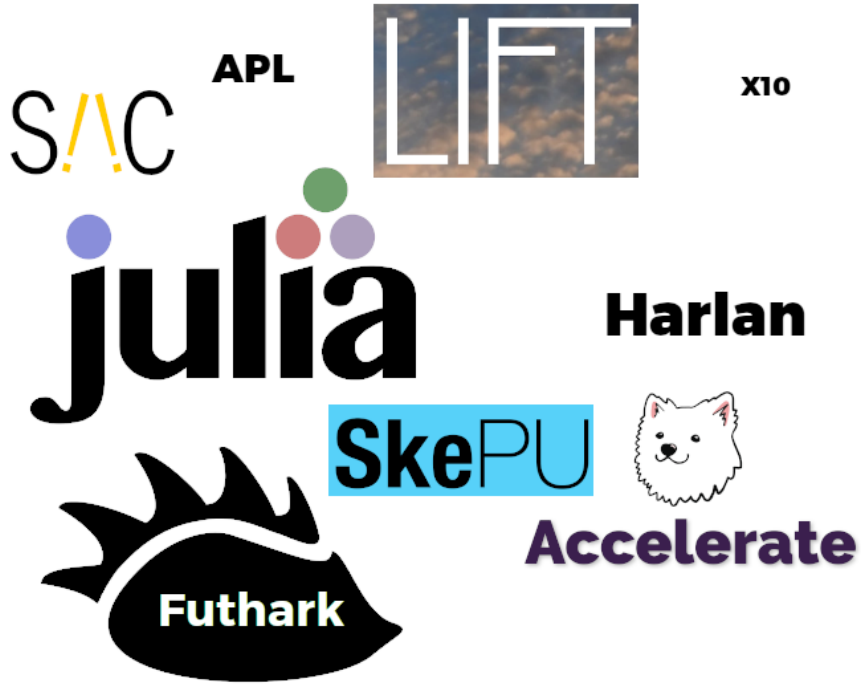
Bounds Checking on GPU

Troels Henriksen

Troels Henriksen
University of Copenhagen
athas@sigkill.dk

9th of July 2020 at HLPP

We have many nice high-level languages for GPUs!



- **Provide all you're used to from high level languages!**
- Abstracts low-level hardware details, memory management, race conditions...
- Safe and productive!

Well, almost safe...

A shared limitation

None of these languages provide bounds checking on GPUs (although some do when generating non-GPU code)

Well, almost safe...

A shared limitation

None of these languages provide bounds checking on GPUs (although some do when generating non-GPU code)

This is perfectly understandable!

Well, almost safe...

A shared limitation

None of these languages provide bounds checking on GPUs (although some do when generating non-GPU code)

This is perfectly understandable!

- Bounds checking only helps **incorrect programs**
- **Slows down** correct programs
- Not obvious how to implement on GPU

Compilers are not very motivated to implement an optional feature that helps wrong programs and wrong good programs.

Well, almost safe...

A shared limitation

None of these languages provide bounds checking on GPUs (although some do when generating non-GPU code)

This is perfectly understandable!

- Bounds checking only helps **incorrect programs**
- **Slows down** correct programs
- Not obvious how to implement on GPU

Compilers are not very motivated to implement an optional feature that helps wrong programs and wrong good programs.

(Also: many high-level languages mainly use *bulk operations* such as `map` and `reduce`, which are index-correct by construction.)

But bounds checking *is* important

Programs tend to start out incorrect!

But bounds checking *is* important

Programs tend to start out incorrect!

Bounds checking must be...

But bounds checking *is* important

Programs tend to start out incorrect!

Bounds checking must be...

- Fast enough to be enabled in production

But bounds checking *is* important

Programs tend to start out incorrect!

Bounds checking must be...

Fast enough to be enabled in production

Informative enough that programmers can fix the problem

But bounds checking *is* important

Programs tend to start out incorrect!

Bounds checking must be...

Fast	enough to be enabled in production
Informative	enough that programmers can fix the problem
Correct	with no false positive or negatives, and no actually unsafe operations such as deadlocks or accessing invalid memory

But bounds checking *is* important

Programs tend to start out incorrect!

Bounds checking must be...

