Julia GraphBLAS 与非阻塞执行等价物

Pascal Costanza*, Timothy G. Mattson†, Raye Kimmerer‡, Benjamin Brock§
* Independent researcher, Sint Truiden, Belgium
† University of Bristol, Ocean Park, WA
‡ National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory, Berkeley, CA
§ Intel, Parallel Computing Lab, San Francisco, CA

摘要—从一开始，GraphBLAS 就被设计为“非阻塞执行”；也就是说，调用 GraphBLAS 方法时，一旦验证了方法的参数并定义了一个图 BLAS 操作的有向无环图（DAG），就会立即返回。这使得 GraphBLAS 实现可以融合函数、消除不必要的对象、利用并行性，并进行任何额外的保持 DAG 不变的转换。存在一些使用非阻塞执行但范围有限的 GraphBLAS 实现。在这篇论文中，我们描述了我们的工作，以实现具有激进非阻塞执行支持的 GraphBLAS。我们展示了 Julia 编程语言的特点如何大大简化了非阻塞执行的实现。这正在进行的工作足以展示非阻塞执行的潜力，并且仅限于支持 PageRank 所需的 GraphBLAS 方法。

I. 介绍

C GraphBLAS API 的初始定义 [8] 指定了一个非阻塞执行模型。程序顺序中的 GraphBLAS 库函数调用定义了一个有向无环图 (DAG)。如果初始化 GraphBLAS 库以进行非阻塞执行，GraphBLAS 实现可以利用惰性求值、操作融合或任何其他满足 DAG 定义语义的执行策略。在 GraphBLAS 2.1 规范 [6] 中，限制多线程执行的功能得到了解决，因此可以为多线程执行定义 DAG。

虽然有限形式的无阻塞执行已被实现 [12], [18] ，但我们不了解任何利用编译器基础无阻塞执行的实现。

在这篇论文中，我们报告了在 Julia 中实现非阻塞 GraphBLAS 的工作。我们将我们的实现称为 applicative GraphBLAS 或*应用程序 GrB*。我们使用*多阶段编程* [22] 在运行时生成代码的符号表示，动态编译它，然后执行它。Julia 对多阶段编程的支持对于这项工作至关重要。它让我们能够在运行时从 GraphBLAS 方法树的第一级表示生成并编译代码，并支持激进的内联和融合到紧密循环中。

II. 非阻塞执行

GraphBLAS C API [7] 定义了一系列作用于矩阵、向量和标量对象的图 *BLAS* 操作。程序中（在*程序顺序*中）有序的 GraphBLAS 操作序列定义了一个有向无环图（DAG），其中节点为 GraphBLAS 操作，边为操作之间的依赖关系。初始化 GraphBLAS 库时，用户可以选择两种执行模式之一：

- **阻塞模式**：当任何 GraphBLAS 操作返回时，其执行已完成，任何副作用都已完全解决，并且相关联的 GraphBLAS 对象已完全具体化。
- **非阻塞模式**：当输入参数被验证后，GraphBLAS 操作可能会返回，但在此之前计算尚未开始。通过边传递的对象虽然计算尚未完成，但仍可用于有向无环图中后续的操作。

非阻塞执行提供了优化 DAG 执行所需的灵活性。执行可以推迟到整个 DAG 定义完成，允许重写规则融合操作、提取并行性、消除未使用的对象或中间结果，并采用其他方法来优化 DAG。DAG 的执行结果在阻塞模式和非阻塞模式下必须相同（除非由于 IEEE-754 算术中的舍入导致的非结合性等影响）。

III. Julia 中的无阻塞执行

AppGrB 使用*多阶段编程* [22] 来生成源代码的符号表示，并在执行前动态编译。多阶段编程的一个很好的例子是 Strymonas 库中的流式计算 [16]。一些编程语言直接支持多阶段编程，包括 BER MetaOCaml、Scala、Common Lisp 和 Julia。在 Julia 中，可以通过*引用*和*插值*在运行时生成代码，并通过*评估*函数在运行时编译这些代码。在 AppGrB 中，我们维护一个显式

的 GraphBLAS 方法 DAG 表示。当需要编译时，DAG 被转换成符号代码表示，即时编译并执行。这会在调用 GraphBLAS 等待方法时发生，以确保生成 GraphBLAS 对象。

