Descend

A Safe GPU Systems Programming Language

Bastian Köpcke, Sergei Gorlatch, Michel Steuwer

University of Münster, Technische Universität Berlin

June 27, 2024

Computing on GPUs is great!

GPU Programming is too hard!

CUDA

```cuda
__global__ void transpose(const float *input, float *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];
  }
}
```

Kernel functions are executed by thousands of threads

CUDA

```cuda
__global__ void transpose(const float *input, float *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];
  }
}
```

Kernel functions are executed by thousands of threads

Threads are hierarchically organized into multi-dimensional *blocks*
Blocks are hierarchically are organized into a multi-dimensional *grid*

CUDA

```cuda
__global__ void transpose(const float *input, float *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];
  }
}
```

explicit indexing to determine where a thread accesses memory

Kernel functions are executed by thousands of threads

Threads are hierarchically organized into multi-dimensional *blocks*
Blocks are hierarchically are organized into a multi-dimensional *grid*

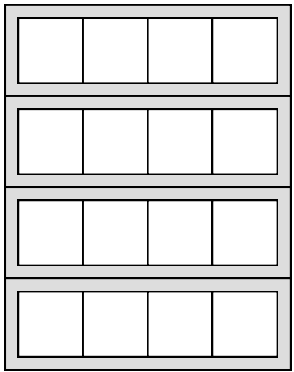
CUDA

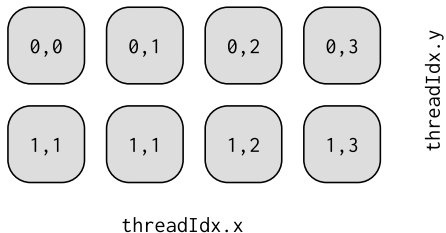
```
1  __global__ void transpose(const float *input, float *output) {
2    __shared__ float tmp[1024];
3    for (int j = 0; j < 32; j += 8) {
4      tmp[threadIdx.y+j*32+threadIdx.x]
5        input[(blockIdx.y*32+threadIdx.y+j)*2048 + blockIdx.x*32+threadIdx.x];
6    }
7    __syncthreads();
8    for (int j = 0; j < 32; j += 8) {
9      output[(blockIdx.x*32+threadIdx.y+j)*2048 + blockIdx.y*32+threadIdx.x] =
10       tmp[threadIdx.x*32+threadIdx.y+j];
11   }
12 }
```

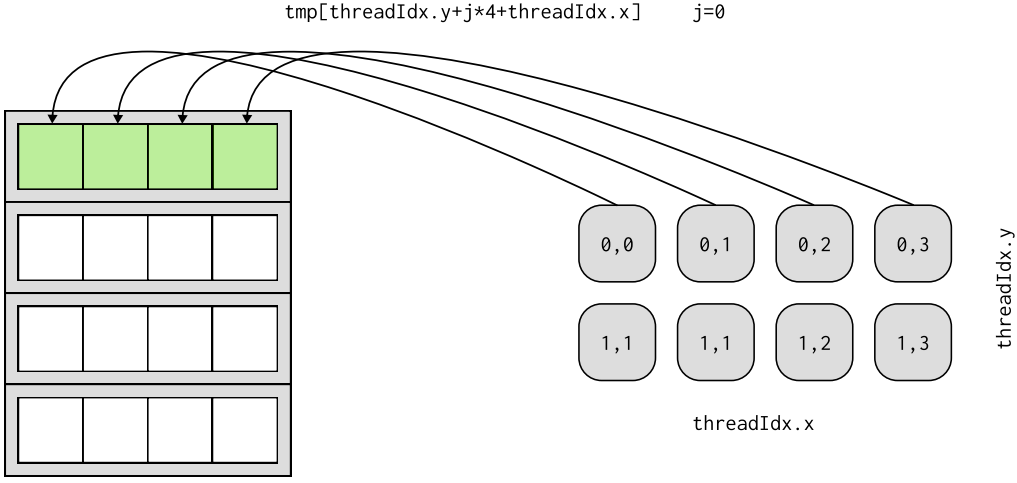
explicit indexing to determine where a thread accesses memory

Is the implementation correct?