Fast	enough to be enabled in production
Informative	enough that programmers can fix the problem
Correct	with no false positive or negatives, and no actually unsafe operations such as deadlocks or accessing invalid memory

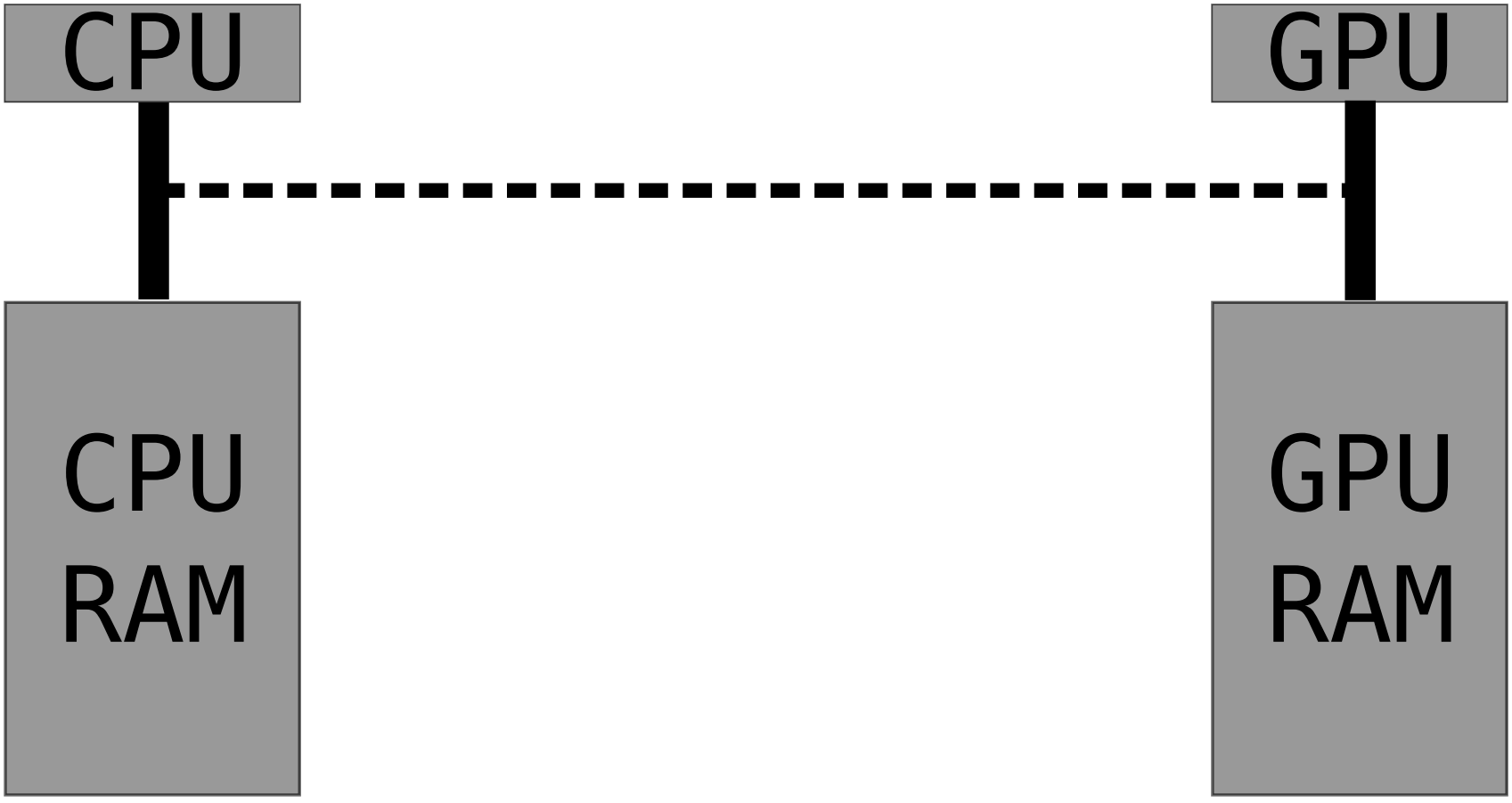
I will present an implementation technique for adding bounds checking to high-level parallel languages that target GPUs.

