Descend: A Safe GPU Systems Programming Language

BASTIAN KÖPCKE, University of Münster, Germany
SERGEI GORLATCH, University of Münster, Germany
MICHEL STEUWER, Technische Universität Berlin, Germany

Graphics Processing Units (GPU) offer tremendous computational power by following a throughput oriented paradigm where many thousand computational units operate in parallel. Programming such massively parallel hardware is challenging. Programmers must correctly and efficiently coordinate thousands of threads and their accesses to various shared memory spaces. Existing mainstream GPU programming languages, such as CUDA and OpenCL, are based on C/C++ inheriting their fundamentally unsafe ways to access memory via raw pointers. This facilitates easy to make, but hard to detect bugs, such as *data races* and *deadlocks*.

In this paper, we present *Descend*: a safe GPU programming language. In contrast to prior safe high-level GPU programming approaches, *Descend* is an imperative GPU *systems programming language* in the spirit of Rust, enforcing safe CPU and GPU memory management in the type system by tracking *Ownership* and *Lifetimes*. *Descend* introduces a new *holistic GPU programming* model where computations are hierarchically scheduled over the GPU's *execution resources*: grid, blocks, warps, and threads. *Descend*'s extended *Borrow* checking ensures that execution resources safely access memory regions without data races. For this, we introduced *views* describing safe parallel access patterns of memory regions, as well as *atomic* variables. For memory accesses that can't be checked by our type system, users can annotate limited code sections as `unsafe`.

We discuss the memory safety guarantees offered by *Descend* and evaluate our implementation using multiple benchmarks, demonstrating that *Descend* is capable of expressing real-world GPU programs showing competitive performance compared to manually written CUDA programs lacking *Descend*'s safety guarantees.

CCS Concepts: • **Software and its engineering → Parallel programming languages**; *Imperative languages*; • **Theory of computation** → *Logic and verification.*

Additional Key Words and Phrases: GPU programming, language design, memory safety, type systems

ACM Reference Format:
Bastian Köpcke, Sergei Gorlatch and Michel Steuwer. 2024. *Descend*: A Safe GPU Systems Programming Language. *Proc. ACM Program. Lang.* 8, PLDI, Article 181 (June 2024), 16 pages. https://doi.org/10.1145/3656411

1 Introduction

Graphics Processing Units (GPUs) are massively parallel hardware devices with a throughput oriented design that prioritizes the runtime of the overall computation performed in parallel by thousands of collaborating threads over single thread performance, as classical CPUs do [Garland and Kirk 2010]. This has made GPUs attractive devices in many domains where high performance is crucial, such as in scientific simulations, medical imaging, and most prominently, machine learning.

Writing correct and efficient software for GPUs is challenging even for advanced programmers. The predominant languages for general purpose GPU programming, CUDA and OpenCL, are low-level imperative *systems programming languages*, giving programmers great control to precisely

Authors' Contact Information: Bastian Köpcke, University of Münster, Münster, Germany, bastian.koepcke@uni-muenster.de; Sergei Gorlatch, University of Münster, Münster, Germany, gorlatch@uni-muenster.de; Michel Steuwer, Technische Universität Berlin, Berlin, Germany, michel.steuwer@tu-berlin.de.

ACM 2475-1421/2024/6-ART181
https://doi.org/10.1145/3656411

```cuda
__global__ void transpose(const double *input, double *output) {
    __shared__ float tmp[1024];
    for (int j = 0; j < 32; j += 8) {
        tmp[threadIdx.y+j *32+threadIdx.x] =
            input[(blockIdx.y*32+threadIdx.y+j)*2048 + blockIdx.x*32+threadIdx.x]; }
    __syncthreads();
    for (int j = 0; j < 32; j += 8) {
        output[(blockIdx.x*32+threadIdx.y+j)*2048 + blockIdx.y*32+threadIdx.x] =
            tmp[threadIdx.x*32+threadIdx.y+j]; } }
```

Listing 1. A CUDA matrix transposition GPU kernel. A subtle indexing bug in line 4 leads to a data race.

influence how each thread accesses memory and when it performs which computational instructions. This control is needed to extract the expected high performance from GPUs, where the difference between an unoptimized naive implementation and a fully optimized implementation can be up to two orders of magnitude [Hijma et al. 2023]— often significantly more than on CPUs.

Unfortunately, in CUDA and OpenCL, this level of control comes with significant challenges for GPU programmers. As both languages are based on C/C++ they inherit their fundamentally unsafe ways to access memory via raw pointers. Furthermore, to coordinate threads and ensure a consistent view of the memory, manual synchronization primitives must be used correctly. This leads to easy-to-make, but often hard to detect bugs, particularly race conditions when accessing the same memory from multiple threads and deadlocks when using synchronization incorrectly.

Listing 1 shows a CUDA kernel function, executed in parallel on the GPU to transpose a matrix. In lines 4–5, each thread copies four matrix elements into a buffer and then—after a synchronization —copies the transposed elements to the output. The correctness of this function depends on correct indexing which is notoriously tricky. In fact, Listing 1 contains a subtle bug: In line 4, `threadIdx.y+j` should be enclosed by parenthesis, so that both terms are multiplied by 32. As a result, a data race occurs as multiple threads will write uncoordinated into the same memory location.

Rust has demonstrated that a systems programming language can be designed in a memory safe way without losing low-level control. It prevents data races, by forbidding the concurrent access of threads to a memory resource if at least one thread is allowed to mutate it [Jung et al. 2021]. Rust enforces this with its type system, specifically with *borrow* checking, that interacts with the concepts of *ownership* and *lifetimes* which primarily ensure safe memory management. Could Rust have prevented the bug in Listing 1? Clearly, `tmp` is shared among the parallel executing threads and, clearly, we mutate its content in line 5. Therefore, Rust would reject this kernel, without even attempting to investigate if the indexing is safe, as Rust's type system has no capabilities of reasoning about safely accessing an array in parallel by multiple threads.

