主要区别在于，x向量是稠密的。负载不均衡会降低 SpMV 的性能 [35]。Li 等人 [32]针对不同的矩阵存储格式，在 GPU 上对 SpMV 进行了详细的性能分析。近期，有研究提出了在长向量架构上高效运行 SpMV 的方法 [22]。

SpMM 执行稀疏矩阵与稠密矩阵之间的乘法。虽然可以通过多次运行 SpMV 来计算 SpMM [45]，也就是将稠密矩阵视为多个稠密向量，但也有人提出了基于整个矩阵的数据访问合并的优化方法 [24]。Koanantakool 等人 [27] 给出了在分布式内存场景下高效运行 SpMDM 的下界和最优算法。Azad 等人 [8] 针对稀疏通用矩阵乘法提出了类似的分析，并将其扩展到包含内存约束 [25]。

向量处理器上的数值算法 此前已有研究提出了在向量架构上高效运行相关数值算法的方法。例如，Lim 等人 [33]研究了如何利用向量指令高效执行稠密通用矩阵-矩阵乘法。他们采用分块版本的算法，结合 AVX-512 指令，并精心选择块大小，以高性能地执行微内核。Kim 等人 [26], 进一步扩展了这项工作，提出一种自动调优机制，通过考虑块大小或预取距离等多个参数来探索缩减后的搜索空间。快速排序和双调排序也已针对 AVX-512 指令集架构 [12], 进行了优化，而 Armejach 等人 [7]则针对 ARM SVE 优化了模板计算。Zhong 等人 [50] 研究了在 MPI 归约中使用向量内建函数的情况，涵盖 AVX-512 和 SVE 指令集架构。

Afanasyev 等人提出了面向 SX-Aurora TSUBASA 向量架构的广度优先搜索 [3]和连通分量 [2]图算法 [28], ，该架构的寄存器可存储 256 个双精度值。Li 等人 [31] 提出了一种基于 SPA、面向同一长向量架构的稀疏通用矩阵乘法算法。本文提出的算法创新带来了比 SPA 更高的性能，尤其适用于超稀疏矩阵。

7 结论

基于SPA的方法构成了在向量架构上运行稀疏通用矩阵乘法的最先进方法，并且对于每列至少具有七个非零元素的矩阵，能够提供良好的性- 能。由于这些方法无法充分发挥长向量架构在处理仅含少量非零元- 素的超稀疏矩阵时的潜力，我们提出了两种能够在向量架构上高效操- 作超稀疏矩阵的算法，即SPARS和HASH。为了成功结合SPA与SPARS（或HASH），并在广泛的稀疏性场景下提供良好性能，我们提出了H-稀疏(t)（或H-哈希(t)）。它采用混合执行模- 型，可在SPA与SPARS（或HASH）之间动态切换。特别是，H-哈希(t)通常是最佳算法，相较于SPA，在由40个稀疏矩阵组成的异构集合上获得了1.57× 的平均加速比。对于其中最稀疏的22个矩阵，H-哈希(t)实现了1.99×

相较于SPA的平均加速比。未来，我们计划提出通过利用向量寄存器文件中的数据重用，进一步加速长向量架构上稀疏通用矩阵乘法内核的方法。

致谢

本研究获得了欧洲高性能计算联合事业（JU）框架合作协议第800928号（欧洲处理器计划）和专项拨款协议第101036168号（EPI SGA2）的资助。JU还获得欧盟地平线2020研究与创新计划以及克罗地亚、法国、德国、希腊、意大利、荷兰、葡萄牙、西班牙、瑞典和瑞士的支持。EPI-SGA2项目PCI2022-132935也由MCIN/AEI /10.13039/501100011033和欧盟下一代欧盟/PRTR共同资助。Marc Casas部分获得了由MCIN/AEI/10.13039/501100011033资助的RYC-2017-23269项目以及欧洲社会基金“投资未来”计划的支持。本研究获得了由MCIN/AEI/10.13039/501100011033资助的PID2019-107255GB-C21项目支持。作者感谢加泰罗尼亚政府研究与大学部对研究团队“新型架构的性能理解、分析与仿真/模拟”（编号：2021 SGR 00865）的支持。