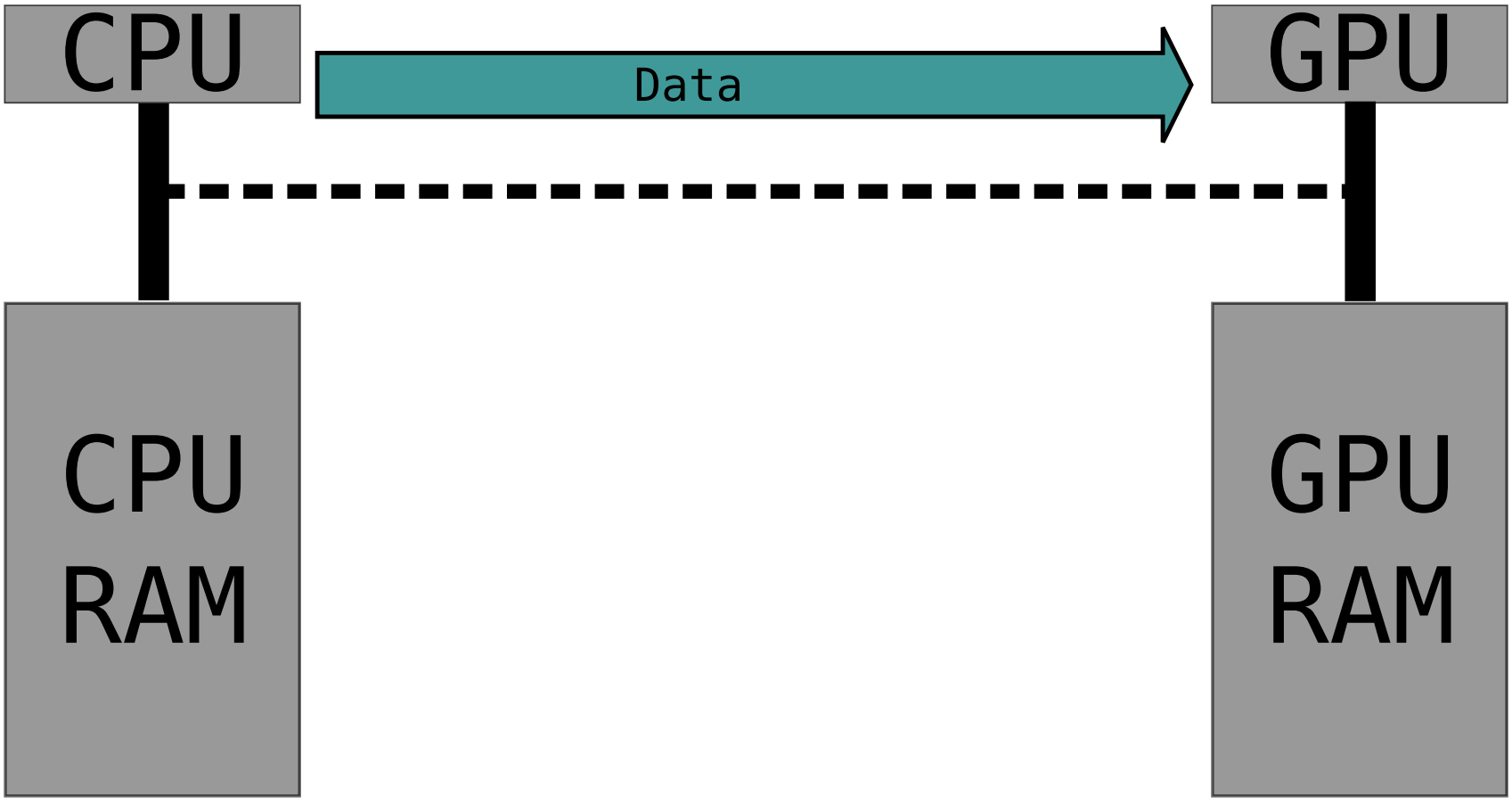
So what's the problem?

