arXiv:2305.03448v1 [cs.PL] 5 May 2023

Descend：一种安全的 GPU 系统编程语言

Bastian Köpcke[1], Sergei Gorlatch[2], 和 Michel Steuwer[3]

[1,2]明斯特大学[3]爱丁堡大学

摘要

图形处理单元（*GPU*）通过遵循吞吐量导向的计算范式提供了巨大的计算能力，在这种范式下，数千个计算单元并行运行。为这种大规模并行硬件编程颇具挑战。程序员必须正确且高效地协调数千个线程及其对各种共享内存空间的访问。现有主流 *GPU* 编程语言（如 *CUDA* 和 *OpenCL*）基于 *C/C++*，继承了通过原始指针访问内存这一根本不安全的方式。这使得数据竞争和死锁等容易产生却难以检测的错误变得更加普遍。

在本文中，我们介绍 *Descend*：一种安全的 *GPU* 系统编程语言。秉承 *Rust* 的理念，*Descend* 的类型系统通过跟踪所有权 和生命周期，确保 *CPU* 和 *GPU* 内存管理的安全。*Descend* 引入了一种全新的整体式 GPU 编程 模型，其中计算按照层级调度到 *GPU* 的执行资源上：网格、块和线程。*Descend* 扩展的借用 检查确保执行资源能够安全访问内存区域，而不会引入数据竞争。为此，我们引入了视图，用于描述内存区域的安全并行访问模式。

我们讨论 *De- scend* 类型系统所提供的内存安全保证，并使用多个基准测试评估 *De- scend* 的实现。结果表明，与缺少 *Descend* 安全性保证、由人工编写的 *CUDA* 程序相比，我们的实现不会引入显著的运行时开销。

1. 引言

图形处理单元（GPU）是一种大规模并行的硬件设备，其吞吐量导向的设计优先考虑由数千个协作线程并行执行的整体计算的运行时间，而非像传统 CPU 那样优先考虑单个线程的性能 [4]。因此，在科学模拟、医学成像，尤其是机器学习等对高性能要求极高的领域，GPU 成为极具吸引力的设备。

即使对于高级程序员而言，编写功能正确且高效的 GPU 软件也是一项艰巨的任务。主流的 GPU 编程语言 CUDA 和 OpenCL 都是低级命令式系统编程语言，让程序员能够精确控制每个线程如何访问内存，以及何时执行哪些计算指令。这种控制对于发挥 GPU 的预期高性能至关重要；而 GPU 中的这一点.....未优化的朴素实现与完全优化的实现之间的差异可能高达两个数量级 [8]——通常远高于 CPU 上的差异。

遗憾的是，在 CUDA 和 OpenCL 中，这种程度的控制给 GPU 程序员带来了重大挑战。由于这两种语言都基于 C/C++，它们继承了通过原始指针访问内存这一根本上不安全的方式。此外，为了协调线程并确保对内存具有一致的视图，必须正确使用手动同步原语。这容易引发一些虽容易犯下、却往往难以发现的错误，尤其是在多个线程访问同一内存位置时出现的数据竞争，以及错误使用同步原语时发生的死锁。

清单1 展示了一个 CUDA 内核函数，它在 GPU 上并行执行，用于转置矩阵。在第 4–7 行，每个线程将四个矩阵元素复制到临时缓冲区中，然后在同步之后，将转置后的元素复制到输出中。该函数的正确性取决于正确的索引，而索引众所周知地容易出错。事实上，清单1 中有一个微妙的错误：在第 5 行，`threadIdx.y+j` 应当用括号括起来，这样两个项都会乘以 32。因此会发生数据竞争，因为多个线程会在没有协调的情况下写入同一个内存位置。

Rust 已证明，系统编程语言可以在不失去底层控制能力的情况下，以内存安全的方式设计。它通过禁止线程并发访问内存资源来防止数据竞争，前提是至少有一个线程可以修改该资源 [10]。Rust 通过其类型系统强制执行这一点，具体来说是通过 借用检查，并结合所有权和生命周期的概念，主要确保内存管理安全。Rust 能否防止清单1中的错误？显然，`tmp` 由多个线程共享

CUDA

```
1  __全局__ void 转置(const double *input,
2                                  double *输出) {
3    __共享__ 浮点数 临时变量[1024];
4    对于 (整数 j  = 0; j  < 32; j  += 8)
5      tmp[ 线程Idx.y+j *32+线程Idx.x] =
6          输入[(blockIdx.y*32+threadIdx.y+j)*2048
7                          + blockIdx.x*32+threadIdx.x];
8    __syncthreads();
9    对于 (整数 j  = 0; j  < 32; j  += 8)
10     输出[(blockIdx.x*32+threadIdx.y+j)*2048
11                       + blockIdx.y*32+threadIdx.x] =
12         <escape>tmp[(threadIdx.x)*32+threadIdx.y+j];</escape>       }
```

图 1：并行执行矩阵转置的 CUDA GPU 内核。第 5 行索引中的一个细微错误导致了数据竞争。

```
Descend
n transpose(input: & gpu.global f ; ; 输出: &uniq gpu.global f ; ; ) 3 -[网
格: gpu.Grid<XY<64,64>,XY<32,8>>]-> () { 4 sched(Y,X) 代码块 in 网格 { 5
let tmp =alloc::<gpu.shared, [[f64; 32]; 32]>(); 6 sched(Y,X) 线程 in 代码
块 { 7 for i in [0..4] { 8 _ _ <32 4>[[ ]][ ] =
tmp.group by row:: , 线程 i input.group
_ _ [[ ]] by tile:: , .
transpose 代码块 .group by row:: , 线程 i }; 11 sync; 12 for i in [0..4]{
_ _ <32 32>[[ ]] 输出.group by
tile:: , 代码块 .group by row:: , 线程 i
] =
_ _ <32 4>[[ ]][ ] tmp.group by
row:: , 线程 i } 16 } } } Li
```

sting 2: A D_{escend} *d* functi$^{on\ per}$ formi$^{ng}_{ma}$ trix t$_{ranspos}$ ition.

并行执行的线程，而且显然，我们在第 5 行修改了它的内容。因此，即使不尝试检查索引是否安全，Rust 也会拒绝这个内核，因为 Rust 的类型系统无法推理多个线程并行访问数组是否安全。

本文介绍 *Descend*，一种面向 GPU 系统、借鉴并扩展 Rust 思想的安全 GPU 系统编程语言。与以往安全的 GPU 编程方案（如 Nvidia 的 Thrust [14] 或 Futhark [7], *Descend* 是一种命令式 GPU 编程语言，让程序员能够在安全保障下行使底层控制。

代码清单 2 展示了*De-scend*中的矩阵转置函数。与 CUDA 不同，该函数不会隐式地由数千个 GPU 线程执行，而是由（一个）GPU 网格执行。程序员描述计算在网格上的层次化调度：首先描述块 的调度（第 4 行），然后描述嵌套线程 的调度（第 6 行）。我们在第 5 行为每个块 分配共享内存。每个线程 执行与 CUDA 中相同的复制操作：先从输入复制到临时缓冲区，再在同步之后复制回输出。*Descend* 中的程序员不使用原始索引，而是使用内存视图来描述对内存的并行访问。*Descend* 会静态检查对视图的访问是否安全，并在类型系统中对其进行特殊处理。这会将内存访问限制为安全的并行访问模式，同时仍允许组合视图来描述复杂的内存访问。对于这个示例，*Descend* 的借用检查能够静态确定，对共享临时缓冲区和输出的并行写入访问是安全的。同样，*Descend*会静态强制正确使用同步，避免遗漏同步或将其放置在错误的位置。