考虑用于两个向量乘法的 GraphBLAS 代码 `z = ewise_mult(*, x, y)`。生成代码的内部循环如图所示 1。[1]。作为另一个示例，考虑一个修改过的 GraphBLAS 代码片段，它将 1 加到第二个输入向量的每个条目上 `z = 逐元素乘 (*, x, 对 (y 应用 (x) -> x+1))`。生成代码的内部循环如图所示 2。在这种情况下，内联 lambda 表达式立即应用于参数。编译器可以轻松地优化掉那个函数调用。这种技术在流抽象中广为人知 [16]，并且可以在 GraphBLAS 方法集合中应用。我们基于两个关于高效 JIT 非阻塞执行的额外观察来行动。

1) 为了获得最佳性能，我们必须最小化动态编译，特别是在内部循环中。因此，必须缓存并重用 JIT 编译的内核。我们将 *GraphBLAS 方法签名*定义为缓存中的键，以查找这些之前已编译的内核。
2) 我们需要针对特定内核的有效算法。例如，在逐元素乘法中，最好循环遍历非零元最少的矩阵的索引。然而，在生成代码时无法知道矩阵的稀疏性。更糟糕的是，如果只有一张矩阵被具体化了，那么迭代其索引可能不是明智之举，因为另一张矩阵可能会有更少的非零元素。这些问题在掩码计算中也会出现，因为这些掩码本身可能尚未具体化。因此，我们使用*估计的填充比率*基于对 GraphBLAS 方法表示的快速计算来驱动算法选择。

我们在下面提供更多有关 AppGrB 的一般方法、多阶段编程、编译方法表示缓存以及估计填充比率的细节。

A. 一般方法

在 AppGrB 中，用户将数据存储在 `GrB` 标量`{T}`、`GrB` 向量`{T, I}`和 `GrBMatrix{T, I}`容器中，其中 `T` 是容器中存储值的标量类型，而 `I` 是用于索引的整数类型。这些面向用户的容器可能包含实际的矩阵本身或稍后将用于生成矩阵的方法树的树表示。这实现为一个包含对摘要 `GrBScalar{T}`、摘要 `GrBVector{T, I}` 或摘要 `GrBMatrix{T, I}`子类型引用的结构。例如对于向量，具体化的子类型包括稀疏向量`{T, I}`、位集向量`{T, I}`和全向量`{T, I}`。

GraphBLAS 方法不会立即执行，而是被延迟以支持非阻塞执行。为了实现这一点，AppGrB 中的 GraphBLAS 方法没有副作用，这与 GraphBLAS C 规范不同，在 GraphBLAS C 规范中，结果会破坏性地赋值给方法参数之一。相反，AppGrB 中的 GraphBLAS 方法返回新的子类型，这些子类型是摘要 `GrB` 容器类型的子类型，并表示方法涉及的计算。

例如，当使用身份值、加法和乘法运算符以及两个输入容器调用 `mxv` 时，会返回一个 `MxV{T, I}`结构的实例，它是摘要 `GrBVector{T, I}`的子类型，如图 3 所示。此实例包含字段，这些字段引用身份值、运算符和两个输入容器，以及其他相关的信息，并且被封装在一个 `GrBVector{T, I}`中。

GraphBLAS 方法既可以在已实现的容器上调用，也可以在未实现的容器上调用，形成以特定（未实现）方法表示为根、已实现容器表示为其叶节点的树。当通过调用等待请求此类方法树的实现时，该树会被翻译成源代码的符号表示形式，动态编译并执行，如下所述。

B. 多阶段编程

多阶段编程是一种元编程技术，其中代码在运行时生成，动态编译，然后执行。为了支持多阶段编程，必须有一种方法来符号化表示代码，并且需要机制来构建这样的表示形式。Python 的*评估*可以被视为一种非常简单的多阶段编程形式。代码以字符串的形式表示，而 Python 的*评估*函数负责将代码转换为字节码并执行该字节码。然而，这种方法并不复杂：代码表示不是符号化的，这使得代码构建变得繁琐。