参考文献

[1] 2023. 源代码和评估文件。https://doi.org/10.5281/zenodo.7574444.[2] Ilya Afanasyev 和 Vladimir V. Voevodin. 2020. 为 NEC SX-Aurora TSUBASA 开发高效的连通分量算法实现。*Lobachevskii* 数学杂志41 (08 2020), 1417 1426 10 1134 1995080220080028 – 。[3] https://doi.org/ 。/S Ilya V. Afanasyev、Vladimir V. Voevodin、Kazuhiko Komatsu 和 Hi- roaki Kobayashi. 2020. 为 NEC SX-Aurora TSUBASA 开发一种高效且适合向量处理的广度优先搜索算法实现。载于并行计算技术，Leonid Sokolinsky 和 Mikhail Zymbler（编）。Springer International Publishing，Cham，131–145。[4] Kadir Akbudak 和 Cevdet Aykanat. 2017. 在众核架构上利用局部性进行稀疏矩阵-矩阵乘法。*IEEE* 并行与分布式系统汇刊28, 8 (2017), 2258 2271 10 1109 2017 2656893 – 。[5] https://doi.org/ 。/TPDS. 。Kadir Akbudak、Oguz Selvitopi 2018 和 Cevdet Aykanat. 。并行稀疏矩阵-矩阵乘法的扩展划分模型。*ACM* 并行计算汇刊4, 3，文章 13（2018 年 1 月），34 页。https://doi.org/ 10 1145 3155292 。/ Pham Nguyen Quang Anh、Rui Fan [6] 和 Yonggang Wen. 2016. 平衡哈希与高效 GPU 稀疏通用矩阵-矩阵乘法。载于*2016* 年国际超级计算会议论文集（土耳其伊斯坦布尔）*(ICS '16)*。计算机协会，纽约，NY，美国，文章 36，12 页。https://doi.org/10.1145/2925426.2926273[7] Adrià Armejach、Helena Caminal、Juan Cebrian、Rubén Langarita、Rekai González-Alberquilla、Chris Adeniyi-Jones、Mateo Valero、Marc Casas 和 Miquel Moretó. 2020. 使用 Arm 可扩展向量扩展模板代码。《超级计算学报》76（2020年3月），2039–2062。https://doi.org/10.1007/s11227-019-02842-5[8] Ariful Azad、Grey Ballard、Aydin Buluç、James Demmel、Laura Grigori、Oded Schwartz、Sivan Toledo 和 Samuel Williams。2016。利用稀疏矩阵-矩阵乘法中的多层并行性。*SIAM*科学计算杂志38，6（2016），C624–C651。https://doi.org/10.1137/15M104253X arXiv:https://doi.org/10.1137/15M104253X [9] Ariful Azad、Aydin Buluç 和 John Gilbert。2015。利用矩阵代数进行并行三角形计数与枚举。载于*2015 IEEE*并行与分布式处理国际研讨会研讨会。804 811 10 1109 2015 75 –。https://doi.org/。/[10] IPDPSW。Grey Ballard、Christopher Siefert 和 Jonathan Hu。2016。通过代数多重网格减少稀疏矩阵乘法的通信开销。*SIAM*科学计算杂志38，3（2016），C203–C231。https://doi.org/10.1137/15M1028807 arXiv:https://doi.org/ 10 1137 15 1028807 。/ M Nathan Bell、Steven Dalton 和 Luke N. Olson。2012。在代数多重网格方法中揭示细粒度并行性。*SIAM*科学计算杂志34，4（2012），C123–C152。https://doi.org/10.1137/ 110838844 10 1137 110838844 arXiv:https://doi.org/。[11] / Berenger Bramas。[12] 2017。一种在 Intel Skylake 上使用 AVX-512 向量化的新型混合快速排序算法。《国际高级计算机科学与应用杂志》8，10（2017）。https://doi.org/ 10 14569 2017 081044 。/IJACSA。Aydin Buluc 和 John [13] R. Gilbert。。并行稀疏矩阵-矩阵乘法的挑战与进展。载于*2008*年第*37*届国际并行处理会议。503–510。https://doi.org/10.1109/ICPP.2008.45[14] RISC-V社区。2022。RISC-V向量扩展。https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc。[15] Steven Dalton、Sean Baxter、Duane Merrill、Luke Olson 和 Michael Garland。2015。使用归并路径优化 GPU 上的稀疏矩阵运算。载于*2015 IEEE*并行与分布式处理国际研讨会。407 416 10 1109 2015 98 –。[16] https://doi.org/。/IPDPS。Steven Dalton、Nathan Bell、Luke Olson 和 Michael Garland。2014。Cusp：用于稀疏矩阵和图计算的通用并行算法。http://cusplibrary.github.io/ 0.5.0版。[17] Steven Dalton、Luke Olson 和 Nathan Bell。2015。针对 GPU 优化稀疏矩阵 - 矩阵乘法。*ACM* 数学软件汇刊 41, 4, 25 2015 20 10 1145 2699470 文章 [18] 2019（10月），页数。https://doi.org/。/ Timothy A. Davis、Mohsen Aznaveh 和 Scott Kolodziej。。快速编写，快速运行：通过 SuiteSparse:GraphBLAS，仅用20分钟开发稀疏深度神经网络。载于*2019 IEEE*高性能极限计算会议（*HPEC*）。1–6。https://doi.org/ 10 1109 2019 8916550 。/HPEC。Timothy A. Davis 和 [19] Yifan Hu。。佛罗里达大学稀疏矩阵集合。*ACM*数学软件汇刊38，1，文章 1（2011年12月），2011 25 10 1145 2049662 2049663 页。https://doi.org/。/。[20] Mehmet Deveci、Christian Trott 和 Sivasankaran Rajamanickam。2018。面向多核和 GPU 架构的多线程稀疏矩阵矩阵乘法。*Parallel Comput.* 78（2018），33–46。https://doi.org/ 10 1016 2018 06 009 。/j.parco。。John R. Gilbert、[21] Cleve Moler 和 Robert Schreiber。1992。MATLAB中的稀疏矩阵：设计与实现。*SIAM J. Matrix Anal. Appl.* 13，1（1992），333–356。https://doi.org/10.1137/0613024 arXiv:https://doi.org/ 10 1137 0613024 。/ Constantino Gómez、Filippo [22] Mantovani、Erich Focht 和 Marc Casas。2021。长向量架构上的高效稀疏矩阵-向量乘法。载于第*26*届*ACM SIGPLAN*并行编程原理与实践研讨会论文集（线上活动，韩国）（*PPoPP '21*）。计算机协会，纽约，美国，292–303。https://doi.org/10.1145/3437801.3441592