Descend 是一种面向由 CPU 和 GPU 组成的异构系统的整体式编程语言。CPU 和 GPU 的物理分离内存体现在引用的类型中，*Descend* 会强制确保这些引用只在正确的执行上下文中解引用。函数会标注执行资源 （如第 3 行的函数签名所示），以指明函数的执行方式。这些标注将 CUDA 中隐含的关于执行内核的线程数和块数的重要假设显式化，并使类型系统能够强制执行这些假设。

总而言之，本文做出了以下贡献：

- 我们介绍了*Descend*，这是一种秉承 Rust 理念的安全 GPU 系统编程语言；
- 我们指出了 GPU 编程面临的挑战，并讨论了 *Descend* 如何帮助应对这些挑战（第2节）；
- 我们讨论了 执行资源、位置表达式和内存视图这些概念如何确保安全（第3节）；
- 我们介绍了 *Descend*的形式化类型系统和扩展借用检查（第4节）；
- 并通过实验评估表明，使用 *Descend* 编写的程序可达到与功能等效的 CUDA 程序相同的性能，而后者缺乏*Descend*的安全性保证（第5节）。

我们将在第6 节讨论相关工作，并在第7节作出总结。

2. GPU 编程的挑战

GPU 编程带来了许多挑战，我们将在本节中讨论这些挑战。我们将其分为两个方面：*1)*源于处理 GPU 的执行层次结构和内存层次结构的挑战，例如成千上万的线程被分组到代码块中，并访问各种 GPU 内存；以及*2)* 源于协调异构系统的挑战，例如在 CPU 和 GPU 内存之间传输数据。在讨论每个方面之前，我们先简要概述由 CUDA 建立的传统 GPU 编程模型。

2.1. CUDA GPU 编程模型

在 CUDA 中，程序员编写内核函数，让它们在 GPU 上并行执行，通常对多维数组进行数据并行计算。这些函数由数千个线程执行，所有线程都运行相同的代码。因此，乍看之下，CUDA 编程模型类似于传统的数据并行编程模型，即同一条指令以锁步方式应用于多个数据元素。然而，在 CUDA 中，这一严格要求有所放宽，因为内核代码可以根据线程索引进行分支。线程索引用于标识各个线程，通常用于数组索引，使每个线程处理不同的数组元素。线程索引是普通整数，用于对普通 C 风格数组进行索引计算，这使得静态检查并行内存访问是否安全颇具挑战，也容易因疏忽引入错误而导致数据竞争。此外，编写内核时往往会对执行内核的线程数量作出隐式假设，因此若不了解这些假设，代码就难以理解；而当 CPU 代码与 GPU 代码开始出现分歧时，这也会成为另一个 bug 来源。

GPU 由多个支持多线程的流式多处理器（SM）组成，每个流式多处理器都能够同时执行多个

线程。这种硬件设计也应体现在软件中。因此，线程 按层次组织成组，由流式多处理器独立执行。在 CUDA 中，这样的线程组称为代码块。所有代码块的集合称为网格。

同样，内存也采用层次化组织，并且与执行层次结构紧密相关。在软件中，不同的地址空间 反映了 GPU 内存的不同种类。速度最慢、容量最大的内存是全局内存，整个网格中的每个线程都可以访问它。每个代码块都提供了速度较快的共享内存，代码块中的每个线程都可以访问它。最后，每个线程都可以独占访问自己的速度最快的私有内存。从主机传输到 GPU 的数据始终存储在全局内存中。为了利用速度更快的内存，必须在不同地址空间之间显式复制数据。

2.2. 执行与内存层次结构的挑战

CUDA 编程模型及其执行和内存层级与 GPU 硬件十分相似，能够实现 GPU 程序的可扩展性，但也带来了两大挑战：如何避免数据竞争，以及如何正确同步代码块中的线程。

数据竞争。当两个或多个线程同时访问同一个内存位置，且至少有一个线程向该内存位置写入数据时，就会发生数据竞争。在 CUDA 中，很容易产生数据竞争，请看以下代码：

```
CUDA
__全局__ 无返回值反转_每_代码块(双精度 *数组) { 双精度 *代码块_部分
 = &数组[代码块索引.x * 代码块维度.x];  _  [  ] =
代码块部分线程索引.x代码块部分代码块维度.x-1 - 线程索引.x];  }
```

在此示例中，输入 `array` 被拆分为每个代码块各自独立的部分。然后，每个代码块中的线程按照线程索引的逆序访问单个元素，并将该值写回数组中与其线程索引对应的位置。这会造成数据竞争：某个线程可能仍在读取一个索引处的值，而另一个线程已经在向该处写入。

在 *Descend* 中，编译器会识别出存在数据竞争的可能性，并通过一条错误信息拒绝该程序：

```
Descend
错误：内存访问冲突   | arr[[线程]] = arr.rev[[线程]];
  |                 ^^^^^^^^^^^^^^^^^^-------------------
  | ^^^^^^^^^^^^^由于以下原因，无法选择内存 |  | 此处存
在冲突的先前选择：--
```

我们将在第 3 节中说明，对于这项检查，*Descend*会执行扩展的 借用（或访问）检查，与 Rust 类似，会追踪哪个线程（形式化为位置表达式）访问了哪个内存位置（形式化为执行资源）。为了使这项检查可行，在 *Descend* 中，程序员通过视图来表达并行内存访问，它们是安全的并行访问模式，例如本例中用于反转的 `rev` 。视图可以组合起来，从而实现复杂但仍可证明安全的并行访问模式。

同步。 由于各个代码块独立运行，只有在 GPU 内核执行完毕时，所有代码块才会同步。不过，由于一个代码块中的线程在同一个流式多处理器上执行，因此可以使用块级屏障将它们同步。为避免未定义行为（包括可能发生的死锁），该代码块中的每个线程都必须到达屏障。遗憾的是，很容易违反这一要求：

```
CUDA
__全局__ 内核(...) { 如果 (线程索引.x < 32) { __同步
() } }
```

在此 CUDA 内核中，每个代码块内只有索引小于 32 的线程才会执行 `__synchronize` 屏障。每个代码块启动的线程数不超过 32 时，每个线程都会按要求到达屏障；但每个代码块启动的线程数超过 32 时，程序的行为未定义。

在 *Descend* 中，如果每个代码块超过 32 个线程，那么像这样的程序将无法编译。等效的*Descend* 程序会报出如下错误信息：

```
Descend
错误：此处不允许使用屏障   |  拆分(X) 代码块位于32{
  |  ^^^^^^^^^^^^^^^^^ `代码块` 在此处拆分 --------
  |   前面的_32_线程 =>{同步}   |
  |                          ^^^^ `同步` 并非   |
由代码块中的所有线程执行 ------
```

我们将在第3节讨论，*Descend* 如何检查同步是否正确执行。在*Descend*中，代码块中的所有线程要么执行相同的指令（使用之前介绍的`sched` 语法时），要么必须使用上述错误信息所示的语法拆分代码块。基本上，禁止在拆分块内嵌套同步操作。*Descend* 还会确保不会遗漏同步操作。同步操作会释放对已同步内存的借用；如果遗漏同步操作，借用检查器就会像上文所示那样将其标记出来。

2.3. 异构性的挑战

编写内核并不是编程 GPU 的唯一方式。GPU 是异构系统的一部分，其中 CPU 和 GPU 会异步执行计算，而 CPU 则负责管理 GPU 上的计算。这会带来两项重大挑战：处理 CPU 和 GPU 上物理分离的分离内存，以及应对 CPU 和 GPU 之间的共享假设；这些假设往往没有被明确编码，可能以微妙的方式破坏正确性。

分离内存。 CPU 和 GPU 是物理上彼此独立的设备。在 CPU 上运行的主机 线程会启动 GPU 内核，并通过应用程序接口与 GPU 传输数据，但无法直接访问 GPU 内存。