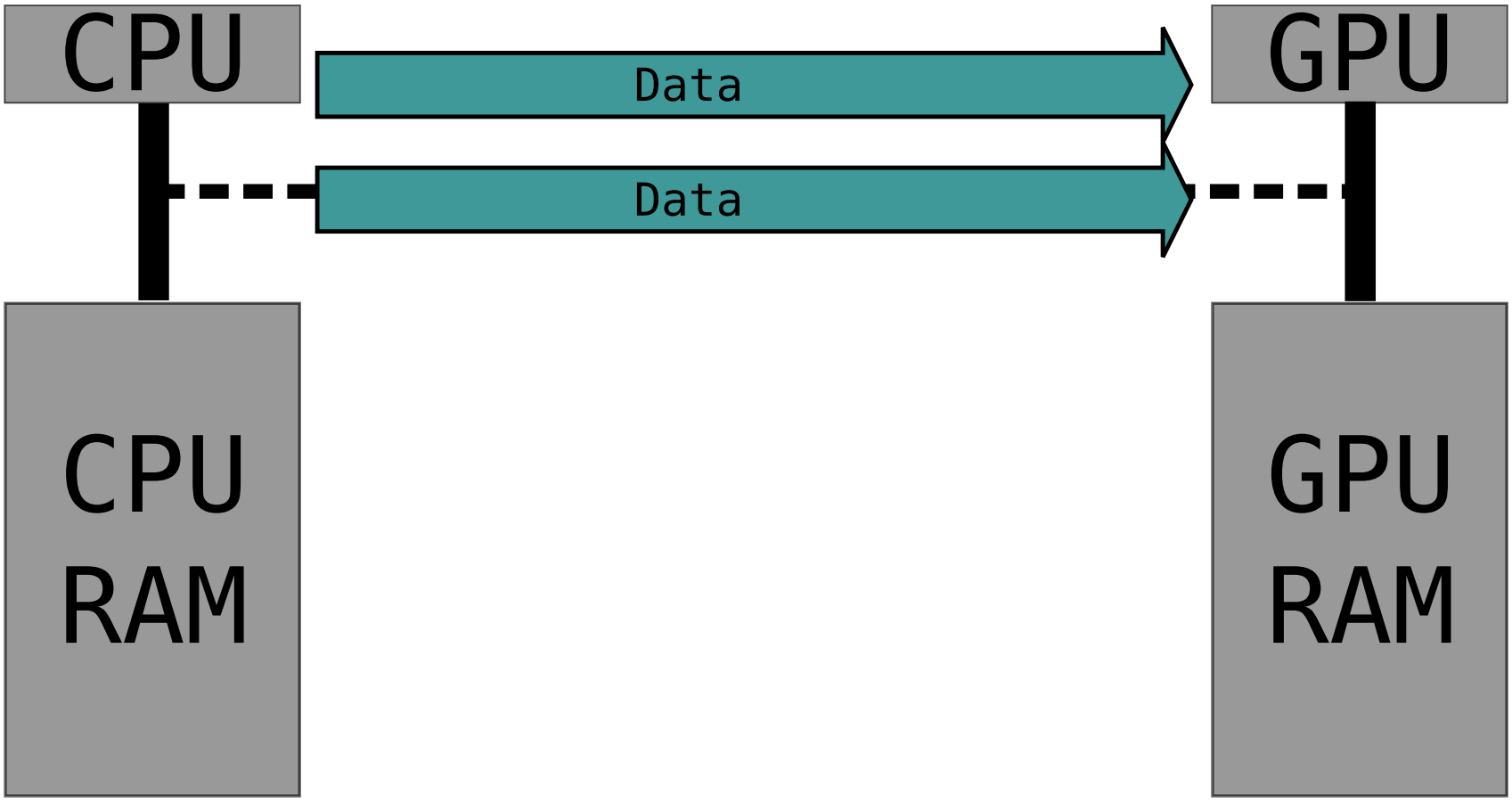
On a CPU, bounds checking is trivial.