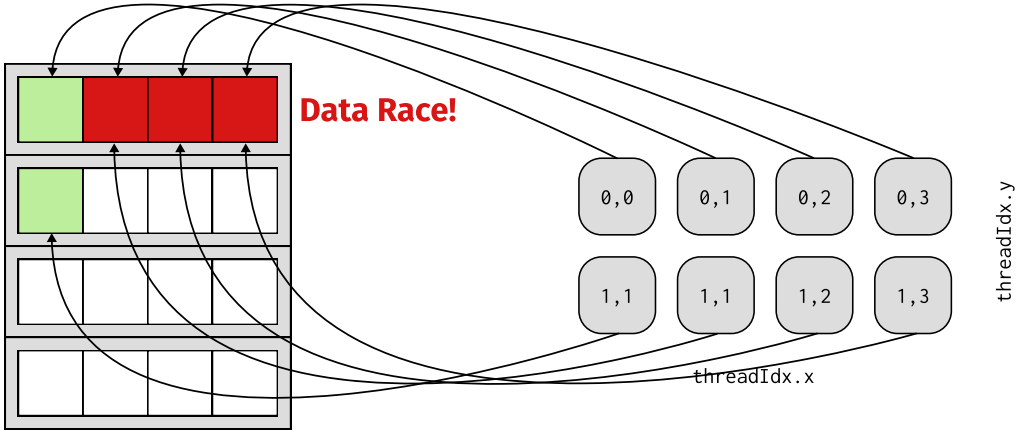
tmp[threadIdx.y+j*4+threadIdx.x] j=0





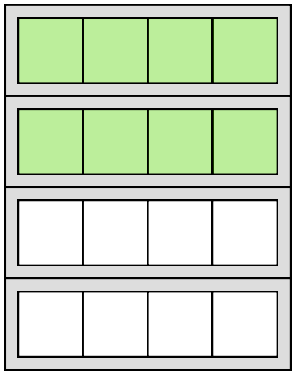

tmp[threadIdx.y+j*4+threadIdx.x]
j=0
0,0
0,1
0,2
0,3
1,1
1,1
1,2
1,3
threadIdx.y
threadIdx.x

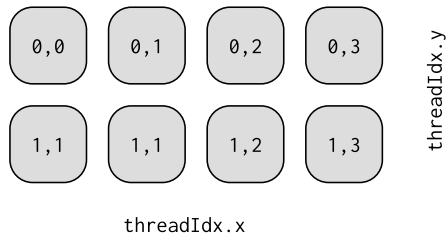
tmp[threadIdx.y+j*4+threadIdx.x] j=0



~~tmp[threadIdx.y+j*4+threadIdx.x]~~ j=0

tmp[(threadIdx.y+j)*4+threadIdx.x]





How can we create a safe GPU programming language?

Descend

A Safe GPU Systems Programming Language

"if the program successfully compiles,
then it is free of data races and other memory problems"

Descend

```
fn transpose(
  input: &gpu.global [[f32;2048];2048],
  output: &uniq gpu.global [[f32;2048];2048],
  [grid.blocks.forall(X).forall(Y)] tmp: &uniq gpu.shared [[f32;32];32]
) -[grid: gpu.grid<XY<64,64>,XY<32,8>>]-> () {
  sched(Y,X) block in grid {
    sched(Y,X) thread in block {
      for i in 0..4 {
        tmp.transpose.group_by_row::<32,4>[[thread]][i] =
          input.group_by_tile::<32,32>.transpose[[block]].group_by_row::<32,4>[[thread]][i]
      };

      for i in 0..4 {
        output.group_by_tile::<32,32>[[block]].group_by_row::<32,4>[[thread]][i]=
          tmp.group_by_row::<32,4>[[thread]][i]
      }
    }
  }
}
```

Descend

```
1  fn transpose(
2    input: &gpu.global [[f32;2048];2048],
3    output: &uniq gpu.global [[f32;2048];2048],
4    [grid.blocks.forall(X).forall(Y)] tmp: &uniq gpu.shared [[f32;32];32]
5  ) -[grid: gpu.grid<XY<64,64>,XY<32,8>>]-> () {
6    sched(Y,X) block in grid {
7      sched(Y,X) thread in block {
8        for i in 0..4 {
9          tmp.transpose.group_by_row::<32,4>[[thread]][i] =
10           input.group_by_tile::<32,32>.transpose[[block]].group_by_row::<32,4>[[thread]][i]
11       };
12
13       for i in 0..4 {
14         output.group_by_tile::<32,32>[[block]].group_by_row::<32,4>[[thread]][i]=
15           tmp.group_by_row::<32,4>[[thread]][i]
16       }
17     }
18   }
19 }
```

function is explicitly annotated with a 2D **grid** of 64 × 64 **blocks** with 32 × 8 **threads** each