适当的多阶段编程使用某种形式的*引用*（有时称为准引用 [4]）来构造和表示代码，这可以理解为一种代码模板。例如，在 Julia 中，`:(2 + 2)` 是表示两个数相加计算的引用代码。*插值*（有时称为*拼接* [4]）允许我们将值插入这种引用代码中。例如，`:(2 + $x)` 表示将一个数字加到 `x` 绑定的值上的代码。每一段引用的代码都是一个可以打印、绑定到变量上、检查以及传递给函数以进一步构建代码的一等值。为了执行表示的代码，Julia 的*评估*函数将翻译后的代码编译成机器语言（使用 LLVM），然后执行它。要一次性编译一段代码，但

[1]变量名是带有数字的生成符号，以防止意外捕获名称。我们的演示通过避免这些以及其他低级细节而得到简化。

图 1. 生成的内循环用于 z = ewise_mult(*, x, y)。

```
for col_1 = 1:nof_cols_2
    result_3[col_1] = vector_ref_4.values[col_1] * vector_ref_5.values[col_1]
end
```

图 2. 生成的内循环为 z = ewise_mult(*, x, apply((x) -> x + 1), y)。

```
for col_1 = 1:nof_cols_2
    result_3[col_1] = vector_ref_4.values[col_1] * ((x) -> x + 1)(vector_ref_5.values[col_1])
end
```

图 3. 乘以 V 结构体。

```
struct MxV{DC, DA, DB, Index <: Integer}
                    <: AbstractGrBVector{DC, Index}
    ncols::Int
    A::GrBMatrix{DA, Index}
    B::GrBVector{DB, Index}
    identity::DC
    add::Any
    mul::Any
end
```

随后多次执行它，可以将引用的代码表示为 lambda 表达式。考虑以下片段：

```
x = 42
f = eval(:(() -> print($x)))
f()  # prints 42
x = 11
f()  # prints 42
```

请注意，由于 x 的值是插值得到的，构造的代码会打印字面值 42，无论随后将哪个值赋给 x。为了确保代码始终打印 x 的当前值，不应对其进行插值（即，应提到 x 而不带前面的$）。换句话说，仔细嵌套引用和插值精确地标记了代码中的哪些部分应在哪个阶段执行：在代码构造期间，还是在稍后的代码执行中。

AppGrB 方法树通过多阶段编程转换为引用代码，其中各种摘要 GrB 类型的每个子类型都为其生成的代码贡献自己的逻辑，包括具体化和非具体化的表示形式。因此，对等待的调用通常会生成一个或两个外部循环以产生结果容器，并将嵌套的方法融合到该循环中。

每个类型摘要 GrB 的子类型都必须实现一组函数，用于表达遍历容器元素的方法。例如，为了迭代非零项，它们必须实现一个每个条目函数。然而，与其直接执行迭代，它生成了迭代的符号表示。参见图 4 以了解全向量和稀疏向量两种情况下每个条目的实现，以及 GraphBLAS 应用方法。每个条目函数接收需要为其专门化生成代码的容器、包含生成代码所需的周围代码中的容器名称，以及一个块函数，该函数传递每个迭代中相应列和值的表达式。后者块函数可以生成进一步的代码。请注意，每条记录方法对向量应用进行递归调用每个条目的方式 — 即，在其基础上进行递归调用 — 也就是说，在向量上对其值应用相应的操作符 — 并且传递给该递归调用的函数包裹了内部迭代器中的相应值，然后调用了外部迭代器接收到的块函数。

在图 5 中，我们展示了一个示例，说明如何使用这个迭代器协议来实现一个任意的 GraphBLAS 方法树，最终生成一个向量。它调用了一个（由 AppGrB 定义的）编译函数，该函数带有向量的签名，并且还有一个能够生成符号代码表示的函数，最终可以返回一个实现了的向量。这个代码是一个 lambda 表达式，期望接收向量方法树，设置低级别的索引和值向量，在其代码中融合了一个循环（由每个条目生成），最后创建一个稀疏的 GraphBLAS 向量。

请注意，这里展示的代码已被简化。还有许多其他迭代选项，包括仅对索引进行迭代而不对值进行迭代，仅对值进行迭代，逐步迭代（使用第一和下一个方法）以在两个输入容器之间交替（例如用于元素级加法），以及针对某些特定乘法情况的随机访问。这里展示为单个每个条目函数的内容被分解为设置、迭代和更细化的索引及值访问器，包括仅对等值容器中的等值访问一次。AppGrB 还允许顺序和并行迭代，这需要生成不同形式的循环。