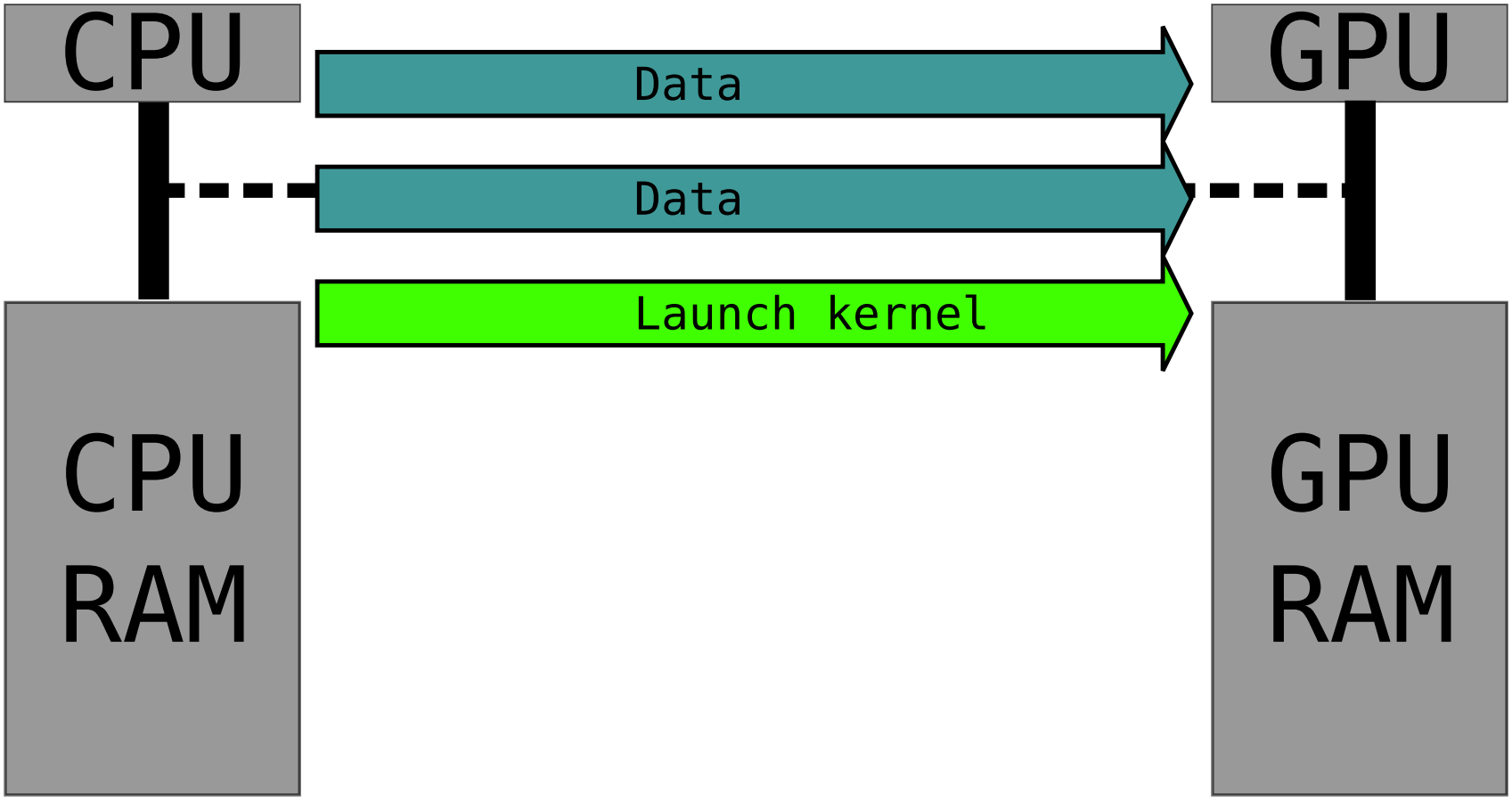
```
if (i < 0 || i > n) {
  // throw exception

  // OR print error message

  // OR terminate program

  // OR do anything else
}
```