In this paper, we introduce *Descend*, a safe GPU programming language adapting and extending the ideas of Rust towards GPU systems. In contrast to prior safe GPU programming approaches, such as Futhark [Henriksen et al. 2017], *Descend* is an imperative systems programming language empowering programmers to exercise low-level control with a safety net.

Listing 2 shows the matrix transposition function in *Descend*. In contrast to CUDA, this function is not implicitly executed by thousands of GPU threads, instead this function is executed by the (*one*) GPU grid. Programmers describe the hierarchical scheduling of the computation over the *grid*, first by describing the scheduling of *blocks* (line 4) and then the nested *threads* (line 5). Each `block` has access to its own shared memory that is passed into the function in line 2. Each `thread` performs the same copies as in CUDA, first from the input into the temporary buffer, and then—after

Descend

```
1  fn transpose(input:  &gpu.global [[f64;2048];2048], output: &uniq gpu.global [[f64;2048];2048],
2               [grid.blocks.forall(X).forall(Y)] tmp: &uniq gpu.shared [[f64;32];32]
3  ) -[grid: gpu.grid<XY<64,64>,XY<32,8>>]-> () {
4    sched(Y,X) block in grid {
5      sched(Y,X) thread in block {
6        for i in 0..4 {
7          tmp.group_by_row::<32,4>[[thread]][i] =
8            input.group_by_tile::<32,32>.transpose[[block]].group_by_row::<32,4>[[thread]][i] };
9        sync(block);
10       for i in 0..4 {
11         output.group_by_tile::<32,32>[[block]].group_by_row::<32,4>[[thread]][i] =
12           tmp.group_by_row::<32,4>[[thread]][i] } } } }
```

Listing 2. A *Descend* function performing a memory safe matrix transposition.

a synchronization— back into the output. Instead of raw indexing, in *Descend* programmers use memory *views* to describe parallel accesses into memory. *Descend* statically ensures that accesses into views are safe, and treats them specially in the type system. This restricts memory accesses to safe parallel access patterns. Compositions of views allow for describing complex memory accesses. For the example, the borrow checking of *Descend* is capable to statically determine that the parallel write access into the shared temporary buffer and the output are safe. Similarly, *Descend* statically enforces the correct use of the synchronization, that cannot be forgotten or placed incorrectly.

Descend is a holistic programming language for heterogeneous systems comprised of CPU and GPU. The physically separated memories of CPU and GPU are reflected in the types of references for which *Descend* enforces that they are only dereferenced in the correct execution context. Functions are annotated with an *execution resource* (as seen in the function signature in line 3) indicating how a function is executed. These annotations make important assumptions, that are implicit in CUDA, about how many threads and blocks execute a kernel, explicit and enforceable by the type system.

With *Descend*, we explore one new point in the design space of GPU programming languages, aiming to uniquely combine the imperative nature and low-level control with the safety so-far only found in higher level GPU programming approaches. As we will see in this paper, *Descend* empowers programmers to safely control low-level details including memory layout and which thread accesses which memory location when. This allows expressing GPU algorithms safely, that cannot be described natively by the user and are built-in in existing safe GPU approaches.

In summary, this paper makes the following contributions:

- we introduce *Descend*, a safe GPU systems programming language in the spirit of Rust;
- we identify the challenges of GPU programming and discuss how *Descend* assists in addressing them (Section 2);
- we discuss how the concepts of *execution resources*, *place expressions*, and memory *views* provide the bases for enforcing memory safety, and how *atomics* and *unsafe* enable escaping the restrictive safe *Descend* code when required (Section 3);
- we present *Descend*'s formal type system and extended borrow checking (Section 4);
- and show in an experimental evaluation that programs written in *Descend* achieve the same performance as equivalent programs written in CUDA, that lack *Descend*'s safety guarantees (Section 5).

We discuss related work and conclude in sections 6 and 7.

2 Challenges of GPU Programming

GPU programming brings a number of challenges, that we group in two areas: *1)* challenges from working with the execution and memory hierarchies of GPUs, such as thousands of threads grouped in blocks accessing various GPU memories; and *2)* challenges from coordinating the heterogeneous system, such as transferring data between CPU and GPU memory. Before we discuss each area, we give a brief overview of the traditional GPU programming model established by CUDA.

CUDA is a sophisticated programming language with many advanced features that have been added over years to keep pace with evolving hardware capabilities. In this paper, we focus on a subset of the CUDA features to capture the core essence of CUDA and model it in *Descend*. Therefore, we focus our discussion in this section on these core features and point out significant advanced features of CUDA, that we hope we will be able to model with *Descend* in the future.

2.1 The CUDA GPU Programming Model

In CUDA, programmers write *kernel* functions that are executed in parallel on the GPU. These functions are executed by many thousands of threads, all executing the same code. Therefore, on a first view, the CUDA programming model resembles traditional data-parallel programming models, where a single instruction is applied to multiple data elements in lock-step. However, in CUDA this strict requirement is relaxed as code can branch based on the *thread index* that identifies the individual thread. It is usually used for indexing into arrays so that each thread processes a different array element. Thread indices are integers used to index plain C-style arrays, making statically checking the safety of parallel memory accesses challenging and leading to data races being introduced easily. Furthermore, kernels are often written with implicit assumptions about how many threads execute them, making kernels hard to understand without knowing these assumptions which are an additional source of bugs, when CPU and GPU code diverge over time.

GPUs are comprised of multiple Streaming Multiprocessors (SM), each capable of executing groupings of 32 parallel *threads*, called *warps*, simultaneously. Threads are hierarchically organized into groups, that are executed independently by the SMs. In CUDA, such groups of threads are called *blocks*. The collection of all blocks is called the *grid*.

Similarly, memory is organized hierarchically as well and closely connected to the execution hierarchy. In software, separate *address spaces* reflect the different kinds of GPU memory. The slowest and largest memory is *global memory*, which is accessible by each thread in the entire grid. Each block provides the fast *shared memory* which is accessible only by each thread in the block. Lastly, each thread has exclusive access to its own and fastest *private memory*. Data transferred from the host to the GPU is always stored in global memory. In order to exploit the faster memories, data has to be copied explicitly between address spaces.