C. 编译方法树的缓存

多阶段编程可以用于在运行时为特定的 GraphBLAS 方法树生成代码。生成的代码考虑了方法调用的

图 4. 每个条目的全量和稀疏实体向量实现，以及 GraphBLAS 应用方法的实现。

```
function foreach_entry(vec::FullVector{T, Index}, vec_name, block) where {T, Index <: Integer}
  @gensym vec_ref col
  :(let $vec_ref::FullVector{$T, $Index} = $vec_name.ref
      for $col in 1:$vec_ref.ncols
        $(block(col, :($vec_ref.values[$col])))
      end
    end)
end

function foreach_entry(vec::SparseVector{T, Index}, vec_name, block) where {T, Index <: Integer}
   @gensym vec_ref index
   :(let $vec_ref::SparseVector{$T, $Index} = $vec_name.ref
       for $index in eachindex($vec_ref.indices)
         $(block(:($vec_ref.indices[$index]), :($vec_ref.values[$index])))
       end
     end)
end

function foreach_entry(vec::VectorApply{Out, In, Index}, vec_name, block) where {Out, In, Index <: Integer}
   @gensym vec_ref vec_apply_base
   :(let $vec_ref::VectorApply{$Out, $In, $Index} = $vec_name.ref,
         $vec_apply_base = $vec_ref.base
       $(foreach_entry(vec.base, vec_apply_base, (col, val) -> block(col, :($(vec.op)($val)))))
     end)
end
```

图 5. 图 BLAS 向量的物化。

```
function materialize(vector::AbstractGrBVector{T, Index}, ref) where {T, Index <: Integer}
  compiled = compile(signature(vector), function ()
                       @gensym vec_name indices values
                       :(($vec_name) ->
                            let $indices = Vector{$Index}(),
                                $values = Vector{$T}()
                              $(foreach_entry(vector, vec_name, (col, val) -> :(begin
                                                                                  push!($indices, $col)
                                                                                  push!($values, $val)
                                                                                end)))
                              SparseVector{$T, $Index}($indices, $values)
                            end)
                     end)
  @invokelatest compiled(ref)
end
```

运行时属性，例如传递给方法的参数的存储格式（例如，它们是稀疏还是密集），是否为同值（即对象的所有定义元素具有相同的值），与单体或半环关联的具体操作（这些可以在代码构建过程中内联），等等。

然而，当一个程序在一个方法树上调用 等待，例如在图 3 中的 mxv，AppGrB 将首先查询缓存并检索先前编译过的内核（如果存在）。为了实现这一点，我们生成了一个递归泛型 签名函数来计算此类查找的关键字。构建的代码是一个期望关联结构体实例的 lambda 表达式（如图 3 所示），并且仅适用于具有相同签名的实例。

D. 估计的填充比率

一般来说，GraphBLAS 方法结果的稀疏性（非零元素的数量）无法提前得知。当我们为两个容器逐个元素相乘生成代码时，让输入中更稀疏的容器引导整个迭代过程更为高效。然而，在 AppGrB 中，输入容器本身

可能是 GraphBLAS 方法树中的内部节点，因此输入容器的稀疏性可能不易获得。唯一可靠推断 GraphBLAS 方法结果稀疏性的方法是立即执行它，这将违背非阻塞执行的目的。不过，存在一些能够以不同程度准确度预测矩阵操作结果稀疏性的方法 [10], [2], [20], [14] 。这些预测器传统上用于估计输出数组的大小，但在这里我们使用它们来驱动算法选择。作为第一步，我们选择了 *朴素元数据估计器* [20]，基于输入数据的明显特性。例如，在乘以两个输入容器时，结果的估计稀疏性仅仅是输入容器稀疏性的乘积（占容器大小的百分比）。其他 GraphBLAS 方法也导致类似的直接估算器。对于已实例化的容器，“估计” 稀疏性只是非零元素的数量除以容器大小。存在一些例外情况。某些 GraphBLAS 选择方法使用任意用户函数来确定输入容器中的哪些元素在结果中保留下来。对于预定义的选择函数，也可以提供一个 “简单” 的估算器。总的来说，我们预见用户可能需要选项来自定义自己的估算器。

目前，我们使用如下估计器：

- 当生成输出向量时，我们使用估计器来确定结果的表示是稀疏、位集还是完整。
- 在逐元素乘法中，估计器确定两个输入容器中的哪一个驱动乘法循环。
- 在逐元素相加中，估计器确定是否使用范围涵盖完整维度的外层循环（当结果预期为密集或几乎密集时），或者是否迭代输入容器的索引。