而且无法直接访问 GPU 的内存。CPU 程序只能访问 CPU 内存，而 GPU 程序只能访问其各类 GPU 内存。CUDA 通过地址空间 体现分离内存，这些地址空间是对指针的注释。CUDA 编译器并不严格强制执行这些注释，因此程序员很容易犯下编译器无法发现的错误，例如误用用于向 GPU 复制数据的应用程序接口：

```CUDA
cudaMemcpy(d_vec, h_vec, size, cudaMemcpyDeviceToHost);
```

函数 `cudaMemcpy` 会将 size 个字节从第二个参数中的源位置复制到第一个参数中的目标位置。最后一个参数指定目标和源是在设备上还是主机上。在上述调用中，目标指针和源指针被交换了，导致程序使用主机指针中的地址访问设备上的内存，从而产生未定义行为。

在 *Descend* 中，引用类型携带额外信息，并会严格强制正确使用。若犯了与上面相同的错误，编译时会出现错误信息：

```Descend
错误：类型不匹配    |复制_内存_到_主机(d_vec,h_vec);
    |                  ^^^^^ 预期为对    |       `
GPU 全局内存`的引用，实际为对 `CPU 内存`的引用
```

在 CUDA 中，可以将 CPU 指针直接传递给 GPU。这意味着 GPU 程序可能会意外尝试直接访问 CPU 内存，如以下代码所示：

```CUDA
void 主机_fun() {    double*vec =
malloc(sizeof(double) * N * N);    init_内核<<<N, N
>>>(vec); }  __global__ void init_内核(double
*vec) {  vec[globalIdx.x] = 1.0; }
```

在这个示例中，主机在 CPU 主存中为一个数组分配空间，并将生成的指针传递给 GPU。随后，GPU 程序尝试初始化这块内存，但它无法访问独立的主存，因此导致未定义行为。

在*Descend*中，这样的程序无法编译，因为编译器识别到我们正试图在 GPU 上访问 CPU 内存。Descend 中等效的 GPU 程序会像这样报错：

```Descend
错误：无法解引用 `*vec` 指向 `CPU 内存`    | sched(X) 网格中的
线程 {    | ---------------------- 执行者：  `GPU线程'
    |           [[      ]] = 1 0 (*vec)线程。在 `CPU 内存'内存
    |      ^^^^
```

在第3节中，我们介绍了执行资源，用于标识谁执行一段代码，重点关注 GPU。不过，这些概念也扩展到了 CPU 线程。形式类型系统，在第 4 节中介绍，通过 内存注解 扩展引用，严格确保内存仅在正确的执行上下文中被解引用。

CPU与GPU之间的共享假设。在 CUDA 和 *Descend* 中，当在 GPU 上启动函数时，主机线程会指定启动配置，即执行内核的线程数量，以及这些线程如何分组成块。实现 GPU 函数时，通常会隐含一些假设，例如将执行该函数的线程数量，以及通过主机内存API分配的内存量。但这些假设很容易在 CPU 端或 GPU 端被违反，例如下面这个用于缩放向量的 GPU 函数：

```CUDA
__全局__ 缩放_向量_内核(双精度 *向量) {  向量[全局索
引.x] = 向量[全局索引.x] * 3.0; }
```

GPU 上的每个线程都会通过其在整个网格中的索引访问向量中的一个元素。这里的假设是，网格中的线程数与向量中的元素数相同。例如，以下从 CPU 启动该 GPU 函数的方式就是错误的：

```CUDA
cudaMalloc(&d_ptr, SIZE);  ...scale_
vec_内核<<<1, SIZE>>>(d_ptr);
```

函数启动的线程数不是与向量元素数量相同，而是与向量中的字节数相同。由于启动 GPU 函数时使用的线程数多于向量元素数量，因此会触发越界内存访问。

在 *Descend* 中，使用错误数量的线程调用 GPU 程序会在编译时产生错误信息：

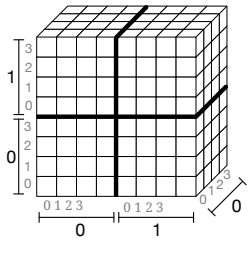
```Descend
错误：类型不匹配    |  scale_vec<<<X<1>,X<SIZE>>>>(d_
vec);    |                              ^^^^^    | 预
期为 `[f64; SIZE]`，实际为 `[f64; ELEMS]`
```

我们将在第 3 节看到，所有函数都会标注执行资源，用于描述函数期望的执行方式。这使得相关假设显式化。第 4 节介绍的类型系统会在编译时强制执行这一要求，并支持使用常量（例如 1024）以及通过多态使用变量的启动配置。

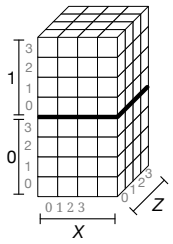
3. 使用*Descend*进行安全的 GPU 编程

本节将讨论*De-scend*用于确保内存安全并生成上一节中所见错误信息的技术机制。我们会先给出相关解释和直观说明，然后在第*Descend*4节中介绍形式化类型系统的几个重要方面。

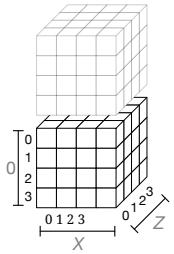
我们首先介绍执行资源、位置表达式和视图，将它们作为形式化推理的核心要素



(a) 由四个块组成的三维网格 每个块包含 $4\times4\times4$ 个线程。



(b) 由两个代码块组成的组，在X和Z维度中调度之后。



(c) 使用拆分选择的单个线程块。

图 1：*Descend*中执行资源分层调度的可视化

以分析执行与内存层次结构，并检查并行内存访问是否安全。

3.1. 执行资源

在*Descend*中，GPU 上的计算会按照层级在由代码块和线程组成的网格上进行调度。在下面的示例代码中，我们使用一个由$2\times2\times1$个代码块组成的 3D GPU 网格执行一个函数，每个代码块包含 $4\times4\times4$ 个线程。该网格的形状如图 1a 所示，并由一个执行资源描述，如第 1 行所示：

```
1fn foo(...) -[grd: gpu.Grid<XYZ<2,2,1>,XYZ<4,4,4>>]-> (){2
sched(X,Z) 块在grd {3    拆分(Y) 块在1 { 4    fstBlock =>...5
sndBlock =>... } } }
```

这里，`grd` 是执行资源，我们通过标注其类型来指定它，该类型描述了网格的形状。下一行由整个网格“执行”。我们将“执行”一词加上引号，因为除单个线程外，执行资源只能分配内存，或调度其嵌套执行资源上的计算。

我们将以下计算调度到网格中所有具有相同 X 和 Z 坐标的块组上。图 1b展示了这样一组块。在代码中，执行资源是 `块`，而在形式上，*Descend* 将其视为 `GPU网格<XYZ<2,2,1>,XZY<4,4,4>>.forall(X).forall(Z)`的别名，后续讨论的安全检查将使用它。可以看到，执行资源会跟踪截至目前网格层级调度的所有信息。

在示例的第 3 行，我们在Y 维度上位置 1 处拆分块，将其分为两个子组。每个子组都由一个独立的执行资源标识：`fstBlock` 和 `sndBlock`，它们可以执行相互独立的计算。图 1c展示 `fstBlock` 并给出了*Descend* 进行推理时所依据的执行资源的形式化表示。

使用 `sched` 和 `拆分` 对网格进行分层调度，直到到达单个线程执行计算。

e	::=	CPU.线程	执行资源
		GPU.网格$\langle d,d\rangle$	
		e.对所有([X | Y | Z])	
		e.拆分(η,[X | Y | Z]).[第一项 | 第二项]	
d	::=	XYZ$\langle\eta,\eta,\eta\rangle$	维度。
		XY$\langle\eta,\eta\rangle$ | XZ$\langle\eta,\eta\rangle$ | YZ$\langle\eta,\eta\rangle$	
		X$\langle\eta\rangle$ | Y$\langle\eta\rangle$ | Z$\langle\eta\rangle$	
η	::=	$0 \mid \ldots \mid 9 \mid n \mid \eta+\eta \mid \eta*\eta \mid \ldots$	自然数