2.2 Challenges of the Execution & Memory Hierarchies

The CUDA programming model with its execution and memory hierarchies, resembles closely the GPU hardware and enables scalability of GPU programs, but it comes with two major challenges: how to avoid *data races* and how to correctly *synchronize* the threads of a block.

Data Races. Data races occur when multiple threads simultaneously access the same memory location and at least one performs a write. It is easy to create a data race in CUDA:

CUDA

```
__global__ void rev_per_block(double *array) {
  double *block_part = &array[blockIdx.x * blockDim.x];
  block_part[threadIdx.x] = block_part[blockDim.x-1 - threadIdx.x]; }
```

In this example, that is inspired by a bug in a real-world kernel described by [Wu et al. 2020], the input `array` is split into independent parts for each block. Then the threads in each block access a single element in the reverse order of their thread index and write the value back into the array at their thread index. This creates a data race: a thread may still be reading a value from an index that another thread is already writing to. In *Descend*, the compiler recognizes the possibility of a data race and would reject the program with an error message:

Descend

```
error: conflicting memory access
  | arr[[thread]] = arr.rev[[thread]];
  |                 ^^^^^^^^^^^^^^^^^-------------------
  | ^^^^^^^^^^^^^ cannot select memory because of a conflicting prior selection here
```

We will explain in Section 3, that for this check *Descend* performs an extended *borrow (or access) checking* similar to Rust, tracing which memory location (formalized as *place expressions*) is accessed by which thread (formalized as *execution resources*). To make this check feasible, in *Descend* parallel memory accesses are performed via *views*, which are safe parallel access patterns, such as `rev` for reverse in this example. Views can be composed to enable complex parallel access patterns that are still known to be safe.

Synchronization. To avoid data races, CUDA provides various forms of barriers. Originally, CUDA provided only barriers for synchronizing threads within the same warp or block. We will focus on modelling these barriers in this paper. More recent versions of CUDA have added support for grid-wide and partial barriers and the underlying GPU supports even more sophisticated barriers, such as named and producer-consumer barriers [Bauer, Treichler, and Aiken 2014]. To avoid undefined behavior, when using a barrier, each participating thread must reach that barrier. Unfortunately, it is easy to violate this requirement, such as for this block-wide barrier:

CUDA

```
__global__ kernel(...) { if (threadIdx.x < 32) { __syncthreads() } }
```

In this CUDA kernel, the `__syncthreads` barrier is executed only by threads that have an index smaller than 32 within each block. When launched with more than 32 threads per block, the behavior of the program is undefined. In *Descend*, a program such as this would not compile, if there are more than 32 threads per block, failing with an error message:

Descend

```
error: barrier not allowed here
  | first_32_threads => { sync }
  |                      ^^^^ `sync` not performed by all threads in the block ------
  |    split(X) block at 32 {
  |    ^^^^^^^^^^^^^^^^^^^ `block` is split here
```

We will discuss in Section 3, how the compiler checks that synchronizations are performed correctly. In *Descend*, either all threads in a block perform the same instructions (when using the `sched` syntax seen before), or the threads in a block must be split using the syntax shown in the error message above. A synchronization performed on a split block or a conditional is forbidden. *Descend* also ensures that synchronizations are not forgotten. A synchronizations releases borrows of the synchronized memories which, if forgotten, are flagged by the borrow checker as seen above.

2.3 Challenges of Heterogeneity

GPUs are not programmed only by writing kernels. They are part of a heterogeneous system, with the CPU and GPU performing computations asynchronously and the CPU managing the computations on the GPU. Two significant challenges arise from this: the handling of the physically *separated memories* on CPU and GPU, and dealing with *shared assumptions* between CPU and GPU that are often not explicitly encoded and can break correctness in subtle ways.

Separated Memories. The CPU and GPU are physically distinct devices with separate memories. Data can either be transferred explicitly or implicitly between these memories. To enable implicit data transfers, CUDA 6 introduced CUDA Unified Memory that unifies the virtual address spaces of CPU and GPU pointers. Multiple studies have investigated the performance of CUDA Unified Memory [Knap and Czarnul 2019; Li et al. 2015; Landaverde et al. 2014] and found that it can result in performance overheads (and sometimes performance benefits) depending on the application's memory access patterns. Controlling data transfers explicitly remains an important and widely used method until today. We are focusing on modelling explicit memory transfers in this paper, adding implicit memory transfers to *Descend* is left for future work.

When transferring data between the CPU and GPU explicitly, a *host* thread that is running on the CPU uses an API to initiate the memory transfer. The host program only accesses CPU memory, while a GPU program only accesses its various GPU memories. Programmers are responsible for keeping track of which pointers point into the CPU or GPU memory. This makes it easy for programmers to make mistakes that are not caught by the compiler, such as misusing the provided API for copying data to the GPU:

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

Function `cudaMemcpy` copies `size` many bytes to the destination in the first argument from the source in the second argument. The last argument specifies whether the destination and source are on the device or host. In the above call, destination and source pointers are swapped, which leads to the address in the host pointer being used to access memory on the device, with undefined behavior. In *Descend*, reference types carry additional information and correct usage is strictly enforced. Making the same mistake as above, leads to a compile-time error message:

Descend
```
error: mismatched types
  | copy_mem_to_host(d_vec, h_vec);
  |                  ^^^^^ expected reference to `gpu.global`, found reference to `cpu.mem`
```

In CUDA, without using Unified Memory, special allocation APIs, or a CUDA-aware allocator in the operating system, it is possible to allocate memory in CPU memory and pass the CPU pointer to the GPU. The GPU kernel may then accidentally attempt to access CPU memory directly, as in the following code:

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

In this example, the host allocates space for an array in the CPU main memory and passes the resulting pointer to the GPU. The GPU program then attempts to initialize the memory, but it has no access to the separated main memory, leading to undefined behavior. With full support of CUDA Unified Memory, this program is well-defined, as the GPU would directly access the CPU main

memory, resulting in an implicit memory transfer. In the initial version of *Descend*, a program such as this would be rejected by the compiler because it recognizes that we are attempting to access CPU memory on the GPU. The equivalent Descend program fails like this:

```
Descend
error: cannot dereference `*vec` pointing to `cpu.mem`
  |  (*vec)[[thread]] = 1.0
  |   ^^^^ dereferencing pointer $\texttt{in}$ `cpu.mem' memory --------------------
  |    sched(X) thread in grid.to_threads {
  |             ^^^^^^ executed by `gpu.Thread`
```

In Section 3, we introduce *execution resources* that identify *who* executes a piece of code with a focus on the GPU. However, these also extend to CPU threads. The formal type system, introduced in Section 4, extends references with *memory annotations* that strictly enforce that memory is only dereferenced in the correct execution context.