[23] Fred G. Gustavson。1978。稀疏矩阵的两种快速算法：乘法与置换转置。*ACM*数学软件汇刊

4, 3 1978 250 269 10 1145 355791 355796 （9月），–。[24] https://doi.org/。/。Guyue Huang、Guohao Dai、Yu Wang 和 Huazhong Yang。2020。GE-SpMM：面向图神经网络的 GPU 通用稀疏矩阵-矩阵乘法。载于*SC20*：高性能计算、网络、存储与分析国际会议。

1 12 10 1109 41405 2020 00076 –。https://doi.org/。/[25] SC。Md Taufique Hussain、Oguz Selvitopi、Aydin Buluç 和 Ariful Azad。2021。极大规模下的通信规避与内存受限稀疏矩阵-矩阵乘法。载于*2021 IEEE*并行与分布式处理国际研讨会（*IPDPS*）。90–100。https://doi.org/ 10 1109 49936 2021 00018 。/IPDPS。[26] Raehyun Kim、Jaeyoung Choi 和 Myungho Lee。2019。使用 Intel AVX-512 自动调优优化并行 GEMM 例程。载于高性能计算亚太地区国际会议论文集（中国广州）（*HPC Asia 2019*）。计算机协会，纽约，美国，101–110。https://doi.org/ 10 1145 3293320 3293334 。/。[27] Penporn Koanantakool、Ariful Azad、Aydin Buluç、Dmitriy Morozov、Sang-Yun Oh、Leonid Oliker 和 Katherine Yelick。2016。通信规避并行稀疏-稠密矩阵-矩阵乘法。载于*2016 IEEE*并行与分布式处理国际研讨会（*IPDPS*）。842–853。https://doi.org/10.1109/IPDPS.2016.117[28] Kazuhiko Komatsu、Shintaro Momose、Yoko Isobe、Osamu Watanabe、Akihiro Musa、Mitsuo Yokokawa、Toshikazu Aoyama、Masayuki Sato 和 Hiroaki Kobayashi。2018。向量超级计算机 SX-Aurora TSUBASA 的性能评估。载于*SC18*：高性能计算、网络、存储与分析国际会议。 685 696 10 1109 2018 00057 –。https://doi.org/。/[29] SC。Rakshith Kunchum、Ankur Chaudhry、Aravind Sukumaran-Rajam、Qingpeng Niu、Israt Nisa 和 P. Sadayappan。2017。提升 GPU 上稀疏矩阵-矩阵乘法性能。载于国际超级计算会议论文集（美国伊利诺伊州芝加哥）（*ICS '17*）。计算机协会，纽约，美国，文章