图 2：执行资源与维度

图2展示了执行资源的形式文法。

除了分层 GPU 网格外，我们还用一种执行资源来描述 CPU 线程（`cpu.thread`）。例如，它可用于标记在 CPU 上执行的函数。`GPU网格`存储两个维度d，用于描述代码块的数量以及最多三维的形状。我们添加了额外的一维和二维形式，例如 `XY<η, η>`，以便检查是否会针对缺失的维度进行调度。维度的大小用自然数表示 η，它可以是常量、变量，也可以是由自然数组成的简单数学表达式。

GPU 网格或代码块可以通过之前介绍的 `sched` 语法，将其中所有元素一视同仁地进行调度；在执行资源中，则用 `forall` 表示法指出要针对哪个维度进行调度。或者，我们也可以像上面的示例那样，在某个位置拆分执行资源，将其分成两个执行独立指令的子组。这在执行资源中用 `split` 表示法表示，并且必须立即选择其中一个子组。回看本节开头的示例，`fstBlock`对应于拆分这些块，然后立即选择第一个块（`blocks.split(1, Y).fst`）。

此处介绍的执行资源在 *Descend* 中有三个主要用途：*1)* 用于检查哪些代码在 CPU 和 GPU 上执行；*2)* 用于检查 GPU 层级结构的哪些部分执行了哪些指令，例如，屏障同步必须由哪一部分执行。

p	$::=$		位置表达式：
		x	变量
		$p.fst \mid p.snd$	投影
		$*p$	解引用
		$p[t]$	索引
		$p[\![e]\!]$	选择
		$p.v :: \langle \overline{\eta}, \overline{\delta} \rangle (v)$	视图

图 3：位置表达式

在代码块中执行；*3)* 它们会跟踪代码生成中使用的维度和大小。

3.2. 位置表达式和视图

位置表达式Rust 引入了位置表达式这一概念，用作内存区域的唯一名称。通过替换被引用的位置表达式，可以解析别名。这样，Rust 的类型系统就能在语法上对它们进行比较，以确保不会同时访问同一内存位置（且至少有一次是可变访问）。由此可以保证不会发生数据竞争。

图3 展示了*Descend*中存在的位置表达式。最简单的位置表达式是变量，它为一个内存区域命名。投影。`fst` 和。`snd` 应用于元组，分别指向两个不重叠的内存区域。解引用运算符会访问引用所指向的内存。对于包含同类型数据数组的内存区域，可以通过索引访问其中的单个元素。这些位置表达式在 Rust 中也都存在。

在*Descend*中，我们引入了另外两种位置表达式。选择表达式从数组中为执行资源选择内存。此操作要求执行资源（例如代码块）由与数组元素数量相同的独立子执行资源（例如线程）组成。随后，每个子执行资源访问数组中的一个元素，从而实现安全的并发数组访问。不过，这种访问受到很大限制。目前无法选择多个元素，也无法更改由哪些子执行资源访问哪些元素。为了提高安全并行内存访问的灵活性，我们引入视图。

视图 视图会重塑数组或重新排列数组元素，因此改变我们访问数组的方式。视图应用于指向数组的位置表达式。对位置表达式应用视图会生成新的位置表达式，从而可以链式应用多个视图。原始数组的内存布局保持不变。只有访问数组时的行为会根据视图而改变。生成代码时，视图会被编译为原始索引，过程与 Lift 编译器 [17] 和 DPIA [1]中采用的过程类似。

代码清单 3 展示了基本视图的类型，位于 *Descend*。split 会在指定位置将数组拆分为两个互不重叠的部分数组，并返回一个包含它们的元组。split 的

```
Descend
拆分<k: 自然数, n: 自然数, d: dty>([[d; n]])
  -> ([[d; k]], [[d; n-k]]) 其中n >= k
组<k: 自然数, n: 自然数, d: 数据类型> ([[d; n]])
  -> [[ [[d; k]]; n/k]] 其中n % k == 0
转置<m: 自然数, n: 自然数, d: dty> ([[ [[d; n]]; m]])
  -> [[ [[d; m]]; n]]
<escape>7反向<n: 自然数,</escape>
                d: dty>([[d; n]])-> [[d; n]]
映射<m: 自然数, n: 自然数, d1: 数据类型, d2: dty>(
    ([[d1; m]])-> [[d2; m]], [[ [[d1; m]]; n]] )
  -> [[ [[d2; m]]; n]]
```

清单 3：基本视图及其类型。

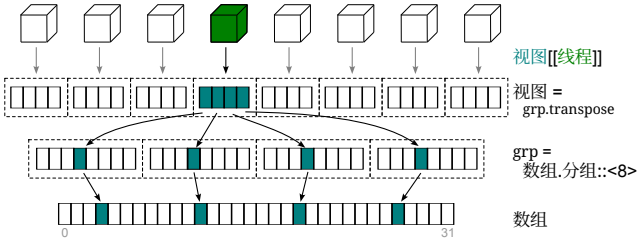


图 4：通过视图，顶部的线程安全地并行访问底部数组中高亮显示的元素。

位置必须在输入数组的大小范围内。此视图使程序员能够选择数组的一部分进行操作，并对另一部分执行其他操作或将其丢弃。分组视图会将输入数组中的连续元素组合成指定大小的嵌套数组。因此，通过对元素进行分组，数组的维度会增加一维。嵌套数组构成外层数组的元素。这样便可以使用选择运算符选择整组元素。转置视图会转置二维数组，而 reverse 会反转元素的顺序。最后，映射会将视图应用于数组的每个元素。

我们将基本视图组合起来，形成更复杂的新视图。例如，清单2中使用的视图就是如此。具体来说，视图 `group_by_row` 定义为 Descend

```
1          _  _   <   _              _          > =
2          <                  >                  视图 按行分组,
行大小: nat, 行数: natgroup::row_size/num_rows.map(transpose)
```

图4展示了顶部的一组线程如何安全地访问底部的数组。完整的位置表达式`array.group::<8>.transpose` 描述了对数组视图进行变换的过程。我们首先将包含 32 个元素的数组分成 4 组，每组 8 个元素，并将得到的二维数组称为 `grp`。接着转置分组后的数组，然后使用选择运算符，通过多个线程并行访问它。此数组访问从构造上就是安全的，因为每个视图都描述了数组的一种安全重塑方式，其结果仅仅是重新映射各线程访问的数组元素。

为了检查位置表达式是否被独占访问，*Descend*与 Rust 一样，会在语法上比较位置表达式之间的差异。例如，我们可以在语法上确定位置表达式 x.split::<32>.fst

以及 `x.split::<32>.snd` 彼此不同（因为它们在同一位置以不重叠的方式拆分），而它们都与表示整个数组的位置表达式 x 重叠。

3.3. *Descend* 中的扩展借用检查

什么是 Rust 的借用检查器？它是如何工作的？ Rust 引入了所有权、借用和生命周期 等概念，以在静态层面保证程序中不存在数据竞争、内存泄漏、释放后使用错误以及其他与内存相关的错误。所有权机制确保内存对象不会出现别名，因为将值赋值给新变量后，只能通过新变量访问该值。任何尝试访问旧变量的行为都会导致编译器错误。

由于这种模型的限制过于严格，因此在该语言中引入了借用机制，并采用了一种受限的别名概念。借用检查器会检查一个线程是否允许创建对某个内存对象的引用，即是否允许“借用”该内存对象。引用分为唯一引用和共享引用。多个共享引用可以同时使用，但只能用于读取，不能用于写入。唯一引用则保证不存在其他可用于访问同一内存位置的引用或变量。因此，可以安全地修改底层内存。