Descend

```
fn transpose(
  input: &gpu.global [[f32;2048];2048],
  output: &uniq gpu.global [[f32;2048];2048],
  [grid.blocks.forall(X).forall(Y)] tmp: &uniq gpu.shared [[f32;32];32]
) -[grid: gpu.grid<XY<64,64>,XY<32,8>>]-> () {
  sched(Y,X) block in grid {
    sched(Y,X) thread in block {
      for i in 0..4 {
        tmp.transpose.group_by_row::<32,4>[[thread]][i] =
          input.group_by_tile::<32,32>.transpose[[block]].group_by_row::<32,4>[[thread]][i]
      };

      for i in 0..4 {
        output.group_by_tile::<32,32>[[block]].group_by_row::<32,4>[[thread]][i]=
          tmp.group_by_row::<32,4>[[thread]][i]
      }
    }
  }
}
```

collectively executed by the entire grid

Descend

```
fn transpose(
  input: &gpu.global [[f32;2048];2048],
  output: &uniq gpu.global [[f32;2048];2048],
  [grid.blocks.forall(X).forall(Y)] tmp: &uniq gpu.shared [[f32;32];32]
) -[grid: gpu.grid<XY<64,64>,XY<32,8>>]-> () {
  sched(Y,X) block in grid {
    sched(Y,X) thread in block {
      for i in 0..4 {
        tmp.transpose.group_by_row::<32,4>[[thread]][i] =
          input.group_by_tile::<32,32>.transpose[[block]].group_by_row::<32,4>[[thread]][i]
      };

      for i in 0..4 {
        output.group_by_tile::<32,32>[[block]].group_by_row::<32,4>[[thread]][i]=
          tmp.group_by_row::<32,4>[[thread]][i]
      }
    }
  }
}
```

collectively executed by each block in the grid

Descend

```
fn transpose(
  input: &gpu.global [[f32;2048];2048],
  output: &uniq gpu.global [[f32;2048];2048],
  [grid.blocks.forall(X).forall(Y)] tmp: &uniq gpu.shared [[f32;32];32]
) -[grid: gpu.grid<XY<64,64>,XY<32,8>>]-> () {
  sched(Y,X) block in grid {
    sched(Y,X) thread in block {
      for i in 0..4 {
        tmp.transpose.group_by_row::<32,4>[[thread]][i] =
          input.group_by_tile::<32,32>.transpose[[block]].group_by_row::<32,4>[[thread]][i]
      };

      for i in 0..4 {
        output.group_by_tile::<32,32>[[block]].group_by_row::<32,4>[[thread]][i]=
          tmp.group_by_row::<32,4>[[thread]][i]
      }
    }
  }
}
```

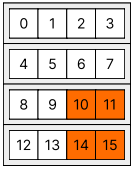
collectively executed by each thread in the block

Descend

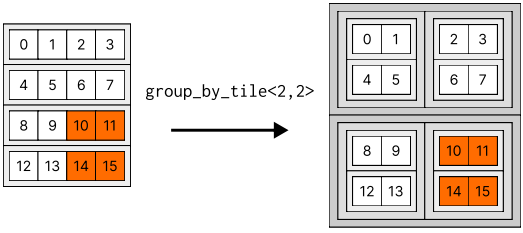
```
1  fn transpose(
2    input: &gpu.global [[f32;2048];2048],
3    output: &uniq gpu.global [[f32;2048];2048],
4    [grid.blocks.forall(X).forall(Y)] tmp: &uniq gpu.shared [[f32;32];32]
5  ) -[grid: gpu.grid<XY<64,64>,XY<32,8>>]-> () {
6    sched(Y,X) block in grid {
7      sched(Y,X) thread in block {
8        for i in 0..4 {
9          tmp.transpose.group_by_row::<32,4>[[thread]][i] =
10           input.group_by_tile::<32,32>.transpose[[block]].group_by_row::<32,4>[[thread]][i]
11       };
12
13       for i in 0..4 {
14         output.group_by_tile::<32,32>[[block]].group_by_row::<32,4>[[thread]][i] =
15           tmp.group_by_row::<32,4>[[thread]][i]
16       }
17     }
18   }
19 }
```

safely write in parallel over blocks and threads using **views**

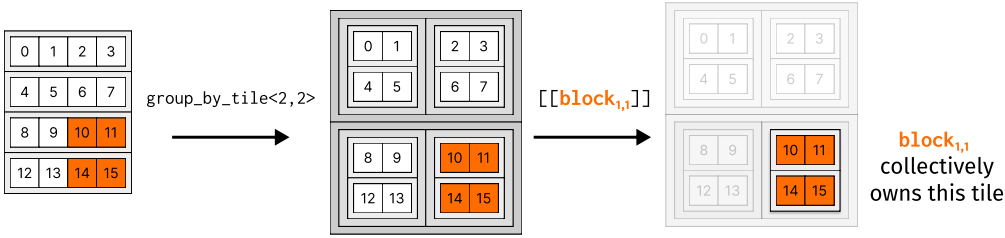
`output.group_by_tile<2,2>[[block_{1,1}]].group_by_row<2,2>[[ thread_{0,1} ]]`