14, 11 10 1145 3079079 3079106 页。[30] https://doi.org/。/。Jeongmyung Lee、Seokwon Kang、Yongseung Yu、Yong-Yeon Jo、Sang-Wook Kim 和 Yongjun Park。2020。针对大规模稀疏网络优化基于GPU的稀疏矩阵乘法。载于*2020 IEEE*第*36*届数据工程国际会议（*ICDE*）。925–936。https://doi.org/

10 1109 48307 2020 00085 。/ICDE。Jiayu Li、[31] Fugang Wang、Takuya Araki 和 Judy Qiu。2019。面向向量引擎和图应用的广义稀疏矩阵-矩阵乘法。载于*2019 IEEE/ACM*以内存为中心的高性能计算研讨会（*MCHPC*）。33–42。https://doi.org/10.1109/MCHPC49590.2019.00012[32] Kenli Li、Wangdong Yang 和 Keqin Li。2015。基于概率建模的 GPU 上稀疏矩阵-向量乘法性能分析与优化。*IEEE* 并行与分布式系统汇刊26，1（2015），196–205 10 1109 2014 2308221 。https://doi.org/。/[33] TPDS。Roktaek Lim、Yeongha Lee、Raehyun Kim 和 Jaeyoung Choi。2018。使用 AVX-512 在 Intel KNL处理器上实现矩阵-矩阵乘法。集群计算21，4（2018年12月），1785–1795。https://doi.org/10.1007/s10586-018-2810-y [34] Weifeng Liu 和 Brian Vinter。2015。面向 GPU 和异构处理器的通用稀疏矩阵-矩阵乘法框架。并行与分布式计算期刊85（2015），47–61。https://doi.org/10.1016/j.jpdc.2015.06.010 IPDPS 2014 数值与组合算法精选论文。[35] Duane Merrill 和 Michael Garland。2016。使用CSR存储格式的基于归并的稀疏矩阵-向量乘法（SpMV）。载于第*21*届*ACM SIGPLAN*并行编程原理与实践研讨会论文集（西班牙巴塞罗那）（*PPoPP '16*）。计算机协会，纽约，美国，文章43，共2页。https://doi.org/10.1145/2851141.2851190[36] Francesco Minervini、Oscar Palomar、Osman Unsal、Enrico Reggiani、Josue Quiroga、Joan Marimon、Carlos Rojas、Roger Figueras、Abra- ham Ruiz、Alberto Gonzalez、Jonnatan Mendoza、Ivan Vargas、César Hernandez、Joan Cabre、Lina Khoirunisya、Mustapha Bouhali、Julian Pavon、Francesc Moll、Mauro Olivieri、Mario Kovac、Mate Kovac、Leon Dragic、Mateo Valero 和 Adrian Cristal。2023。Vitruvius+：一种面向高性能计算应用的面积高效RISC-V解耦向量协处理器。*ACM Trans. Archit. Code Optim.* 20，2，文章

28 2023 25 10 1145 3575861 （mar），页数。[37] https://doi.org/ 。/Yusuke Nagasaka、Satoshi Matsuoka、Ariful Azad 和 Aydın Buluç。2019。多核和众核处理器上稀疏矩阵-矩阵乘积的性能优化、建模与分析。*Parallel Comput.* 90，C（2019年12月），13页。https://doi.org/10.1016/j. parco.2019.102545[38] Yuyao Niu、Zhengyang Lu、Meichen Dong、Zhou Jin、Weifeng Liu 和 Guangming Tan。2021。TileSpMV：GPU上的稀疏矩阵-向量乘法分块算法。载于*2021 IEEE*并行与分布式处理国际研讨会（*IPDPS*）。68–78。https://doi.org/10. 1109/IPDPS49936.2021.00016[39] Yuyao Niu、Zhengyang Lu、Haonan Ji、Shuhui Song、Zhou Jin 和 Weifeng Liu。2022。TileSpGEMM：GPU上的并行稀疏通用矩阵-矩阵乘法分块算法。载于第*27*届*ACM SIGPLAN*并行编程原理与实践研讨会论文集（韩国首尔）（*PPoPP '22*）。计算机协会，纽约州纽约市，美国，90–106。https://doi.org/ 10.1145/3503221.3508431[40] Mathias Parger、Martin Winter、Daniel Mlakar 和 Markus Steinberger。2020。spECK：通过轻量级分析加速GPU稀疏矩阵-矩阵乘法。载于*PPoPP '20*：第*25*届*ACM SIGPLAN*并行编程原理与实践研讨会，*2020*年*2*月*22*日至*26*日，美国加利福尼亚州圣迭戈，Rajiv Gupta 和 Xipeng Shen（编）。ACM，