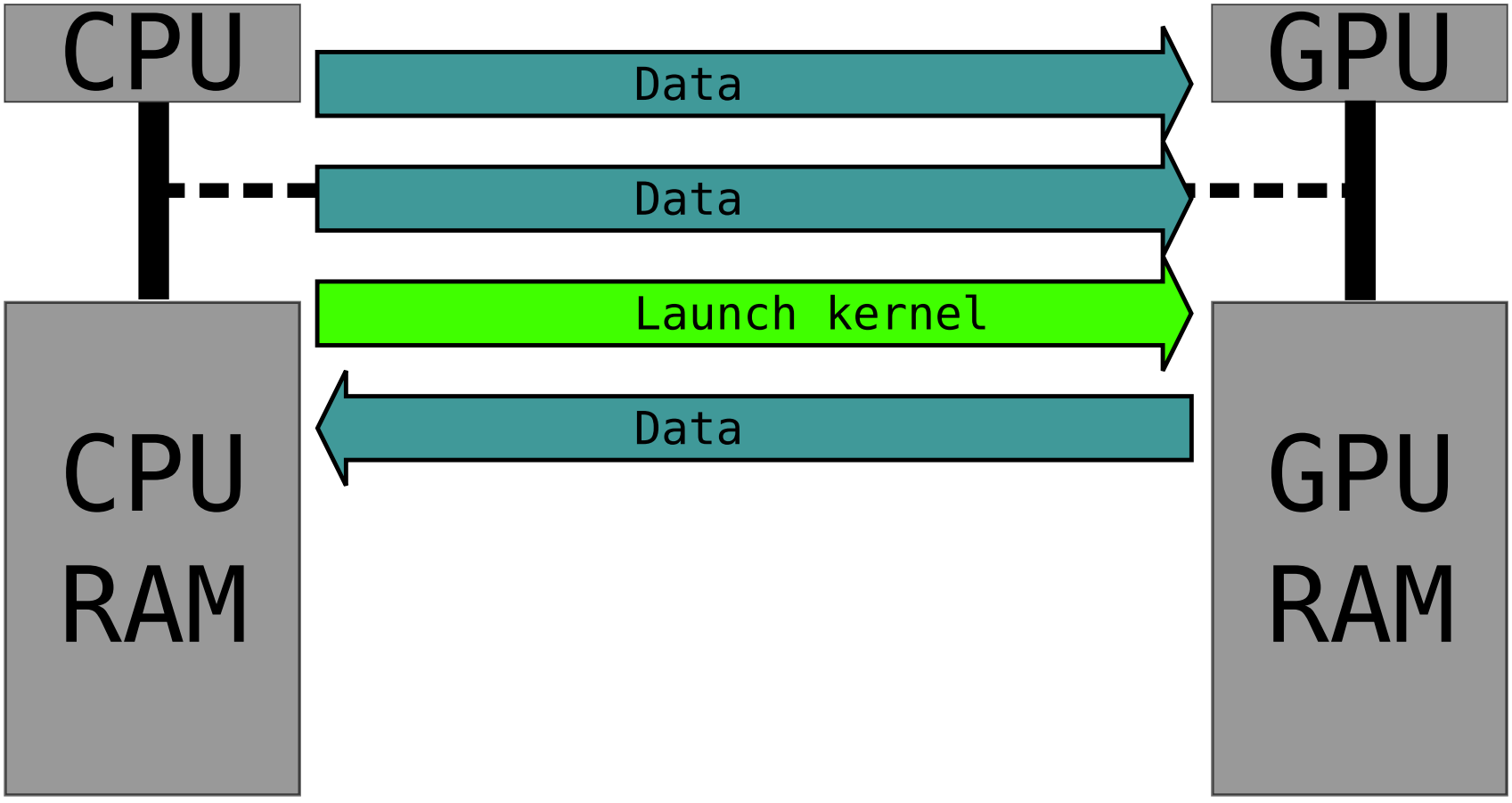
Let's look at the conceptual GPU execution model to see why it's more difficult there.