Shared Assumptions between CPU and GPU. In CUDA—and *Descend*—when launching a function on the GPU, the host thread specifies the launch configuration, i.e., the number of threads executing the kernel and their grouping into blocks. For GPU functions, there are often implicit assumptions about the number of threads that are going to execute the function as well as the amount of memory that is allocated via the host's memory API. But these assumptions are easily violated on either the CPU or GPU side, such as for this GPU function scaling a vector:

```
CUDA
__global__ scale_vec_kernel(double *vec) { vec[globalIdx.x] = vec[globalIdx.x] * 3.0; }
```

Each GPU thread accesses a single element of the vector at its index within the entire grid. The assumption made here, is that the grid contains as many threads as there are elements in the vector. For example, the following launch of the GPU function from the CPU is erroneous:

```
CUDA
cudaMalloc(&d_ptr, SIZE);
...
scale_vec_kernel<<<1, SIZE>>>(d_ptr);
```

Instead of starting as many threads as there are vector elements, the function is executed by as many threads as there are *bytes* in the vector. By launching the GPU function with more threads than vector elements, out-of-bounds memory accesses are triggered. In *Descend*, calling a GPU program with the wrong number of threads leads to an error message at compile time:

```
Descend
error: mismatched types
  |  scale_vec<<<X<1>, X<SIZE>>>>(d_vec);
  |                               ^^^^^ expected `[f64; SIZE]`, found `[f64; ELEMS]`
```

We will see in Section 3, that all functions are annotated with an execution resource describing how the function expects to be executed. This makes assumptions explicit. The type system, presented in Section 4, enforces this at compile time.

3 Safe GPU programming with *Descend*

In this section, we discuss the technical mechanism that *Descend* uses to guarantee memory safety and produce the error messages seen in the previous section. We first give explanations and

intuitions, before we will present the most important aspects of the formal type system of *Descend* in Section 4. We start by introducing *execution resources*, *place expressions*, and *views* as the central ingredients to formally reason about the execution and memory hierarchy, and to check that parallel memory accesses are performed safely.

3.1 Execution Resources

As shown in the previous section, the correctness of a GPU program often depends on the specific grid of blocks and threads that executes it. Depending on the number of threads that the kernel is executed with, a memory location may or may not be accessed by multiple threads at the same time, memory accesses may go out-of-bounds or the kernel may simply not compute a complete result. Furthermore, some instructions are required to be executed by specific sets of threads. For example, the barrier `__syncthreads` must be executed by all threads within a block, and the warp-shuffle instruction `__shufl_sync` that moves values between threads without using slow global or shared memory, must be executed by a subset of threads within the same warp.

In traditional GPU languages, the grid is specified when the GPU kernel is executed. The kernel itself has no static knowledge about the grid and can only dynamically refer to its dimensions or blocks and threads. In addition, a kernel is often not safe for many possible configurations of grids.

To enable reasoning statically about the execution of GPU functions, *Descend* introduces *execution resources*, which represent groupings of blocks and threads. The type system uses execution resources to track whether values are owned or instructions are executed by the entire grid, specific blocks, warps or individual threads. For this, we need to be able to syntactically compare execution resources, which is the reason to define them with a formal syntax.

Formal Syntax of Execution Resources. Listing 3 shows the formal grammar of execution resources. Identifiers are abstract execution resources. The execution resource `cpu.thread` represents a single thread on the CPU. The `gpu.grid` stores two *dimensions* d that describe the up to three-dimensional shape of the blocks. The size of a dimension is represented as a natural number η that can either be a constant, a variable, or simple mathematical expressions over natural numbers. For example, a two-dimensional grid that consists of 2×2 blocks, where each block consist of 4×4 threads, is represented as: $\texttt{gpu.grid}\langle \texttt{xy}\langle 2, 2\rangle, \texttt{xy}\langle 4, 4\rangle\rangle$. To refer to all blocks of the grid we write:

$$\texttt{gpu.grid}\langle \texttt{xy}\langle 2, 2\rangle, \texttt{xy}\langle 4, 4\rangle\rangle\texttt{.blocks}\langle \texttt{all; xy}\langle [0..2], [0..2]\rangle\rangle$$

In case we select everything, we use a simplified notation: $e\texttt{.blocks} = e\texttt{.blocks}\langle \texttt{all; xy}\langle [..][..]\rangle\rangle$.

Analogously, we refer to collections of threads or a one-dimensional collection of warps (which in-turn consist of a one-dimensional collection of threads). Using the *conditional selection*, we represent when only a subset of threads (or blocks, or warps) has been selected based on a runtime condition. This is important when detecting if a synchronization is safe or might lead to a deadlock.