最后，生命周期确保引用所指向的内存仍然存在且尚未被释放。如果尝试在程序中底层内存已被释放的位置进行解引用，则会导致编译器报错。

***Descend*的扩展借用检查器** 在 CPU 上，*De-scend* 实现了与 Rust 完全相同的规则。在 GPU 端，所有权和借用规则得到了扩展，并且与 Rust 有所不同。在 Rust 中，独占所有权始终属于单个线程。在 *Descend* 中，每种执行资源（例如网格或代码块）都可能取得某个内存对象的所有权。类似地，执行资源也可能创建引用，即进行借用。这意味着，块集合或线程集合，以及单个线程，都拥有并借用内存对象，这些对象在形式上由位置表达式表示。函数的参数由该函数的执行资源拥有。为了使单个线程能够通过独占访问某个内存位置来写入其中，必须使用*Descend* 的层次化调度、选择和视图来收窄 所有权和借用关系。

收窄 收窄描述了在沿着执行层次结构从网格到块再到线程的过程中，所有权和借用关系如何逐步细化。例如，网格对一个数组的所有权会通过各个块共同借用该数组而收窄到网格中的各个块，每个块借用数组中一个不同的部分。这一过程还可以进一步收窄到块中的各个线程。

但如下所示，收窄也可能被违反：

```
Descend
1fn kernel(arr: &uniq gpu.global [f32; 1024]) 2 -[grd: gpu.Grid<X<32>,X<32>>]-> () { 3 sched(X) 代码块 in grd { 4 let in_borrow =&uniq *arr; // 违反收窄规则 5 sched(X) 线程 in 代码块 { 6 let grp =&uniq arr.group::<32>[[线程]]; 7 // 违反收窄规则 8 arr.group::<32>[[代码块]][[线程]]; } } }
```

在此示例中，参数 `arr` 由网格拥有。在网格的块已完成调度之后，尝试在第 4 行借用 `arr` 会违反收窄规则，因为网格中的每个块都将获得对整个数组的唯一写入访问权限。

第 6 行显示了另一个收窄违规。此处，数组被分组，组的数量与每个代码块中的线程数相同。然后，每个线程选择一个组，并唯一地借用该组。然而，由于尚未针对该 `代码块` 执行选择，因此选择操作会针对每个代码块执行。因此，来自不同代码块的线程将获得对同一内存位置的访问权限。

第 8 行展示了正确的收窄。数组被分组，每个代码块独占借用数组的一部分，之后每个代码块中的线程都会从中选择一个元素。

同步 *Descend* 的收窄系统确保任意两个线程都无法对同一内存位置进行可变访问。然而，有时我们确实希望通过共享内存与另一个线程通信，此时另一个线程也必须能够访问同一内存位置。因此，我们需要一种方法，允许另一个线程随后进行访问，同时保证该访问不会导致数据竞争。通过在屏障处同步线程，我们可以保证屏障之前的所有内存访问不会与屏障之后的内存访问发生冲突。

在本节中，我们讨论了*Descend* 如何应对第2节中指出的两项挑战： 防止数据竞争并处理同步。我们将在第 *4* 节看到，在类型系统中，*Descend* 会在类型判断的环境中跟踪一个映射，记录哪个执行资源访问了哪些位置表达式。类型规则使用此环境执行访问安全检查，包括正确的收窄。在同步时，我们会从映射中移除代码块内所有线程之前的访问记录。在接下来的章节中，我们将讨论Descend 如何应对管理异构系统的挑战。

3.4. 在*Descend*中处理分离内存

追踪内存空间 *Descend* 为引用类型标注了地址空间。CUDA 也采用类似做法，但 CUDA 没有专门用于 CPU 指针的地址空间，通常也不会严格强制正确使用这些指针。在*Descend*中，所有引用都带有地址空间，包括 cpu.mem 地址空间，它包含存储在 CPU 栈和堆中的值。对于 GPU，我们区分全局内存 gpu.global 和共享内存 gpu.shared，它们拥有各自独立的地址空间，

借助执行资源，*Descend* 会确保引用仅在正确的执行上下文中解引用，例如防止在 CPU 上解引用 GPU 引用。

Descend 还支持内存空间多态：通过引入类型级变量 m，用它代替具体的地址空间。

分配内存 动态内存分配，即在 CPU 堆和 GPU 全局内存中的分配，通过唯一智能指针进行管理，以确保内存能够安全释放且不会发生泄漏。我们将这些值的类型称为 @ 类型，因为它们带有注释 *at*，注明其分配所在的地址空间。智能指针在作用域结束时销毁，内存也随之释放。因此，我们的类型 T @ CPU 内存。对应于 Rust 中的 Box<T>，以及 C++ 中的 std :: unique_ptr<T>。下面的代码展示了如何分配并初始化内存：

```
Descend
1{ let cpu_array: [i32,n] @ CPU 内存 = CpuHeap::new([0;n]);
2 { let global_array: [i32;n] @ GPU 全局内存3 =
GpuGlobal::alloc_copy(&cpu_array); 4 } // 释放 global_array 5 } //
释放堆 _array
```

在外层代码块中，分配堆内存，并使用一个大小为 n、所有元素均初始化为 0 的整数数组进行初始化。随后，将管理该内存分配的智能指针存储在变量 cpu_array 中。在内层代码块中，为 heap_array 所指向的数据分配 GPU 全局内存，将数据复制到 GPU，并将生成的智能指针存储在 global_array 中。此处显示的类型注解是可选的，但它们展示了类型中存储的信息。

3.5. 在*Descend*中明确隐式假设

CPU 程序负责调度 GPU 函数执行。在 *Descend* 中，这通过一种特殊的函数调用来实现，与 CUDA 一样，调用时不仅要提供函数参数，还要指定执行该函数的 GPU 网格；此处的网格由 32 个代码块组成，每个代码块包含 32 个线程：

```
Descend
1缩放_向量::<<<X<32>, X<32>>>>(&唯一 向量);
```

与 CUDA 不同，在 *Descend* 中，GPU 函数签名携带了允许执行该函数的网格配置信息：Descend

```
1函数 缩放_向量(向量: &唯一 gpu.global [i32; 1024]) 2 -[
网格: GPU网格<X<32>, X<32>>]-> ();
```

Descend 会检查调用方与函数声明是否匹配，以确保对函数编写方式和调用方式的假设不会出现偏差。*Descend* 还支持网格大小的多态性，因此可以编写 GPU 函数，例如启动与输入数组大小相同数量的线程。在这种情况下，调用方会指定用于实例化网格大小变量的具体值。

CPU 线程会等待 GPU 函数调用完成，这意味着每次 GPU 计算结束时，GPU 网格都会发生隐式同步。

4. *Descend* 的类型系统

本节介绍 *Descend* 的形式化基础，包括项和类型的形式语法，以及最重要的类型规则，以解释确保安全性的形式化推理。我们的类型系统基于对 Oxide 中 Rust 类型系统的形式化 [18]。Descend 的完整类型系统技术报告将在论文发表时提供。

4.1. 项的语法

图 5展示了 *Descend* 项的形式语法。位置表达式是表示内存访问的项，如第 3.2 节所述。let 绑定用于引入并初始化新变量。赋值会计算等号右侧的项，并将其值存入左侧位置表达式所指向的内存中。引用可以选择性地标注为唯一，以支持通过该引用进行写入。默认情况下，引用是只读的。代码块会引入一个新作用域，其中包含一系列项。函数应用会用静态求值的自然数（η）、内存和数据类型实例化多态函数 f，并以一系列项作为参数调用所得函数。for 循环有两种：遍历集合的 for-each 循环，以及遍历静态求值自然数范围的 for 循环。调度原语接受一个维度，并在某个执行资源内嵌套的子执行资源上调度相同的嵌套计算，例如代码块中的线程。拆分执行原语会沿给定维度，在指定位置将执行资源拆分为两个独立部分。然后，它会指定每个部分在其主体中执行的计算。最后，屏障同步原语会同步代码块中的所有线程。