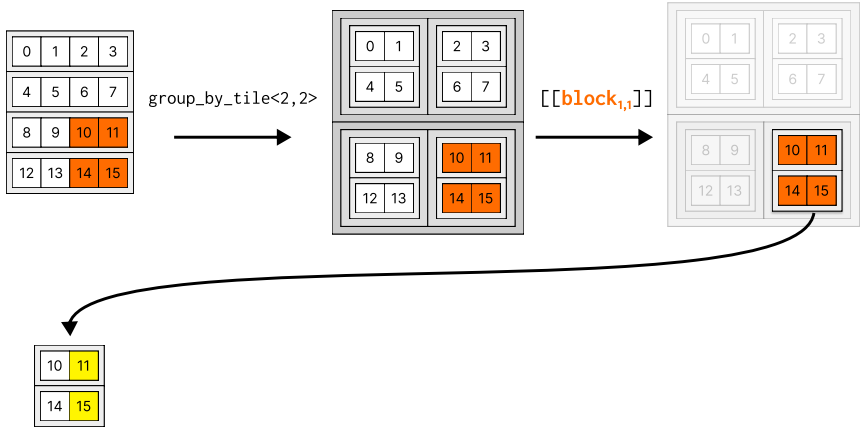
output.group_by_tile<2,2>[[block$_{1,1}$]].group_by_row<2,2>[[thread$_{0,1}$]]



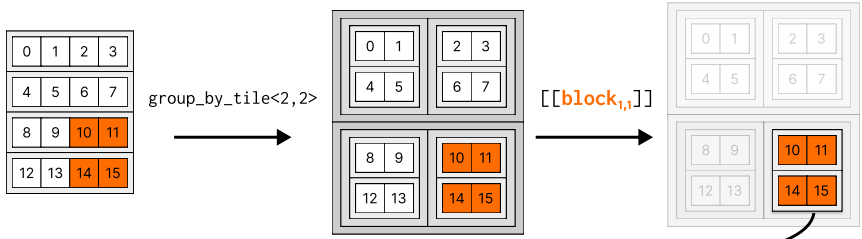
output.group_by_tile<2,2>[[block$_{1,1}$]].group_by_row<2,2>[[thread$_{0,1}$]]



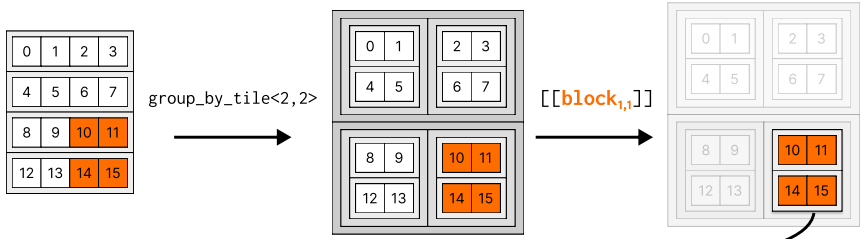
output.group_by_tile<2,2>[[block$_{1,1}$]].group_by_row<2,2>[[thread$_{0,1}$]]

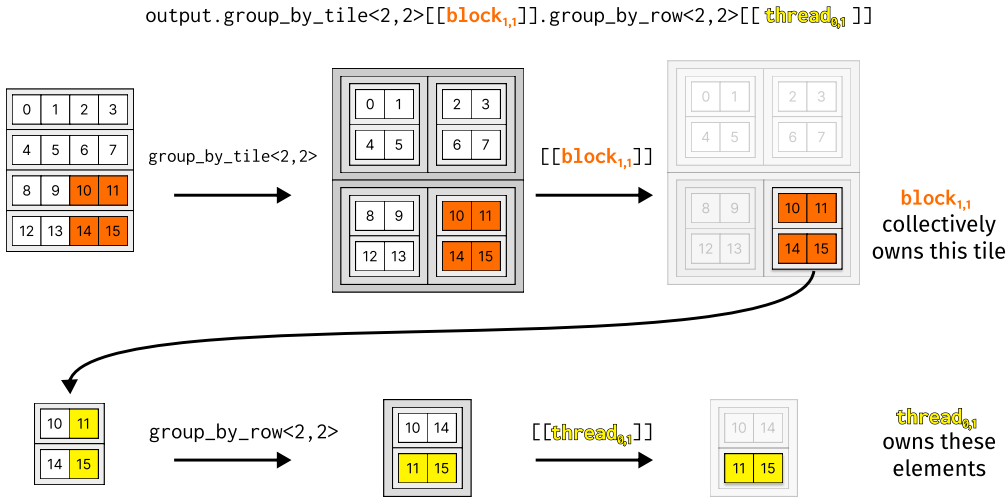


output.group_by_tile<2,2>[[block$_{1,1}$]].group_by_row<2,2>[[thread$_{0,1}$]]