e	::=		*Execution Resources:*	$\mathscr{d}$	::=		*Dimensions:*
		y	*identifier*			$\texttt{xyz}\langle \eta,\eta,\eta\rangle$	*3-dim*
		`cpu.thread`	*CPU Thread*			$\texttt{xy}\langle \eta,\eta\rangle \mid \texttt{xz}\langle \eta,\eta\rangle \mid \texttt{yz}\langle \eta,\eta\rangle$	*2-dim*
		$\texttt{gpu.grid}\langle \mathscr{d}, \mathscr{d}\rangle$	*GPU Grid*			$\texttt{x}\langle \eta\rangle \mid \texttt{y}\langle \eta\rangle \mid \texttt{z}\langle \eta\rangle$	*1-dim*
		$e\texttt{.blocks}\langle \texttt{c;}[\eta..\eta]_{\texttt{d}}\rangle$	*blocks*	η	::=		*Natrual Numbers:*
		$e\texttt{.warps}\langle \texttt{c;}[\eta..\eta]\rangle$	*warps*			n	*identifier*
		$e\texttt{.threads}\langle \texttt{c;}[\eta..\eta]_{\texttt{d}}\rangle$	*threads*			$\underline{\eta}$	*literal*
		$e\texttt{.forall}(d)$	*forall*			$\eta \oplus \eta$	*binary operation*
		$e[\eta..\eta]_{\texttt{d}}$	*sub-selection*	c	::=	$\texttt{cond} \mid \texttt{all}$	*conditional select*
d	::=	$\texttt{x} \mid \texttt{y} \mid \texttt{z}$	*Dim-Selector*				

Listing 3. Grammar for Execution Resources and Dimensions

Using *forall*, we quantify over a dimension, representing, e.g., a slice of blocks sharing the same index in one dimension. In a naive matrix-multiplication implementation, each row of blocks within the grid takes ownership over all the rows of the output matrix for which the block's threads compute a result. In order to express this, we need a way of referring to each row of blocks separately. Each row of blocks is represented by quantifying over the y-dimension of blocks:

$$\texttt{gpu.grid}\langle \texttt{xy}\langle 2, 2\rangle, \texttt{xy}\langle 4, 4\rangle\rangle\texttt{.blocks.forall(y)}$$

Lastly, the *sub-selection* operator changes the amount of selected sub-execution resources in a given dimension. This is useful in many scenarios. For example, to perform a reduction in a single block, some data is accumulated iteratively. With each iteration, the number of threads performing accumulations is halved. This means that only a subset of threads within the block is active, which we can represent as: $\texttt{block}\left[0..\left(4/2^i\right)\right]_x$ where i is the iteration index.

With *execution resources* we have a simple yet powerful formal way of referring to collections of blocks and threads within a multidimensional grid. Next, we have a look at how execution resources are managed in *Descend*.

Scheduling over Execution Resources. In *Descend* execution resources are not specified freely anywhere in the program. Instead, users schedule computations over every dimension of the highest level in the execution hierarchy. In the following example code, we execute a function with a two-dimensional GPU grid of 2×2 blocks, each comprised of 4×4 threads.

Descend

```
1 fn f(...) -[grid: gpu.Grid<XY<2,2>,XY<4,4>>]-> (){
2   sched(Y) blockRow in grid.blocks {
3     sched(X) block in blockRow {
4       split(Y) block.threads at 1 {
5         fstThreadGroup => ...
6         sndThreadGroup => ... } } } }
```

Here, `grid` is the execution resource that executes function `f`. The type annotation in line 1 specifies the shape of the grid. The body of the function is executed by the function's execution resource, so in this case, the entire grid. In line 2, we *schedule* the nested computation over all groups of blocks in the grid with the same y-dimension (rows of blocks). For that we specify the execution resource to schedule over (`grid.blocks`) and provide an identifier (`blockRow`) to refer to the sub-execution resources. *Descend* requires scheduling over the outer hierarchy levels completely before further scheduling computations within the inner levels. This means, that rows of blocks are not allowed to schedule computations over threads. Instead, a row of blocks must schedule the following computations over all blocks in the x-dimension. The scheduled identifier `block` is equivalent to execution resource $\texttt{gpu.grid}\langle \texttt{xy}\langle 2, 2\rangle, \texttt{xy}\langle 4, 4\rangle\rangle\texttt{.forall(y).forall(x)}$. Only when scheduling computations within a single block are we allowed to refer to the threads in the block. In line 4, each block splits the collection of its threads into two execution resources at index 1 along the y-dimension. `fstThreadGroup` contains all threads with a thread index less than 1 in y-dimension. `sndThreadGroup` contains the rest. The identifier `fstThreadGroup`, is equivalent to the following execution resource:

$$\texttt{gpu.grid}\langle \texttt{xy}\langle 2, 2\rangle, \texttt{xy}\langle 4, 4\rangle\rangle\texttt{.blocks.forall(y).forall(x).threads}\,[0..1]_y$$

The execution resources introduced here have three main purposes: *1)* they are used to checking what code is executed on the CPU and GPU; *2)* they are used to checking which instructions are executed by which part of the GPU hierarchy, such as barrier synchronization must be executed inside a block; *3)* they keep track of dimensions and sizes that are used in the code generation.

p ::=	*Place Expressions:*	v ::=	*Views*
x	*variable*	`group`::⟨η⟩	
p`.fst` | p`.snd`	*projections*	`take_left`::⟨η⟩	
*p	*dereference*	`take_right`::⟨η⟩	
p[η]	*index*	`transpose`	
p⟦e⟧	*select*	`reverse`	
$p.v$	*view*	`map`(v)	

Listing 4. Grammar for Place Expressions

3.2 Place Expressions and Views

To reason about memory locations and safe memory accesses we define *Place Expressions* and *Views*.

Place Expressions. Rust introduces the concept of a *place expression* as unique names for a memory object. Aliases are resolved by substituting the referenced place expressions. This allows them to be compared syntactically in Rust's type system to ensure that the same memory location is not (mutably) accessed simultaneously. This guarantees data race freedom.

Listing 4 shows *Descend*'s place expressions. The simplest place expression is a variable naming a region of memory. Projections `.fst` or `.snd` are applied to tuples referring to two non-overlapping regions of memory. The dereference-operator accesses the memory that a reference refers to. Single elements of an array are accessed by indexing. These place expressions exist in Rust as well.

In *Descend*, we introduce two additional place expressions: *select*s and *view*s. The select expression p⟦e`.forall`(d)⟧ distributes the ownership of elements within the array place expression p over each sub-execution resource in dimension d. This requires the execution resource (e.g., block) to consists of as many sub-execution resources (e.g., threads) as there are elements in the array. Each sub-execution resource takes ownership over one element, ensuring a safe concurrent array access.