我们期望使用掩码操作的估计器，并且已经用它们来预分配输出数组。重要的是要强调，这些估计器对涉及算法的正确性没有任何影响。当估计器不准确时，它最坏的情况是会对性能产生负面影响。

IV. 性能结果

为了表征我们在 Julia 中实现的非阻塞 GraphBLAS 的性能，我们考虑了与多个 GraphBLAS 系统一起使用的 PageRank 算法。

- **非阻塞 AppGrB**：本文中使用 Julia 描述的无阻塞 GraphBLAS。
- **应用程序分块格布式编程**：AppGrB 的非阻塞代码通过在每次操作后调用 `GrB_等待()` 被强制以阻塞模式执行；即，迫使系统等待每个操作完成，并确保每一个 GraphBLAS 对象都完整，从而与 GraphBLAS 中阻塞模式的定义相匹配。
- **应用程序抓取 C++ 存根**：为了理解由 Julia 生成的 LLVM 代码的质量，我们取了生成的 Julia 代码，并用 C++手动实现了它。
- **套件稀疏矩阵库**：我们使用了来自 GraphBLAS 版本 10.1.0 的 SuiteSparse 实现以及 LAGraph 版本 1.1.4 中的 PageRank 代码（该代码基于算法 [21]）。

图 6. AppGrB 中的 PageRank 实现。

```
function pagerank_gap(AT::GrBMatrix{Float32, Index},
                      out_degree,
                      damping,
                      tolerance,
                      itermax)
                            where {Index <: Integer}
    n = nrows(AT)
    scaled_damping = (1 - damping) / n
    teleport = scaled_damping
    rdiff = GrBScalar(1.0f0)
    t = GrBVector{Float32, Index}(n)
    r = GrBVector{Float32, Index}(n, 1.0f0 / n)
    d = GrBVector{Float32, Index}(n, 1 / damping)
    d = ewise_add(:((x, y) -> max(x / $damping, y)),
                  conv(Float32, out_degree), d)
    wait(d; parallel=true)
    rt = GrBVector{Float32, Index}(n, teleport)
    iter = 1
    while iter <= itermax && rdiff() > tolerance
        t = r
        r = ewise_mult(:(/), t, d)
        r = mxv(0.0f0, :(+), :((_, x) -> x), AT, r)
        r = ewise_add(:(+), rt, r)
        r = wait(r; parallel=true)
        t = ewise_add(:((x, y) -> abs(x - y)), t, r)
        rdiff = reduce(:(+), 0.0f0, t)
        iter += 1
    end
    (r, iter)
end
```

基准测试在一个配备有两个 Intel® Xeon® Platinum 8368 处理器和 503 GB 内存的系统上运行。每个处理器有 16 个性能核心和 22 个效率核心，两个处理器总共拥有 76 个核心。当在 GraphBLAS 方法树上调用 wait 时，Julia 程序会经历以下步骤。

1) 确定方法签名
2) 使用方法签名从编译缓存中查找可能之前已编译的代码。
3) 如果之前编译的代码不存在，则生成符号表示，将其编译为机器码，并添加到编译缓存中。
4) 执行编译后的代码。

我们运行了一次程序以填充代码缓存，这被称为预热运行，然后报告从接下来的 16 次运行中的平均运行时间。在预热运行中，上述定义的所有 4 个步骤都会执行，但在随后的运行中，跳过了第 3 步。忽略这一成本（大约三秒）使我们有了一种更一致的方式来比较 appGrB 执行与其他方法，但这通常是可以接受的，因为在实践中，一个 GraphBLAS 方法在一个程序中会被多次调用，使得第 3 步的一次性成本变得微不足道。在未来的工作中，我们将研究使用持久缓存来保存运行之间的编译代码以用于反复出现的签名。

所有四个程序都使用 32 位索引。这些索引足以支持在 GAP 基准测试 [3] 中定义的测试用例。尽管在一般情况下，SuiteSparse GraphBLAS 与 GraphBLAS C API 规范要求一致，使用 64 位索引，但它会在矩阵和向量维度范围内的时候自动将索引减少到 32 位。这是在 SuiteSparse GraphBLAS 第 10 版中添加的最近优化。