4.2. 类型语法

图 6 展示了类型和种类的形式语法。Descend 中的类型 *Descend* 由数据类型（δ）和函数类型组成。数据类型包含标准的标量类型和元组类型。数组按其大小索引，该大小以符号形式记录在类型中。我们为经过视图转换的数组引入了一种特殊的数组视图类型。标准数组保证在内存中连续存储，而数组视图类型则不具备这一保证。引用类型的建模方式与 Oxide 类似。`uniq` 限定符表示引用具有唯一性。没有限定符的引用则是共享的，并且只读。我们在原始定义中加入了内存注解 μ，用于记录引用所指向的内存空间。可能的内存地址空间显示在

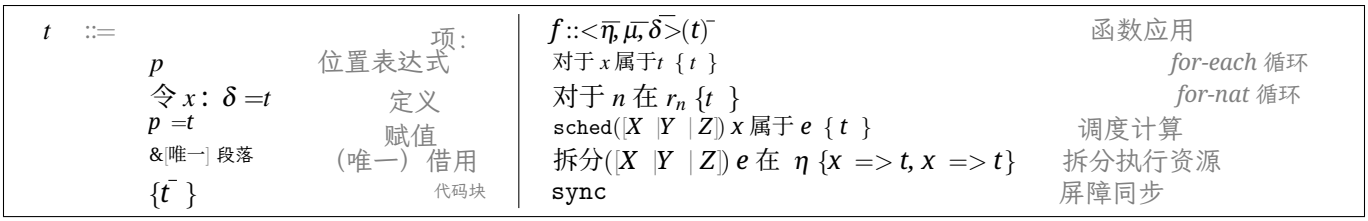


图 5：*Descend* 项的形式语法

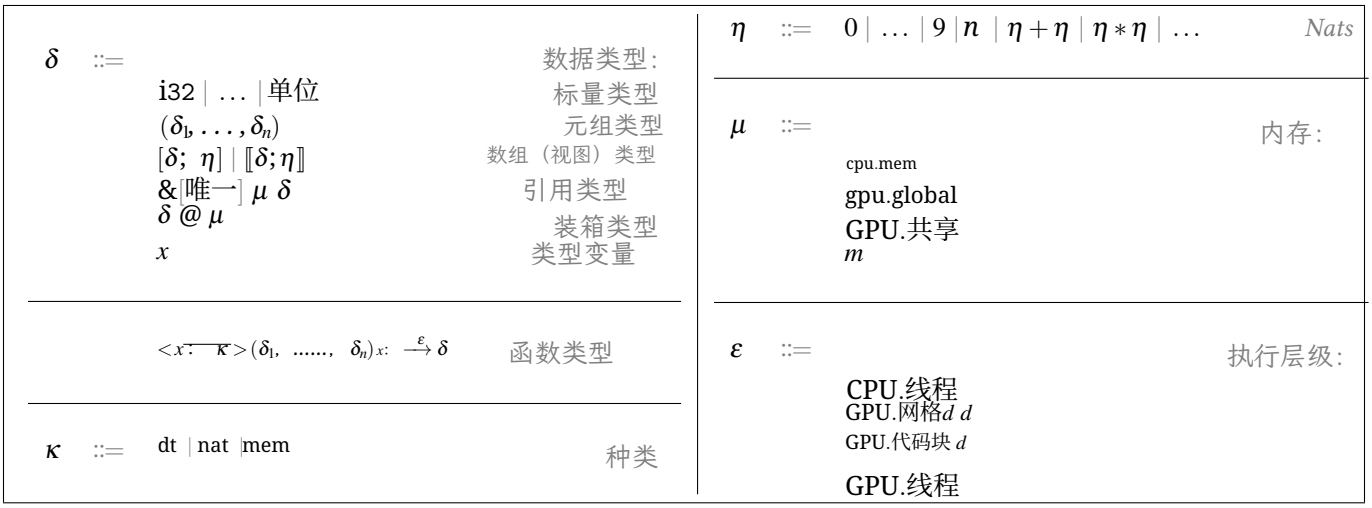


图 6：*Descend* 中种类和类型的形式语法。

图的右侧：`cpu。mem`，`gpu。global`，以及`gpu。shared`。这里省略了每个引用所携带的生命周期变量的介绍。这些变量对于借用检查很重要，但会使表述复杂化。生命周期的处理已在 Oxide [18] 和 FR [15]中形式化。装箱类型会记录其智能分配的值存储在哪个内存空间中。最后，类型变量可以出现在多态函数定义中。

函数类型可以是多态的，包含一个类型级变量列表，且每个变量都标注了其种类（κ）。类型级变量可以取数据类型、自然数（η）或内存空间。我们目前将函数参数和返回类型限制为数据类型，因此不支持 *Descend* 中的高阶函数。虽然在 CPU 上支持这一特性很容易，但要在 GPU 上高效实现高阶函数并不简单，因此我们将其留待未来工作。在函数箭头上方还有一个额外参数，即执行资源，并标注了一个执行级别（ε）。执行级别通过比较调用端执行资源的执行级别与函数标注的执行级别，来决定哪些执行资源可以调用该函数。

4.3. 类型规则

类型判断 *Descend*中的所有安全检查，包括关键的借用检查，都通过类型规则进行了形式化。因此，正式的类型判断相当复杂，需要使用多个以大写希腊字母表示的环境来跟踪各种信息。类型判断会考虑以下信息：类型变量的种类（Δ）、全局可访问函数的类型（Γ_g）、函数内局部变量和活跃借用的类型（Γ_l）、临时借用（Θ）、执行当前函数的执行资源及其级别（ef: ε）、执行当前语句的执行资源（e），以及访问环境A，该环境跟踪哪个执行资源能够访问哪个位置表达式。此外，类型判断具有流敏感性，这意味着类型环境和访问环境会在类型检查过程中发生变化。例如，访问一个自有值后，我们不允许再次访问它（因为它已被移动），因此会将它从类型环境中移除。同样，从一个表达式的类型检查转到下一个表达式时，可访问权限也会发生变化，正如下文所述。

因此，最终的类型判断如下：

$$\Delta; \Gamma_g; \Gamma_l; \Theta \mid e_f : \varepsilon; e \mid \mathrm{A} \vdash \boxed{t : \delta} \dashv \Gamma_l' \mid \mathrm{A}'$$

也就是说，项t 在所述环境下具有类型 δ，并产生更新后的类型环境Γ_l' 和访问环境 A'。

类型规则 我们重点介绍图 7 中的三条重要类型规则。它们更形式化地概述了 *Descend* 最重要特性之一的要求：避免数据竞争。这些规则基于 Oxide，并针对 GPU 的额外需求进行了调整，涵盖了我们的执行资源和扩展的位置表达式语法。

$$\textsc{T-Sched}\ \frac{\Delta;\Gamma_g;\Gamma_l;\Theta \mid e_f:\varepsilon;e.\texttt{forall}(d) \mid \mathrm{A} \vdash \boxed{t:\texttt{unit}} \dashv \Gamma_l' \mid \mathrm{A}'}{\Delta;\Gamma_g;\Gamma_l;\Theta \mid e_f:\varepsilon;e \mid \mathrm{A} \vdash \boxed{\texttt{sched}(d)\,x\,\texttt{in}\,e\,\{\,t\,\}:\texttt{unit}} \dashv \Gamma_l' \mid \mathrm{A}'}$$