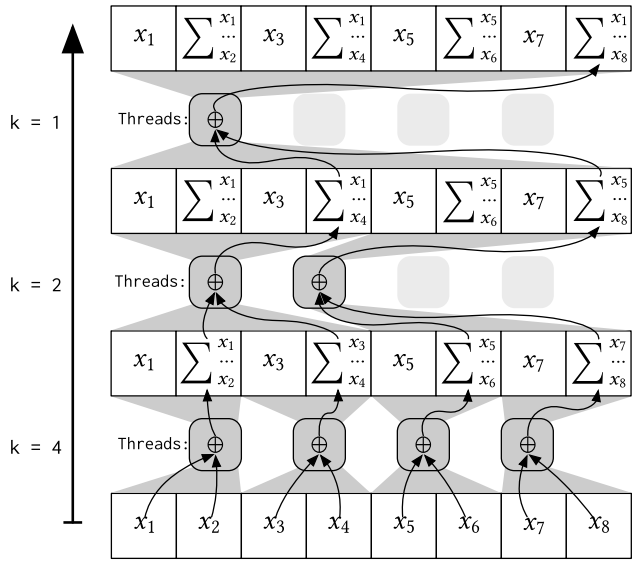
However, assigning ownership of single array elements to execution resources is very restricting. In practice, it is often required to take ownership over multiple array elements with a single thread. Similarly, the first element of an array would always be mapped to the first sub-execution resource. We require a way of reshaping which sub-execution resource takes ownership over which elements. To increase the flexibility of safe parallel memory accesses *Descend* introduces *views*.

Views. By applying a view v to an array place expression p with $p.v$, the underlying array is reshaped. The result of applying a view to an array is a view-array. View-arrays are very similar to ordinary arrays, but they are not guaranteed to be contiguous in memory. The reason for this is, that reshaping and reordering is not performed directly in memory. Instead, accesses to the resulting view-array are being transformed to create the impression of accessing data that has been reshaped and reordered in memory, while in fact, the original array's memory is being accessed. The memory layout of the original array remains the same. When generating code, views are compiled into indices following a process similar to the one taken in the Lift compiler [Steuwer, Remmelg, and Dubach 2017] and DPIA [Atkey et al. 2017].

Listing 4 shows the basic views in *Descend*, which can be combined to express more complex array accesses. View `group` combines consecutive array elements into nested arrays of a given size. This enables assigning ownership of groups of elements to an execution resource by using the select operator. All groups must have the same size by requiring that the group size perfectly divides the array size. `take_left` splits the array into two non-overlapping partial arrays at a given position and returns its left-hand side, while `take_right` returns the right-hand side of the split. We ensure that

Descend

```
1  for k in [4, 2, 1] {
2    sync(block);
3    split(X) k block.threads {
4      active => sched thread in active {
5        let myBorrow = &uniq
6          (*arr).group::<8/k>[[thread]];
7        (*myBorrow)[8/k-1] =
8            (*myBorrow)[8/k-1]
9          + (*myBorrow)[4/k-1];
10     },
11     inactive => { () }
12   }
13 }
```



Listing 5. Data owned by threads during the upsweep phase of scan algorithm

the position of the split is within the size of the input array. These views enable programmers to select only a part of an array to work on and do something else with the other part or discard it. View `transpose` transposes a two-dimensional array and `reverse` reverses the order of elements. Finally, `map` applies a view to each element of a multidimensional array.

Listing 5, shows an example of how views are used to control which thread in a block takes ownership over array elements during the upsweep phase of a work-efficient block-wide scan algorithm. The code on the left shows an implementation of the upsweep phase in *Descend*. The graphic on the right shows the individual loop iterations from the bottom to the top. The dark shaded areas show which array elements each thread has access to in each iteration. The arrows depict the data flow of the computation showing that each thread only performs reads and writes within the section of the array they have exclusive access to.

In each iteration, the number of active threads is halved using the `split` construct in line 3. Each active thread is assigned a group of elements using the `group` view in line 6. The size of the group increases with each iteration. Two elements are added, and the last owned element is overwritten with the result (lines 7–9). In between each iteration a synchronization is required (line 2) as we perform the writes to a shared array and change the memory access pattern in each iteration.

This example shows, how views are used to distribute ownership of array elements over many threads, while guaranteeing that there is a clear separation of elements between execution resources. We can give this guarantee, because all views either group, remove or permute elements of an array and select always assigns threads to distinct elements. However, redistributing elements that were already distributed previously, may still introduce the possibility of multiple execution resources sharing ownership of elements. In order to guarantee that no memory location can be (mutably) accessed by more than one thread at the same time, we must guarantee that two place expressions aren't aliases for the same memory locations. In Rust, this guarantee is achieved through borrow checking. In *Descend*, we have extended borrow checking to guarantee alias freedom between memory accessed by different execution resources.

3.3 Extended borrow checking in *Descend*

Rust uses *ownership*, *borrowing*, and *lifetimes* to statically guarantee that there are no data races, memory leaks, use-after-free errors and other memory related errors. Ownership ensures that there is no aliasing of memory objects, as on assignment to a new variable the value can only be accessed via the new variable. Attempts to access memory via the old variable lead to compile-time errors.

As ownership alone is too restrictive, with borrowing, a restricted notion of aliasing is reintroduced into the language. The *borrow checker* checks if a thread is allowed to create a (unique or shared) reference to a memory object, i.e., "borrow" it. Multiple shared references can be used at the same time, for reading only. A unique reference, guarantees that there are no other ways to access the same memory location. It is, therefore, safe to mutate the underlying memory.

Finally, lifetimes ensure that the underlying memory of a reference hasn't been deallocated. Attempting to dereference when the memory might have been freed results in a compiler error.

Descend's extended borrow checker. On the CPU, *Descend* implements the same rules as Rust. On the GPU side, the ownership, and borrowing rules are extended and diverge from Rust. In Rust, ownership always belongs to a single thread. In *Descend*, each execution resource, such as the grid or a block might take ownership of a memory object or create references, i.e., they might borrow. This means that collections of blocks or threads, as well as single threads, own and borrow memory objects, formally represented as place expressions. For a single thread to be able to write into a memory location by having exclusive access to it, the ownership and borrows must be *narrowed* using *Descend*'s hierarchical scheduling, selections and views.