结果呈现在图 7 中。我们使用了 GAP-web.mtx 数据集 [5]，这是一个有 5060 万个顶点和 194 亿条边的有向图。它具有显著的局部性，并且顶点间平均度数很高。

比较 Julia 结果（*AppGrb* 阻塞和 *AppGrb* 非阻塞）与 C++ 存根代码，可以看到由 Julia 后端生成的 LLVM 代码有很大的改进空间。对于高线程数（76、38 和 19），最佳结果是使用 C++ 存根代码。对于较低的线程数，SuiteSparse GraphBLAS 表现更好。并行效率在 76 个线程时 AppGrB C++ 存根为 34%，而 SuiteSparse GraphBLAS 为 21%。

AppGrB 在非阻塞执行模式下明显比相同代码在阻塞模式下运行得更快（AppGrB 阻塞）。这证明了我们在 GraphBLAS 中采用非阻塞执行方法的整体价值。不仅所有线程数量下的性能一直更好，在 76 个线程时，非阻塞模式的并行效率为 0.21，而阻塞模式下为 0.18。

V. 相关工作

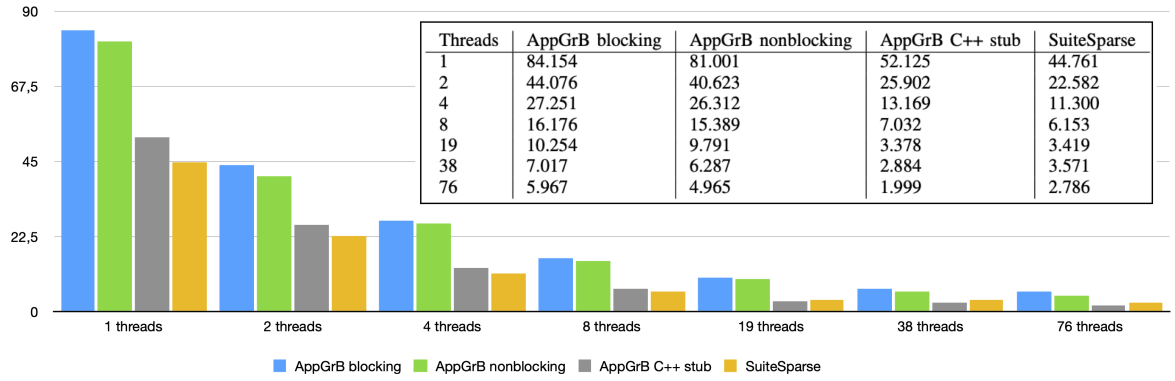
ALP/GraphBLAS 库 [18] 在非阻塞模式下执行时，支持对逐元素操作进行分块和并行执行。ALP/GraphBLAS 延迟执行，并生成一个表示操作及其迭代空间的对象链。一旦对象具体化，运行时将以分块形式遍历迭代空间，在移动到下一个块之前应用该块内的所有操作。这提高了缓存效率，但不会在生成融合内核前组装整个 DAG。相反，它依赖于 lambda 表达式来结构化分块执行，SpMV 和 SpMM 强制具体化。ALP/GraphBLAS 还动态分析操作之间的依赖关系，并将它们放置在可以并行执行的队列中。没有依赖的操作可以放入新的队列，而有依赖的操作则被放在预先存在的队列中，在必要时合并队列。

对于稀疏线性代数，向稀疏数组中更新和删除值会增加大量开销。在 SuiteSparse 的 GraphBLAS 实现中，如 [13], [12] 所述，使用非阻塞模式来减少这种开销。当一个值需要被删除时，它会被标记为*僵尸*。该值仍然存在于稀疏数据结构内部，但删除操作会被延迟。同样地，当一个值添加到稀疏矩阵中的空位置时，它可以被标记为*待处理元组*，使其在 GraphBLAS 操作中可用，但不会立即提交更新到稀疏数组本身。对这些稀疏数据结构的修改被推迟，以便以后以更优化的方式进行操作。从更大规模来看，有时操作会产生一组杂乱无章的元组来表示一个稀疏数组。与其立即将其排序为 SuiteSparse 的存储格式之一，可以延迟或在某些情况下甚至省略排序。SuiteSparse GraphBLAS 在其 JIT 中使用多阶段编程来优化用户定义的操作符。当使用用户定义的操作符时，JIT 将通过将代表用户定义的二元操作符和/或幺半群的字符串粘贴到内核骨架中，并使用外部文件编译器生成新的内核并进行编译。然后运行这个用用户定义的操作符内联生成的 JIT 编译内核，而不是使用一个通过函数指针调用操作符的预定义内核，这通常较慢。利用非阻塞执行的进一步优化是 SuiteSparse GraphBLAS 项目中的持续工作。