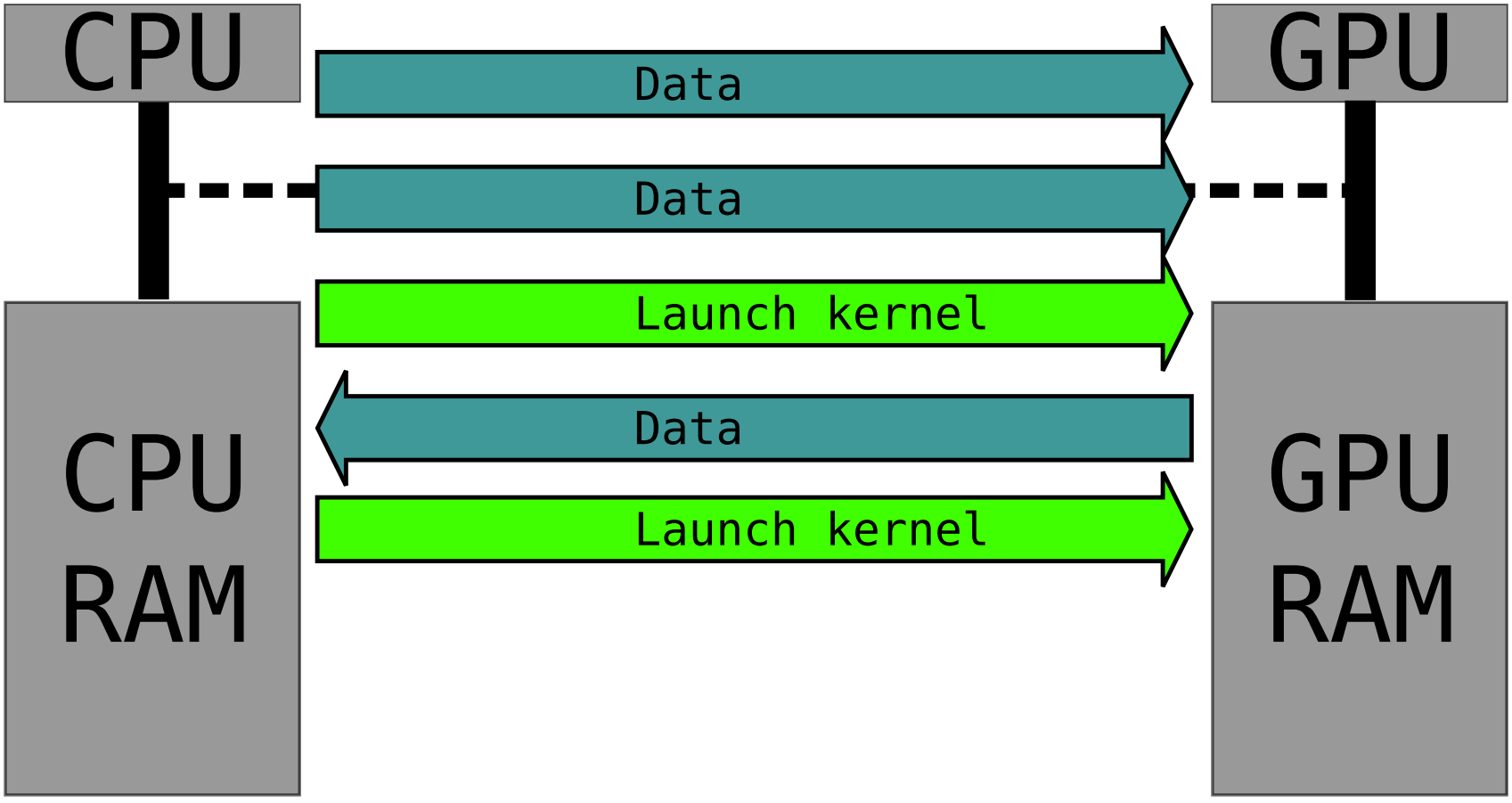

CPU
GPU
CPU
RAM
GPU
RAM


CPU
Data
GPU
CPU
RAM
GPU
RAM


CPU
GPU
Data
Data
CPU
RAM
GPU
RAM


CPU
GPU
Data
Data
Launch kernel
CPU
RAM
GPU
RAM


CPU
GPU
Data
Data
Launch kernel
Data
CPU
RAM
GPU
RAM


CPU
GPU
Data
Data
Launch kernel
Data
Launch kernel
CPU
RAM
GPU
RAM

The basic problem

The GPU is a co-processor, and the CPU can neither directly control it, not be controlled by it.