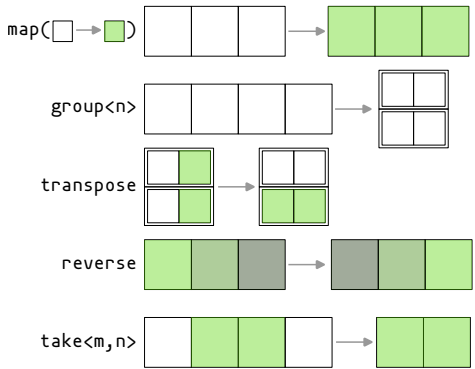
output.group_by_tile<2,2>[[block$_{1,1}$]].group_by_row<2,2>[[thread$_{0,1}$]]



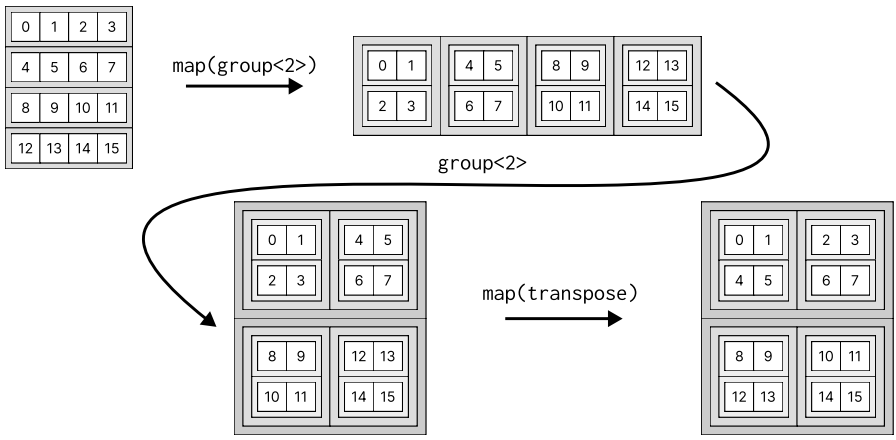


Descend's extended ownership model ensures that memory is accessed safely via views

Descend's five basic view primitives



```
group_by_tile<2,2> = map(group<2>).group<2>.map(transpose)
```



Views are composed to express complex memory access patterns

Descend

```
1  fn transpose(
2    input: &gpu.global [[f32;2048];2048],
3    output: &uniq gpu.global [[f32;2048];2048],
4    [grid.blocks.forall(X).forall(Y)] tmp: &uniq gpu.shared [[f32;32];32]
5  ) -[grid: gpu.grid<XY<64,64>,XY<
6    sched(Y,X) block in grid {
7      sched(Y,X) thread in block {
8        for i in 0..4 {
9          tmp.group_by_row::<32,4>[[thread]][i] =
10           input.group_by_tile::<32,32>.transpose[[block]].group_by_row::<32,4>[[thread]][i]
11       };
12
13       for i in 0..4 {
14         output.group_by_tile::<32,32>[[block]].group_by_row::<32,4>[[thread]][i] =
15           tmp.transpose.group_by_row::<32,4>[[thread]][i]
16       }
17     }
18   }
19 }
```

threads write safely into memory tmp

Descend

```
1  fn transpose(
2    input: &gpu.global [[f32;2048];2048],
3    output: &uniq gpu.global [[f32;2048];2048],
4    [grid.blocks.forall(X).forall(Y)] tmp: &uniq gpu.shared [[f32;32];32]
5  ) -[grid: gpu.grid<XY<64,64>,XY<32,8>>]-> () {
6    sched(Y,X) block in grid {
7      sched(Y,X) thread in block {
8        for i in 0..4 {
9          tmp.group_by_row::<32,4>[[thread]][i] =
10           input.group_by_tile::<32,32>.transpose[[block]].group_by_row::<32,4>[[thread]][i]
11       };
12
13       for i in 0..4 {
14         output.group_by_tile::<32,32>[[block]].group_by_row::<32,4>[[thread]][i] =
15           tmp.transpose.group_by_row::<32,4>[[thread]][i]
16       }
17     }
18   }
19 }
```

threads read safely from memory tmp

Descend

```
1  fn transpose(
2    input: &gpu.global [[f32;2048];2048],
3    output: &uniq gpu.global [[f32;2048];2048],
4    [grid.blocks.forall(X).forall(Y)] tmp: &uniq gpu.shared [[f32;32];32]
5  ) -[grid: gpu.grid<XY<64,64>,XY<32,8>>]-> () {
6    sched(Y,X) block in grid {
7      sched(Y,X) thread in block {
8        for i in 0..4 {
9          tmp.group_by_row::<32,4>[[thread]][i] =
10           input.group_by_tile::<32,32>.transpose[[block]].group_by_row::<32,4>[[thread]][i]
11       };
12
13       for i in 0..4 {
14         output.group_by_tile::<32,32>[[block]].group_by_row::<32,4>[[thread]][i] =
15           tmp.transpose.group_by_row::<32,4>[[thread]][i]
16       }
17     }
18   }
19 }
```