GBTLX [19] 使用 SPIRAL 从一个从 GBTL GraphBLAS 程序收集的执行跟踪中生成优化的 C 代码。虽然 GBTLX 在其生成的跟踪中确实将操作融合到单个内核中，但它仅针对阻塞 GraphBLAS 程序这样做，这意味着用户无法利用 GraphBLAS 非阻塞模式与 GBTLX。

PyGB [9]，一个用于 GraphBLAS 的 Python 接口，通过构建表达式树来延迟执行，这与我们的方法树类似，表示获取每个 GraphBLAS 对象所需的计算。当必须实现表达式树时，它会被用来生成一个 GBTL 程序，然后执行该程序。虽然理论上可以融合表达式树并以并行方式执行，但 GBTL 当前没有对非阻塞模式的支持，并且逐个操作地执行表达式树而不进行融合。

Julia 库 GraphBLAS.jl [15] 正在开发中。它以非阻塞模式执行惰性 DAG 融合，类似于 AppGrB。然而，它使用 Suite-Sparse GraphBLAS 来执行这个 DAG，而不



Threads	AppGrB blocking	AppGrB nonblocking	AppGrB C++ stub	SuiteSparse
1	84.154	81.001	52.125	44.761
2	44.076	40.623	25.902	22.582
4	27.251	26.312	13.169	11.300
8	16.176	15.389	7.032	6.153
19	10.254	9.791	3.378	3.419
38	7.017	6.287	2.884	3.571
76	5.967	4.965	1.999	2.786

图 7. **页面排名性能** – PageRank 和 GAP-web.mtx 数据集的 16 次运行平均时间（秒）。

是直接生成代码。因此，可以应用的优化是有限的。

最近在张量编译器 [1], [17] 方面的工作开发了通过结构化迭代融合操作的技术，其中编译器理解稀疏结构。尽管这些技术尚未应用于 GraphBLAS 操作，但我们预计我们的工作可以扩展以利用其中的一些技术。

VI. 结论与未来工作

我们使用多阶段编程在 Julia 中实现了 GraphBLAS 的 AppGrB 版本，以实现无阻塞执行。本文介绍了我们在 AppGrB 上的正在进行的工作，目前该工作仅限于足以支持 LAGraph PageRank 算法的 GraphBLAS 子集。对于所有线程计数而言，非阻塞 AppGrB 比阻塞版本更快，验证了我们对非阻塞执行的一般方法的有效性。然而，它并未在优化后的非阻塞 SuiteSparse 基准上提供加速，这可能是由于我们在生成代码中的低效所致。未来，我们计划将非阻塞功能扩展到 GraphBLAS 的其余部分，并且我们认为更复杂的操作可能为融合提供更多机会，从而为非阻塞执行带来更多的加速效果。提高生成代码的效率也将有助于缩小差距，正如我们的手工生成 C++代码（即 AppGrB C++存根）所展示的那样，在高线程计数下其速度远超 SuiteSparse。

斯特里蒙纳斯的多阶段编程方法 [16]，对我们设计的影响描述了一个在特定极端情况下存在的限制，这种情况不应出现在 GraphBLAS 中。然而，GraphBLAS 可能会带来其自身的挑战。例如，使用估计填充比率优化选择操作可能较为困难。不过，我们认为我们的方法树表示将支持矩阵链乘法的算法优化 [11]，并且我们可以利用基于瓷砖的融合 [18] 和调度到优化库 [15] 来实现更高的性能。

参考文献

[1] Willow Ahrens, Daniel Donenfeld, Fredrik Kjolstad, and Saman Amarasinghe. Looplets: A language for structured coiteration. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, CGO '23, page 41 – 54, New York, NY, USA, 2023. Association for Computing Machinery.

[2] Rasmus Resen Amossen, Andrea Campagna, and Rasmus Pagh. Better size estimation for sparse matrix products. *Algorithmica*, 69:741–757, 2014.