Narrowing describes how ownership and borrows are refined when navigating the execution hierarchy from grid, to blocks and threads. For example, the ownership of an array by a grid is narrowed to the grid's blocks by the blocks collectively borrowing the array, each block a distinct part. This might be further narrowed to the block's threads. But narrowing can be violated:

Descend

```
1 fn kernel(arr: &uniq gpu.global [f32; 1024]) -[grd: gpu.Grid<X<32>,X<32>>]-> () {
2   sched(X) block in grd {
3     let in_borrow = &uniq *arr; // Narrowing violated
4     sched(X) thread in block {
5       let group = &uniq arr.group::<32>[[thread]]; // Narrowing violated
6       arr.group::<32>[[block]][[thread]]; } } }
```

In the example, the parameter `arr` is owned by the grid. Attempting to borrow `arr` in line 3 *after* having scheduled the blocks of the grid violates narrowing, because each block in the grid would get unique write access to the entire array.

Another narrowing violation is shown in line 5. The array is grouped so that there are as many groups as threads per block. Then each thread selects a group and borrows that group uniquely. However, the same selection is performed for each block, because there was no selection for `block`. Therefore, threads from different blocks would gain access to the same memory location.

Line 6 shows correct narrowing. The array is grouped, and each block exclusively borrows a part of the array, before each thread in each block selects an element from it.

Synchronization. Descend's narrowing ensures that no two threads have mutable access to the same memory location. However, sometimes we do want to communicate with another thread via shared memory and then the other thread must be able to access the same memory location as well. We, therefore, need a way to allow an subsequent access by another thread while guaranteeing that this access cannot lead to a data race. By synchronizing threads at a barrier, we get the guarantee that all memory accesses before the barrier cannot conflict with memory accesses after the barrier.

3.4 Shared Write Access through Atomics

Up until this point, we have explained how *Descend* enforces that two threads cannot (mutably) perform write access to the same memory location. However, this requirement is often too strict. Some applications, such as histograms, require threads to update memory locations concurrently. This is achieved without race conditions by using atomic read-modify-write (RMW) operations.

Descend supports a limited form of RMW operations via atomic types that resemble the atomic types in Rust: RMW operations are performed on values with atomic types that are referred to by shared references. This allows multiple threads to perform RMW operations through the same reference, without violating the extended borrow checking rules. For example, operation `atomic_fetch_add_u32` has the following function signature:

```
Descend
atomic_fetch_add_u32<r:prv, m:mem>(atom_ref:&r shrd m AtomicU32, val:u32) -[t:gpu.Thread]-> u32
```

The function takes two arguments: a shared reference to a memory location and a value to add to the memory location. It must be called by a single thread and returns the value stored at the memory location before performing the RMW.

In the current version of *Descend* RMW operations are performed with a relaxed memory ordering, which does not impose constraints on other reads and writes. Implementations that require a more sophisticated use of atomics must resort to unsafe *Descend* code, possibly using inlined CUDA code. In future work, we would like to investigate weak memory models and other memory orderings such as sequential consistency and acquire-release.

3.5 Escaping Restrictions Using Unsafe

For some algorithms, there is no statically known partitioning of the memory that *Descend*'s views can express. Even when views can be used to express the general access pattern, indices and natural numbers must be statically guaranteed to be within array bounds or fulfill constraints such as a size being dividable by a specific number, e.g., the size of a `group` view must divide the size of the grouped array. This can be a problem for such algorithms, for example for graph algorithms, where the nodes are often stored linearly in memory and referred to with an index that is read from memory or computed at runtime, depending on the shape of the graph.

Descend offers programmers a way to escape these restrictions, enabling them to write programs that would otherwise not be expressible in *Descend* – at the cost of introducing unsafe blocks of code. Preceding an expression with `unsafe` tells the type checker to not perform extended borrow checking as well as constraint checks. For example, an optimized implementation of the Single-Source Shortest Path (SSSP) algorithm for GPU, chooses edges to process at runtime. After choosing the edge, the index of the edge's destination node `dst` is read from memory and then the array `node_data` containing the data for all the graph's nodes is accessed as follows:

```
Descend
unsafe &shrd (*(*graph).node_data)[dst]
```

Without `unsafe` the above line would not type check, as `dst` is not known statically when checking that the index is within the range of the array. The SSSP implementation uses a highly optimized worklist to track nodes whose distances must be (re-)computed. To avoid data races on this worklist while still being efficient, the implementation utilizes CUDA's weak memory model. This use of atomics goes beyond what can currently be expressed in safe *Descend*. Using `unsafe` we can write inline CUDA code to call directly into the existing worklist CUDA library. `Unsafe` code lacks the strong safety guarantees and must be used with caution, but it greatly increases the usefulness of *Descend* while we gradually work on increasing the expressiveness of safe code.

3.6 Handling Separated Memories in *Descend*

Tracking Memory Spaces. Descend annotates all reference types with *address spaces*. This is similarly done in CUDA, but CUDA does not have an address space for CPU pointers and generally does not strictly enforce their correct use. In *Descend*, the `cpu.mem` address space comprises values stored in the CPU stack and heap. For the GPU, we differentiate between the global `gpu.global`, shared memories `gpu.shared` and thread local memories `gpu.local`, which have separate address spaces. Using execution resources, *Descend* enforces that references are only dereferenced in the correct execution context, such as preventing dereferencing a GPU reference on the CPU. *Descend* also supports polymorphism over memory spaces, by introducing a type-level variable *m* that is used in place for a concrete address space.