~~threads read safely from memory tmp~~

Descend

```
fn transpose(
  input: &gpu.global [[f32;2048];2048],
  output: &uniq gpu.global [[f32;2048];2048],
  [grid.blocks.forall(X).forall(Y)] tmp: &uniq gpu.shared [[f32;32];32]
) -[grid: gpu.grid<XY<64,64>,XY<32,8>>]-> () {
  sched(Y,X) block in grid {
    sched
      for
        t

      };

      for
        o

      }
    }
  }
}
```

missing synchronization leads to an error

Descend

```
error: conflicting memory access
  | line 15: tmp.transpose.group_by_row::<32,4>[[thread]][i]
  |           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  | line 9: tmp.group_by_row::<32,4>[[thread]][i]
  |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Descend

```
fn transpose(
  input: &gpu.global [[f32;2048];2048],
  output: &uniq gpu.global [[f32;2048];2048],
  [grid.blocks.forall(X).forall(Y)] tmp: &uniq gpu.shared [[f32;32];32]
) -[grid: gpu.grid<XY<64,64>,XY<32,8>>]-> () {
  sched(Y,X) block in grid {
    sched(Y,X) thread in block {
      for i in 0..4 {
        tmp.group_by_row::<32,4>[[thread]][i] =
          input.group_by_tile::<32,32>.transpose[[block]].group_by_row::<32,4>[[thread]][i]
      };
      sync(block);
      for i in 0..4 {
        output.group_by_tile::<32,32>[[block]].group_by_row::<32,4>[[thread]][i] =
          tmp.transpose.group_by_row::<32,4>[[thread]][i]
      }
    }
  }
}
```

explicit barrier synchronization on the entire block

Descend