$$\textsc{T-Read-By-Copy}\ \frac{\begin{array}{c}\{\ ^{\texttt{shrd}}p_i\,\} = \texttt{access\_safety\_check}(p,\texttt{shrd},\ \Delta;\Gamma_l;\Theta,e_f:\varepsilon;e,\mathrm{A}) \\ \Delta;\Gamma_l \mid e_f:\varepsilon;e \mid \texttt{shrd} \vdash_{\texttt{pl}} p:\delta \qquad \texttt{is\_copyable}(\delta)\end{array}}{\Delta;\Gamma_g;\Gamma_l;\Theta \mid e_f:\varepsilon;e \mid \mathrm{A} \vdash \boxed{p:\delta} \dashv \Gamma_l \mid \mathrm{A}, e \to \mathrm{A}(e) \cup \{\ ^{\texttt{shrd}}p_i\,\}}$$

$$\textsc{T-Write}\ \frac{\begin{array}{c}\{\ ^{\texttt{uniq}}p\,\} = \texttt{access\_safety\_check}(p,\texttt{uniq},\ \Delta;\Gamma_l';\Theta,e_f:\varepsilon;e,\mathrm{A}') \\ \Delta;\Gamma_l' \mid e_f:\varepsilon;e \mid \texttt{uniq} \vdash_{\texttt{pl}} p:\delta \qquad \Delta;\Gamma_g;\Gamma_l;\Theta \mid e_f:\varepsilon;e \mid \mathrm{A} \vdash \boxed{t:\delta} \dashv \Gamma_l' \mid \mathrm{A}'\end{array}}{\Delta;\Gamma_g;\Gamma_l;\Theta \mid e_f:\varepsilon;e \mid \mathrm{A} \vdash \boxed{p = t:\texttt{unit}} \dashv \Gamma_l' \mid \mathrm{A}', e \to \mathrm{A}'(e) \cup \{\ ^{\texttt{uniq}}p_i\,\}}$$

图 7：*Descend* 中用于读取和写入内存、执行高级访问安全检查，以及在执行层次结构上调度计算的重要类型规则。

T-Sched 规则展示了在使用 sched 遍历执行层次结构时，如何跟踪执行资源。sched 的主体使用当前执行资源 e 进行类型检查，并通过 forall 扩展该资源，加入相应维度，以表示该主体由 e 中的所有执行资源执行。

T-Read-By-Copy 规则检查位置表达式 p，该表达式用于从内存中读取可复制的值；与之相对的是，根据 Rust 所有权规则而被移动的值。该规则指出，如果横线以上的前提成立，那么 p 具有良好类型，且数据类型为 δ。为执行此检查，另一个位置表达式类型判断会检查 p 的结构，以确定 p 是否具有类型 δ。此判断还需要知道 p 是以共享还是唯一的方式使用，也就是说，它是被读取还是写入，或者是以共享或唯一方式借用。这样才能确保解引用（例如唯一指针的解引用）是允许的（如果 p 以共享方式使用，则不允许解引用）。函数_copyable 会检查数据类型是否可复制而非可移动。最后，access_safety_check 会执行最关键的 GPU 特定检查，包含以下三个逻辑步骤：
1. 收窄检查：检查多个执行资源是否以唯一方式使用位置表达式。此检查确保每个执行资源都从中选择各自不同的部分。 2. 访问冲突检查：检查在某个执行资源中使用位置表达式，是否与访问映射环境中记录的其他执行资源先前的访问相冲突。 3. 借用检查：执行与 Rust 中相同、并在 Oxide 中形式化的借用检查。

对于此规则，如果 p 在给定环境中可被共享访问，则检查成功。该函数会计算出 p_i 的所有可能别名的集合，并将它们全部标记为共享。该规则会在结论右侧生成更新后的访问环境。我们记录当前执行资源 e 以共享方式访问位置表达式 p_i，方法是将映射添加到 A。

与前一条规则类似，T-Write 检查将项t 赋值给位置表达式p 是否安全，并生成新的类型环境和访问映射环境。项t 和位置表达式p 分别独立进行类型检查，且必须具有相同的类型 δ。与前一条类型规则不同，位置表达式的类型判断和访问安全性检查均使用 uniq 限定符，以检查对内存的写入是否安全。类型检查完成后，访问环境 A 中的映射将更新，以反映p 及其别名所引用的内存已被唯一访问。

一份详细介绍 *De-scend* 整个类型系统的技术报告将在发表时提供。

5. 代码生成与评估

本节简要介绍*Descend*的代码生成实现，并将*Descend* 代码与手写 CUDA 代码进行比较评估。我们将展示，如何在不牺牲性能的前提下，将使用*Descend*整体式编程模型编写的程序转换为使用内核编程模型的 CUDA 程序。

代码生成 *Descend* 编译器将*De-scend* 代码转换为 CUDA C++ 代码。面向 CPU 线程的*Descend* 函数会被转换为 C++ 函数。在 GPU 上运行的函数则会被转换为 CUDA 内核。在生成代码之前，我们会内联执行资源并非 GPU 上的完整网格或 CPU 上的线程的函数调用，例如由 GPU 块执行的函数调用。

在 CUDA 内核编程模型中，所有块和线程都并发工作。这正是*Descend* 中的嵌套调度原语所表达的。因此，sched 不会出现在生成的 CUDA 代码中，除非是用于

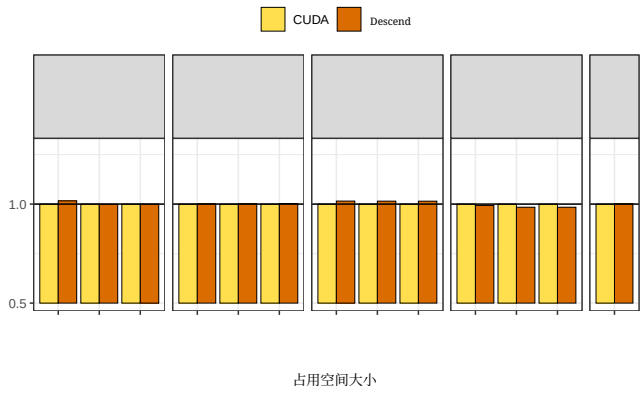


图 8：手写 CUDA 与 De- scend 实现的相对运行时间。柱形越高，表示性能越好。

其主体的作用域；绑定的执行资源变量会编译为等效的索引，用于标识 CUDA 中的线程或代码块。将位置表达式上的选择转换为原始内存索引时，会用到代码块和线程索引。从视图中选择或进行索引时，这些索引会经过转换，以表达这些视图所描述的访问模式。此过程按相反顺序执行，从最后应用的视图开始。每个视图都会接收前一个索引并对其进行转换，直到最终索引表达出所有视图的组合。其余*Descend* 语法会直接转换，同时舍弃 CUDA C++ 中不需要的静态信息，例如引用类型上的内存注解。

实验设置 我们使用搭载 Debian GNU/Linux 10 和 CUDA 11.6 的 Google Cloud 实例，在 Tesla P100 GPU 上进行了实验评估。我们比较了四种通常为 GPU 实现的不同基准测试：块级并行归约、矩阵转置、扫描和矩阵乘法。每种算法都使用*Descend* 实现，并由此生成 CUDA 代码。为进行比较，我们使用 CUDA 手写实现了这些算法，并采用相同的优化方法和访问模式。所有实验均使用三种不同的内存容量：小、中和大，分别使用 256MB、512, MB 和 1GB 的 GPU 内存。我们对每项基准测试运行了 100 次，并测量内核运行时间。扫描基准测试使用两个不同的内核，我们测量从第一个内核开始到第二个内核结束的运行时间。

实验结果图8 展示了*Descend* 相较于手写 CUDA 代码的中位运行时间比值。结果表明，*Descend* 与 CUDA 在所有基准测试和规模下的性能不相上下，性能差异小于 3%。我们看到，*Descend* 具有足够的表达能力，能够编写出性能与手写 CUDA 实现相当的程序，同时提供强大的安全性保证，并能像第 2 节所示那样捕获错误。