Allocating Memory. Dynamic memory allocations, i.e., allocations on the CPU heap and in global GPU memory, are managed via unique smart pointers to ensure that they are freed safely without leaking memory. We call their types @-types, as they carry an annotation *at* which address space they have been allocated. The memory is freed when the smart pointer is destroyed at the end of a scope. Therefore, our type `T @ cpu.mem` corresponds to `Box<T>` in Rust. The following code shows how memory is allocated and initialized:

Descend

```
1 { let cpu_array: [i32,n] @ cpu.mem = CpuHeap::new([0;n]);
2   { let global_array: [i32;n] @ gpu.global = GpuGlobal::alloc_copy(&cpu_array);
3   } // free global_array
4 } // free heap_array
```

In the outer block, heap memory is allocated and initialized with an array of size `n` filled with 0. The smart pointer that manages the allocation is then stored in variable `cpu_array`. In the inner block, GPU global memory is allocated for the data pointed to by `heap_array`, the data is copied to the GPU and the resulting smart pointer is stored in `global_array`. The type annotations shown here are optional, but show the information stored in the type.

3.7 Making Implicit Assumptions Explicit in *Descend*

The CPU program is responsible for scheduling a GPU function for execution. In *Descend*, this is a special function call, as in CUDA, where not just the function arguments are provided, but also the executing GPU grid is specified; here comprising 32 blocks with 32 threads each:

Descend

```
scale_vec::<<<X<32>, X<32>>>>(&uniq vec);
```

In contrast to CUDA, in *Descend*, the GPU function signature carries the information on which grid configuration is allowed to execute the function:

Descend

```
fn scale_vec(vec: &uniq gpu.global [i32; 1024]) -[grid: gpu.grid<X<32>, X<32>>]-> ();
```

Descend checks that the call site and the function declaration matches, to ensure that the assumptions about how the function is written and how it is invoked do not diverge. *Descend* also supports polymorphism over grid sizes, allowing GPU functions to be written that, for example, launch as many threads as the size of the input array. In this case, the call site specifies the concrete values that are used for instantiating the grid size variables.

The CPU thread waits for the GPU function call to finish, meaning there is an implicit synchronization of the GPU grid at the end of each GPU computation.

t ::=		*Term:*
	$\underline{l}$	*literal*
	$\ominus\, t \mid t \oplus t$	*unary & binary operation*
	p	*place expression*
	$\&r\ \omega\ p$	*reference/borrow*
	$p = t$	*assignment*
	`for` n `in` $\eta..\eta\ \{\ t\ \}$	*static for-loop*
	`while` $t\ \{\ t\ \}$	*while-loop*
	`if` $t\ \{\ t\ \}$ `else` $\{\ t\ \}$	*if-else*
	$t\ ;\ t$	*sequence*

v ::=	*Views*
	`group`::$\langle\eta\rangle$
	`take_left`::$\langle\eta\rangle$
	`take_right`::$\langle\eta\rangle$
	`transpose`
	`reverse`
	`map`(v)

Listing 6. Formal syntax of *Descend* terms

4 The Type System of *Descend*

In this section, we present the formal foundations of *Descend*, including the syntax of terms and types, and the most important typing rules, explaining the formal reasoning behind ensuring safety. Our type system is based on the formalization of Rust's type system in Oxide [Weiss et al. 2019]. A technical report with the full type system of *Descend* is available at https://descend-lang.org.

4.1 Syntax of types

5 Code Generation and Evaluation

6 Related Work

7 Conclusion

Bibliography

Robert Atkey, Michel Steuwer, Sam Lindley, and Christophe Dubach. 2017. Strategy Preserving Compilation for Parallel Functional Code. *Corr*. http://arxiv.org/abs/1710.08332

Michael Bauer, Sean Treichler, and Alex Aiken. 2014. Singe: Leveraging Warp Specialization for High Performance on Gpus. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Ppopp '14, Orlando, FL, USA, February 15-19, 2014*, José E. Moreira and James R. Larus (Eds.). ACM, 119–30. https://doi.org/10.1145/2555243.2555258

Michael Garland and David Blair Kirk. 2010. Understanding Throughput-Oriented Architectures. *Commun. ACM* 53, 11 (2010), 58–66. https://doi.org/10.1145/1839676.1839694

Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-Programming with Nested Parallelism and in-Place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 556–71. https://doi.org/10.1145/3062341.3062354

Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben van Werkhoven, and Henri E. Bal. 2023. Optimization Techniques for GPU Programming. *ACM Comput. Surv.* 55, 11 (March 2023). https://doi.org/10.1145/3570638

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2021. Safe Systems Programming in Rust. *Commun. ACM* 64, 4 (2021), 144–52

Marcin Knap and Pawel Czarnul. 2019. Performance Evaluation of Unified Memory with Prefetching and Oversubscription for Selected Parallel CUDA Applications on NVIDIA Pascal and Volta Gpus. *J. Supercomput.* 75, 11 (2019), 7625–45. https://doi.org/10.1007/S11227-019-02966-8

Raphael Landaverde, Tiansheng Zhang, Ayse K. Coskun, and Martin C. Herbordt. 2014. An Investigation of Unified Memory Access Performance in CUDA. In *IEEE High Performance Extreme Computing Conference, HPEC 2014, Waltham, MA, USA, September 9-11, 2014*. IEEE, 1–6. https://doi.org/10.1109/HPEC.2014.7040988

Wenqiang Li, Guanghao Jin, Xuewen Cui, and Simon See. 2015. An Evaluation of Unified Memory Technology on NVIDIA Gpus. In *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Ccgrid 2015, Shenzhen, China, May 4-7, 2015*. IEEE Computer Society, 1092–98. https://doi.org/10.1109/CCGRID.2015.105

Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: A Functional Data-Parallel IR for High-Performance GPU Code Generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, Vijay Janapa Reddi, Aaron Smith, and Lingjia Tang (Eds.). ACM, 74–85. http://dl.acm.org/citation.cfm?id=3049841

Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. 2019. Oxide: The Essence of Rust. *Corr*. http://arxiv.org/abs/1903.00982

Mingyuan Wu, Yicheng Ouyang, Husheng Zhou, Lingming Zhang, Cong Liu, and Yuqun Zhang. 2020. Simulee: Detecting CUDA Synchronization Bugs via Memory-Access Modeling. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 937–48. https://doi.org/10.1145/3377811.3380358