```
fn transpose(
  input: &gpu.global [[f32;2048];2048],
  output: &uniq gpu.global [[f32;2048];2048],
  [grid.blocks.forall(X).forall(Y)] tmp: &uniq gpu.shared [[f32;32];32]
) -[grid: gpu.grid<XY<64,64>,XY<32,8>>]-> () {
  sched(Y,X) block in grid {
    sched(Y,X) thread in block {
      for i in 0..4 {
        tmp.group_by_row::<32,4>[[thread]][i] =
          input.group_by_tile::<32,32>.transpose[[block]].group_by_row::<32,4>[[thread]][i]
      };
      sync(block);
      for i in 0..4 {
        output.group_by_tile::<32,32>[[block]].group_by_row::<32,4>[[thread]][i] =
          tmp.transpose.group_by_row::<32,4>[[thread]][i]
      }
    }
  }
}
```

Descend ensures that there are not data races

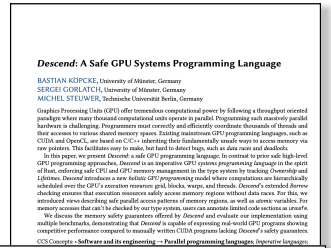
Current State of Descend

Further guarantees provided by Descend:

- Respect separation between CPU & different GPU memories
- Assumptions between CPU and GPU code match
- Safe memory accesses in branching code
- Absence of barrier divergence

What we did:

- Formalized the type system
- Implemented compiler to CUDA



***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 resource: 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.**

3 Typing Rules

$$\Delta;\Gamma \mid e;\mathsf{a} \mid t \vdash \boxed{\delta :: \Gamma'} \dashv \mathsf{A}' \mid$$

T-White-Local

$$\frac{\mathsf{isPlace}(p) \qquad \Delta;\Gamma \mid y;\varepsilon;e \mid \mathsf{A} \vdash \boxed{t :: \delta_t} \dashv \Gamma' \mid \mathsf{A}' \qquad \Gamma'(p) = (\hat{\Delta}_p, \pi_p) \qquad e \rightrightarrows e_p \qquad \Delta;\Gamma' \vdash^{e} \delta_t \rightsquigarrow \hat{\delta}_p \dashv \Gamma' \qquad \Gamma' \vdash^{\mathsf{uniq}_{\mathsf{lft}}} \mathbf{borrow}\ p \Rightarrow \{\ {}^{\mathsf{uniq}_{\mathsf{lft}}}p\ \}}{\Delta;\Gamma \mid y;\varepsilon;e \mid \mathsf{A} \vdash \boxed{p = t : \mathbf{unit}} \dashv \Gamma[p \mapsto \hat{\delta}_t] \mid \mathsf{A}'}$$

T-White-SharedMem

$$\frac{\neg\mathsf{isPlace}(p) \qquad y :\varepsilon \vdash e : \mathbf{GpuThread} \qquad \Delta;\Gamma \mid y;\varepsilon;e \mid \mathsf{A} \vdash \boxed{t :: \delta_t} \dashv \Gamma' \mid \mathsf{A}' \qquad \Delta;\Gamma' \mid y;\varepsilon;e \mid \mathsf{A}' \vdash^{\mathsf{uniq}_{\mathsf{lft}}} p :: \delta_p \qquad \Delta;\Gamma' \vdash^{e} \delta_t \rightsquigarrow \delta_p \dashv \Gamma' \qquad \Gamma' \vdash^{\mathsf{uniq}_{\mathsf{lft}}} \mathbf{borrow}^{e}\ p \Rightarrow \{\ {}^{\mathsf{uniq}_{\mathsf{lft}}}p\ \}}{\Delta;\Gamma \mid y;\varepsilon;e \mid \mathsf{A} \vdash \boxed{p = t : \mathbf{unit}} \dashv \Gamma^{\mathsf{w}} \mid \mathsf{A}', \ell}$$

T-Read-By-Copy

$$\frac{\mathsf{isPlace}(p) \qquad \mathsf{isCopyable}(\delta) \qquad \Delta;\Gamma \mid y;\varepsilon;e \mid \mathsf{A} \vdash^{\mathsf{shrd}}_{\mathsf{pl}} p :: \delta \qquad \Gamma \vdash^{\mathsf{shrd}} \mathbf{borrow}^{e}\ p \Rightarrow \{\ {}^{\mathsf{shrd}}p\ \}}{\Delta;\Gamma \mid y;\varepsilon;e \mid \mathsf{A} \vdash \boxed{p :: \delta} \dashv \Gamma \mid \mathsf{A}}$$