6. 相关工作

不安全的 GPU 编程系统 CUDA [13] 很可能是最流行的 GPU 编程语言。OpenCL [5] 以及较新的 SYCL [6] 都是厂商无关的语言，设计非常相似。人们为 C/C 以外的语言编写了许多语言绑定++，例如 Python 中的 PyCUDA/PyOpenCL [11]，但它们通常原样暴露 CUDA 编程模型。[9] 将 Rust 扩展为能够表达 GPU 程序并将其编译为 PTX 代码。GPU 程序支持共享指针和唯一指针，但仍采用传统的 CUDA 内核编程模型实现，因此保留了第2节中指出的所有问题。

安全的 GPU 编程系统 有一类数组语言旨在遵循函数式思想，为高性能 GPU 编程提供安全抽象，包括 Futhark [7], Lift [17] 及其精神上的继承者 Rise [16], 和 Accelerate [12]。在这些语言中，程序天生安全。它们运用 映射 和 归约 等函数式模式，以较高层次描述计算，并据此生成底层 GPU 代码。然而，这些高级抽象是以失去控制权为代价的。*Descend* 旨在让程序员在安全保障下掌握控制权，正如 Rust 对 CPU 所承诺的那样。

GPU 验证工具 *De-scend* 的主要目标之一是避免数据竞争。此前已有针对 GPU 的静态数据竞争检测工具，例如 GPUVerify [2] 和 Faial [3]。这些工具会分析 CUDA C 代码，试图检测数据竞争。Faial 会记录内存访问历史，并区分读访问和写访问。这种历史记录与*De-scend*类型系统中的访问映射环境在理念上相似。当然，经过分析的代码仍可能包含本文提到的其他一些问题，而*Descend* 能够在静态阶段防止这些问题。

Rust 的形式化 Rust 的所有权、借用和生命周期已在 FR Langauge [15]和 Oxide [18] 中完成形式化。FR 专注于借用和生命周期的核心思想，以及它们在当前 Rust 版本中的实现方式，同时保持语言尽可能简单，并支持可能的扩展。Rust 的一些实用特性未在 FR 中建模。Oxide 是我们工作的重要基础之一，专注于当前正在开发的新版本借用检查器的借用规则。此外，它还对 Rust 的其他语言特性进行了形式化，例如多态函数和切片；切片可被视为*Descend*视图的早期灵感来源，而循环则是大多数实际应用所必需的。与 Rust 一样，这些语言无法面向 GPU。

7. 结论

GPU 编程以难度高而闻名，但借助 *De-scend*，我们证明了在保证内存

安全的同时，可以达到与 CUDA 代码相同的性能，并在编译时拒绝存在数据竞争和同步错误的程序。*Descend* 还帮助程序员管理 CPU 和 GPU 内存，并明确规定此前 GPU 代码并行执行中的隐式假设。*Descend* 通过引入执行资源和视图扩展了 Rust 的形式化类型系统，以管理 GPU 执行层级并确保并行内存访问安全。

引用

[1]Robert Atkey、Michel Steuwer、Sam Lindley 和 Christophe Dubach。面向并行函数式代码的保策略编译。*CoRR*，abs/1710.08332，2017年。

[2] Adam Betts、Nathan Chong、Alastair F. Donaldson、Shaz Qadeer 和 Paul Thomson。GPUVerify：一种 GPU 内核验证器。载于 Gary T. Leavens 和 Matthew B. Dwyer 编，第*27*届 *ACM SIGPLAN* 面向对象编程、系统、语言与应用年会论文集，*OOPSLA 2012*，*SPLASH 2012* 的组成部分，美国亚利桑那州图森，*2012*年*10*月*21*日至*25*日，第113–132页。ACM，2012年。

[3] Tiago Cogumbreiro、Julien Lange、Dennis Lie w Zhen Rong 和 Hannah Zicarelli。组合式检查 GPU 内核的数据竞争自由性。载于 Alexandra Silva 和 K. Rustan M. Leino 编，计算机辅助验证——第*33*届国际会议，*CAV 2021*，线上活动，*2021*年*7*月*20*日至*23*日，论文集，第一部分，《计算机科学讲义》第12759卷，第403–426页。施普林格，2021年。

[4] Michael Garland 和 David Blair Kirk。理解面向吞吐量的架构。 *Commun.ACM*，53(11):58–66，2010。

[5] Khronos OpenCL 工作组。OpenCL C 规范，2023年。3.0.14 版。

[6] Khronos SYCL 工作组。SYCL 2020 规范，2023年。第7次修订。

[7] Troels Henriksen、Niels G. W. Serup、Martin Elsman、Fritz Henglein 和 Cosmin E. Oancea。Futhark：一种支持嵌套并行和原地数组更新的纯函数式 GPU 编程语言。载于 Albert Cohen 和 Mar- tin T. Vechev 编，第*38*届 *ACM SIGPLAN* 编程语言设计与实现会议论文集（*PLDI 2017*），西班牙巴塞罗那，*2017*年*6*月*18*日至*23*日，第556–571页。ACM，2017年。

[8] Pieter Hijma、Stijn Heldens、Alessio Sclocco、Ben van Werkhoven 和 Henri E. Bal。GPU 编程的优化技术。*ACMComput. Surv.*，55(11)，2023年3月。

[9] Eric Holk、Milinda Pathirage、Arun Chauhan、Andrew Lumsdaine 和 Nicholas D. Matsakis。使用 Rust 进行 GPU 编程：在系统级语言中实现高级抽象。载于*2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, Cambridge, MA, USA, May 20-24, 2013*，第315–324页。IEEE，2013年。

[10] Ralf Jung、Jacques-Henri Jourdan、Robbert Krebbers 和 Derek Dreyer。《Rust 中的安全系统编程》。美国计算机学会通讯，64(4):144–152，2021 年。

[11] Andreas Klöckner、Nicolas Pinto、Yunsup Lee、Bryan Catanzaro、Paul Ivanov 和 Ahmed Fasih。PyCUDA 和 PyOpenCL：基于脚本的 GPU 运行时代码生成方法。并行计算。，38(3):157–174，2012 年。

[12] Trevor L. McDonell、Manuel M. T. Chakravarty、Gabriele Keller 和 Ben Lippmeier。优化纯函数式 GPU 程序。载于 Greg Morrisett 和 Tarmo Uustalu（编），*ACM SIGPLAN* 函数式编程国际会议，*ICFP'13*，美国马萨诸塞州波士顿——*2013* 年 *9* 月 *25—27* 日，第 49–60 页。ACM，2013 年。

[13] John Nickolls、Ian Buck、Michael Garland 和 Kevin Skadron。使用 CUDA 进行可扩展并行编程。*ACM Queue*，6(2):40–53，2008年。

[14] NVIDIA。Thrust，CUDA C++ 模板库，2023年。版本 2.1.0。

[15] David J. Pearce。Rust 中引用生命周期与借用的轻量级形式化方法。*ACM Trans.Program.Lang.Syst.*，43(1):3:1–3:73，2021年。

[16] Michel Steuwer、Thomas Koehler、Bastian Köpcke 和 FedericoPizzuti。RISE& Shine：面向语言的编译器设计。*CoRR*，abs/2201.03611，2022年。

[17] Michel Steuwer、Toomas Remmelg 和 Christophe Dubach。Lift：用于高性能 GPU 代码生成的函数式数据并行 IR。载于 Vijay Janapa Reddi、Aaron Smith 和 Lingjia Tang 编，*2017* 年国际代码生成与优化研讨会论文集（*CGO 2017*），美国德克萨斯州奥斯汀，*2017* 年 *2* 月 *4—8* 日，第 74–85 页。ACM，2017年。

[18] Aaron Weiss、Daniel Patterson、Nicholas D. Matsakis 和 Amal Ahmed。Oxide：Rust 的本质。*CoRR*，abs/1903.00982，2019 年。