[3] Ariful Azad, Mohsen Mahmoudi Aznaveh, Scott Beamer, Maia P. Blanco, Jinhao Chen, Luke D'Alessandro, Roshan Dathathri, Tim Davis, Kevin Deweese, Jesun Firoz, Henry A Gabb, Gurbinder Gill, Balint Hegyi, Scott Kolodziej, Tze Meng Low, Andrew Lumsdaine, Tugsbayasgalan Manlaibaatar, Timothy G Mattson, Scott McMillan, Ramesh Peri, Keshav Pingali, Upasana Sridhar, Gabor Szarnyas, Yunming Zhang, and Yongzhe Zhang. Evaluation of graph analytics frameworks using the GAP benchmark suite. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 216–227, 2020.

[4] Alan Bawden. Quasiquotation in Lisp. In Olivier Danvy, editor, *Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas, USA, January 22-23, 1999. Technical report BRICS-NS-99-1*, pages 4–12. University of Aarhus, 1999.

[5] Scott Beamer, Krste Asanovic, and David Patterson. The GAP benchmark suite. arXiv:1508.03619, 2015.

[6] Benjamin Brock, Aydın Buluç, Raye Kimmerer, Jim Kitchen, Manoj Kumar, Timothy Mattson, Scott McMillan, José Moreira, Michel Pelletier, and Erik Welch. The GraphBLAS C API Specification, ver. 2.1. 2023.

[7] Benjamin Brock, Aydın Buluç, Timothy Mattson, Scott McMillan, and José Moreira. The GraphBLAS C API Specification. *GraphBLAS. org, Tech. Rep.*, version 1.3.0, 2019.

[8] Aydin Buluç, Tim Mattson, Scott McMillan, José Moreira, and Carl Yang. Design of the GraphBLAS API for C. In *Intr. Parallel & Distributed Processing Symposium Workshops*, pages 643–652, 2017.

[9] Jesse Chamberlin, Marcin Zalewski, Scott McMillan, and Andrew Lumsdaine. PyGB: GraphBLAS DSL in Python with dynamic

compilation into efficient C++. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 310–319, 2018.

[10] Edith Cohen. Size-estimation framework with applications to transitive closure and reachability. *Journal of Computer and System Sciences*, 55(3):441–453, 1997.

[11] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[12] Timothy A Davis. SuiteSparse GraphBLAS repository. https://github.com/DrTimothyAldenDavis/GraphBLAS.

[13] Timothy A Davis. Algorithm 1037: SuiteSparse: GraphBLAS: Parallel graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software (TOMS)*, 49(28):1–30, 2023.

[14] Zhaoyang Du, Yijin Guan, Tianchan Guan, Dimin Niu, Nianxiong Tan, Xiaopeng Yu, Hongzhong Zheng, Jianyi Meng, Xiaolang Yan, and Yuan Xie. Predicting the output structure of sparse matrix multiplication with sampled compression ratio. In *2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 483–490, 2023.

[15] Raye Kimmerer. Graphblas. jl v0. 1: An Update on GraphBLAS in Julia. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 516–519. IEEE, 2024.

[16] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, page 285 – 299, New York, NY, USA, 2017. Association for Computing Machinery.

[17] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.

[18] Aristeidis Mastoras, Sotiris Anagnostidis, and Albert-Jan N. Yzelman. Design and implementation for nonblocking execution in GraphBLAS: Tradeoffs and performance. *ACM Trans. Archit. Code Optim.*, 20(1), November 2022.

[19] Sanil Rao, Anurag Kutuluru, Paul Brouwer, Scott McMillan, and Franz Franchetti. GBTLX: A first look. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2020.

[20] Johanna Sommer, Matthias Boehm, Alexandre V. Evfimievski, Berthold Reinwald, and Peter J. Haas. Mnc: Structure-exploiting sparsity estimation for matrix expressions. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1607 – 1623, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Gábor Szárnyas, David A Bader, Timothy A Davis, James Kitchen, Timothy G Mattson, Scott McMillan, and Erik Welch. LAGraph: Linear algebra, network analysis libraries, and the study of graph algorithms. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 243–252. IEEE, 2021.

[22] Walid Taha. A gentle introduction to multi-stage programming. In Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*, pages 30–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.