362 375 10 1145 3332466 3374521 –。[41] https://doi.org/ 。/ 。Alejandro Rico、José A. Joao、Chris Adeniyi-Jones 和 Eric Van Hensbergen。2017。ARM HPC生态系统与向量的再兴起：特邀论文。载于计算前沿会议论文集，*CF'17*，意大利锡耶纳，*2017*年*5*月*15*日至*17*日。ACM，329–334。https: //doi.org/

10 1145 3075564 3095086 。/ 。Yousef Saad。 。稀疏线性系统的迭代方法（第二版）。SIAM。https://doi.org/[42] 2003 10 1137 1 9780898718003 。/ 。Avinash Sodani、Roger Gramunt、Jesus Corbal、Ho-Seop Kim、Krishna Vinod、Sundaram Chinthamani、Steven Hutsell、Rajat Agarwal 和 Yen-Chen Liu。2016。Knights Landing：第二代Intel Xeon Phi产品。*IEEE Micro* 36（2016）。https://doi.org/10.1109/MM.2016.25[44] Nigel Stephens、Stuart Biles、Matthias Boettcher、Jacob Eapen、Mbou Eyole、Giacomo Gabrielli、Matt Horsnell、Grigorios Magklis、Alejandro Martinez、Nathanael Premillieu 等。2017。ARM 可扩展向量扩展。*IEEEmicro* 37，2（2017），26–39。[45] Yangzihao Wang、Andrew Davidson、Yuechao Pan、Yuduo Wu、Andy Riffel 和 John D. Owens。2016。Gunrock：GPU上的高性能图处理库。载于第*21*届*ACM SIGPLAN*并行编程原理与实践研讨会论文集（西班牙巴塞罗那）（*PPoPP '16*）。计算机协会，纽约州纽约市，美国，文章11，12页。https: // doi.org/ 10 1145 2851141 2851145 。/ 。Martin Winter、[46] Daniel Mlakar、Rhaleb Zayer、Hans-Peter Seidel 和 Markus Steinberger。2019。GPU上的自适应稀疏矩阵-矩阵乘法。载于第*24*届并行编程原理与实践研讨会论文集（哥伦比亚特区华盛顿）（*PPoPP '19*）。计算机协会，纽约州纽约市，美国，

68 81 10 1145 3293883 3295701 –。https://doi.org/ 。/[47] 。Zhen Xie 、 Guangming Tan 、 Weifeng Liu 和 Ninghui Sun。2022。面向多核和众核架构的基于模式的SpGEMM库。*IEEE* 并行与分布式系统汇刊33，1

（ 2022 159 175 10 1109 2021 3090328），–。

[48] 2018 https://doi.org/ 。/TPDS。。Yohei Yamada 和 Shintaro Momose。。NEC全新超级计算机SX-Aurora TSUBASA的向量引擎处理器。载于高性能芯片研讨会论文集，*Hot Chips*，第30卷。19–21。[49] Marco Zagha 和 Guy E. Blelloch。1991。面向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（美国新墨西哥州阿尔伯克基）（*Supercomputing '91*）。

计算机协会，纽约，纽约州，美国，712–721。https://doi.org/ 10 1145 125826 126164 。/ 。

[50] Dong Zhong、Qinglei Cao、George Bosilca 和 Jack Dongarra。2022。使用长向量扩展进行 MPI 归约。*Parallel Comput.* 109 (2022), 102871。https://doi.org/10.1016/j.parco.2021.102871