T-Read-By-Move

$$\frac{\mathsf{isPlace}(p) \qquad \neg\mathsf{isCopyable}(\delta) \qquad \Delta;\Gamma \mid y;\varepsilon;e \vdash^{\mathsf{shrd}}_{\mathsf{pl}} p :: \delta \qquad \Delta;\Gamma \mid y;\varepsilon;e \mid \mathsf{A} \vdash^{\mathsf{uniq}} \mathbf{borrow}^{e}\ p \Rightarrow \{\ {}^{\mathsf{uniq}}p\ \}}{\Delta;\Gamma \mid y;\varepsilon;e \mid \mathsf{A} \vdash \boxed{p :: \delta} \dashv \Gamma[p \mapsto \delta^{\dagger}] \mid \mathsf{A}}$$

T-Read-SharedMem

$$\frac{\neg\mathsf{isPlace}(p) \qquad \mathsf{isCopyable}(\delta) \qquad \Delta;\Gamma \mid y;\varepsilon;e \vdash^{\mathsf{shrd}}_{\mathsf{pl}} p :: \delta \qquad \Delta;\Gamma \mid y;\varepsilon;e \mid \mathsf{A} \vdash^{\mathsf{shrd}} \mathbf{borrow}^{e}\ p \Rightarrow \{\ {}^{\mathsf{shrd}}p\ \}}{\Delta;\Gamma \mid y;\varepsilon;e \mid \mathsf{A} \vdash \boxed{p :: \delta} \dashv \Gamma \mid \mathsf{A}, \ell}$$

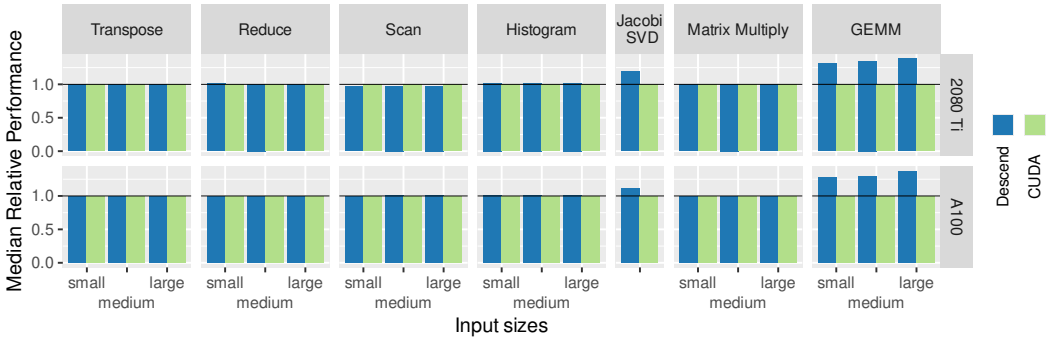
T-Borrow

$$\frac{\neg\mathsf{isPlace}(p) \qquad \Gamma(r) = \emptyset \qquad \Delta;\Gamma \mid y;\varepsilon;e \vdash^{\omega}_{\mathsf{pl}} p :: \delta, \mu \qquad \Delta;\Gamma \mid y;\varepsilon;e \mid \mathsf{A} \vdash^{\omega} \mathbf{borrow}^{e}\ p \Rightarrow \{\ {}^{\omega}p\ \}}{\Delta;\Gamma \mid y;\varepsilon;e \mid \mathsf{A} \vdash \boxed{\&r\,\omega\ p :: \&r\,\omega\,\mu\,\delta} \dashv \Gamma \mid \mathsf{A}}$$

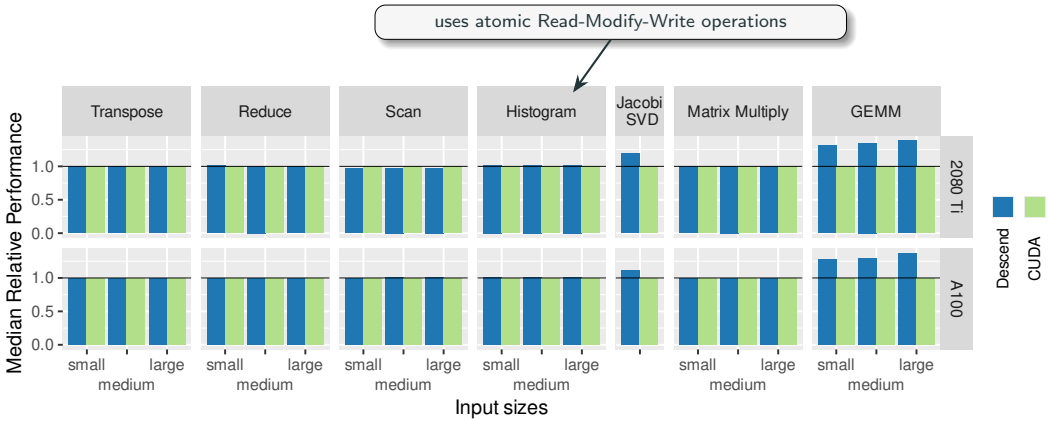
T-Let

$$\frac{\Delta;\Gamma \vdash^{\ell} \delta' \rightsquigarrow \delta \dashv \Gamma'', (\mathcal{F}) \qquad \forall r \in \text{free-regions}(\delta).\ \Gamma''(\mathcal{F}) \vdash r\ \mathbf{nrb} \qquad \Delta;\Gamma \vdash \delta : \mathsf{dty} \qquad \Delta;\Gamma \mid y;\varepsilon;e \mid \mathsf{A} \vdash \boxed{t :: \delta'} \dashv \Gamma' \mid \mathsf{A}'}{\Delta;\Gamma \mid y;\varepsilon;e \mid \mathsf{A} \vdash \boxed{\mathbf{let}\ x : \delta = t : \mathbf{unit}} \dashv \Gamma'', (\mathcal{F}, x : \delta) \mid \mathsf{A}'}$$

Evaluation

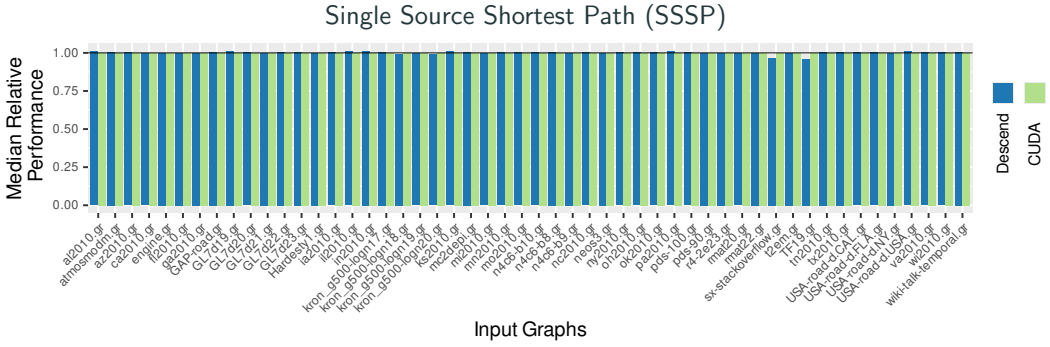


Programs generated from Descend achieve competitive performance to handwritten CUDA.



Programs generated from Descend achieve competitive performance to handwritten CUDA.

unsafe: Interfacing with CUDA



unsafe code provides a quick method of interfacing with existing CUDA code

Conclusion

- *Descend* assists programmers in managing CPU and GPU memory and enforcing previously implicit assumptions
- *Descend* extends the concept of ownership to the thread hierarchy and uses views to ensure safe memory accesses on the GPU
- The evaluation shows that *Descend* is expressive enough to write programs that achieve performance on-par with handwritten CUDA

Website for Descend: `https://descend-lang.org`

bastian.koepcke@uni-muenster.de