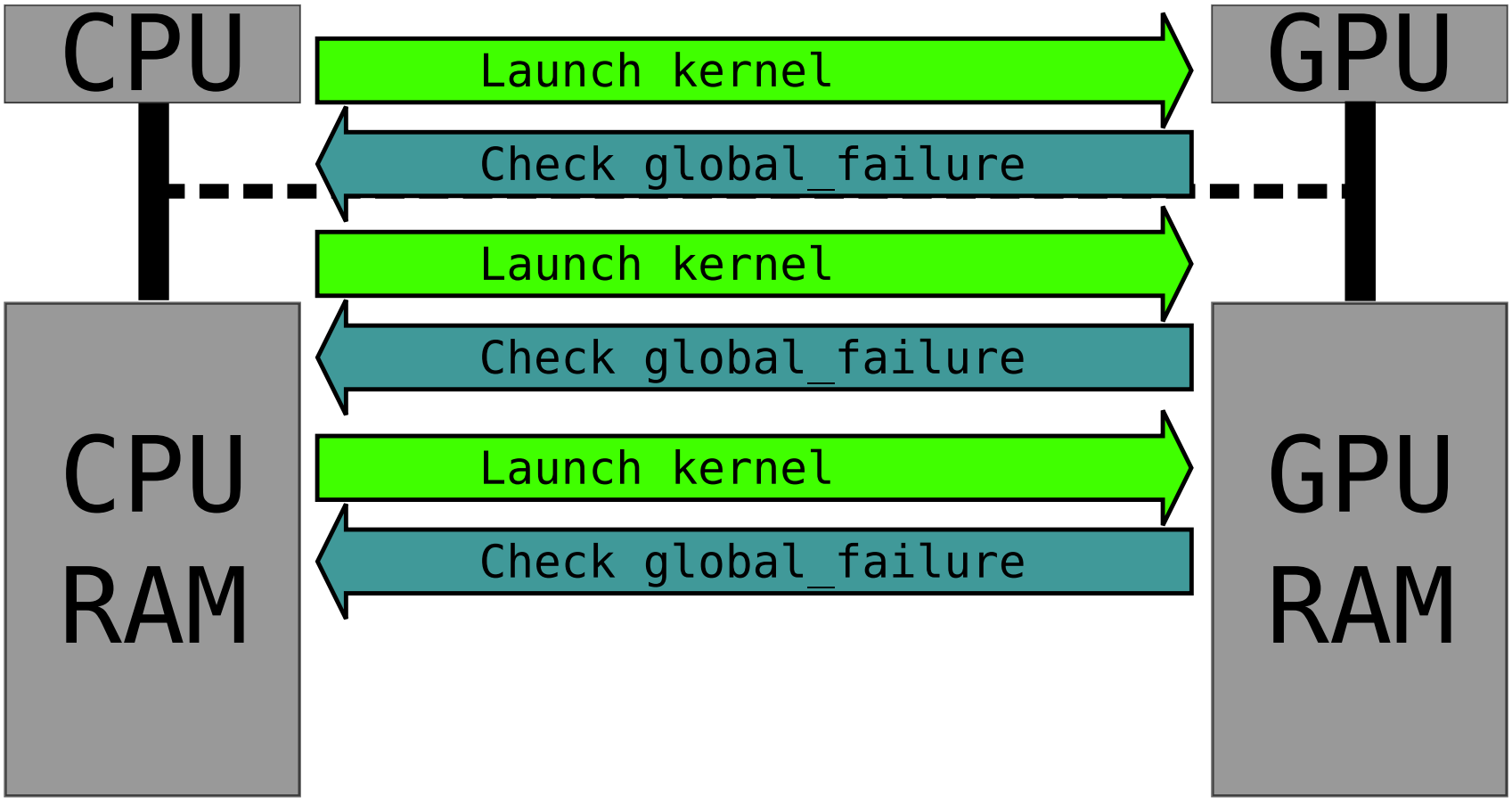
The basic problem

The GPU is a co-processor, and the CPU can neither directly control it, not be controlled by it.

Solution idea

Whenever a bounds check on the GPU fails, write to a known location in GPU memory, and copy the value at that location back to the CPU after every GPU kernel.

```
if (i < 0 || i >= n) {
  *global_failure = 1;
  return;  // Terminate GPU thread
}
```



Surely reading a single byte after every kernel (which usually processes tens of thousands of elements) must be fast?

Some experimental evaluation

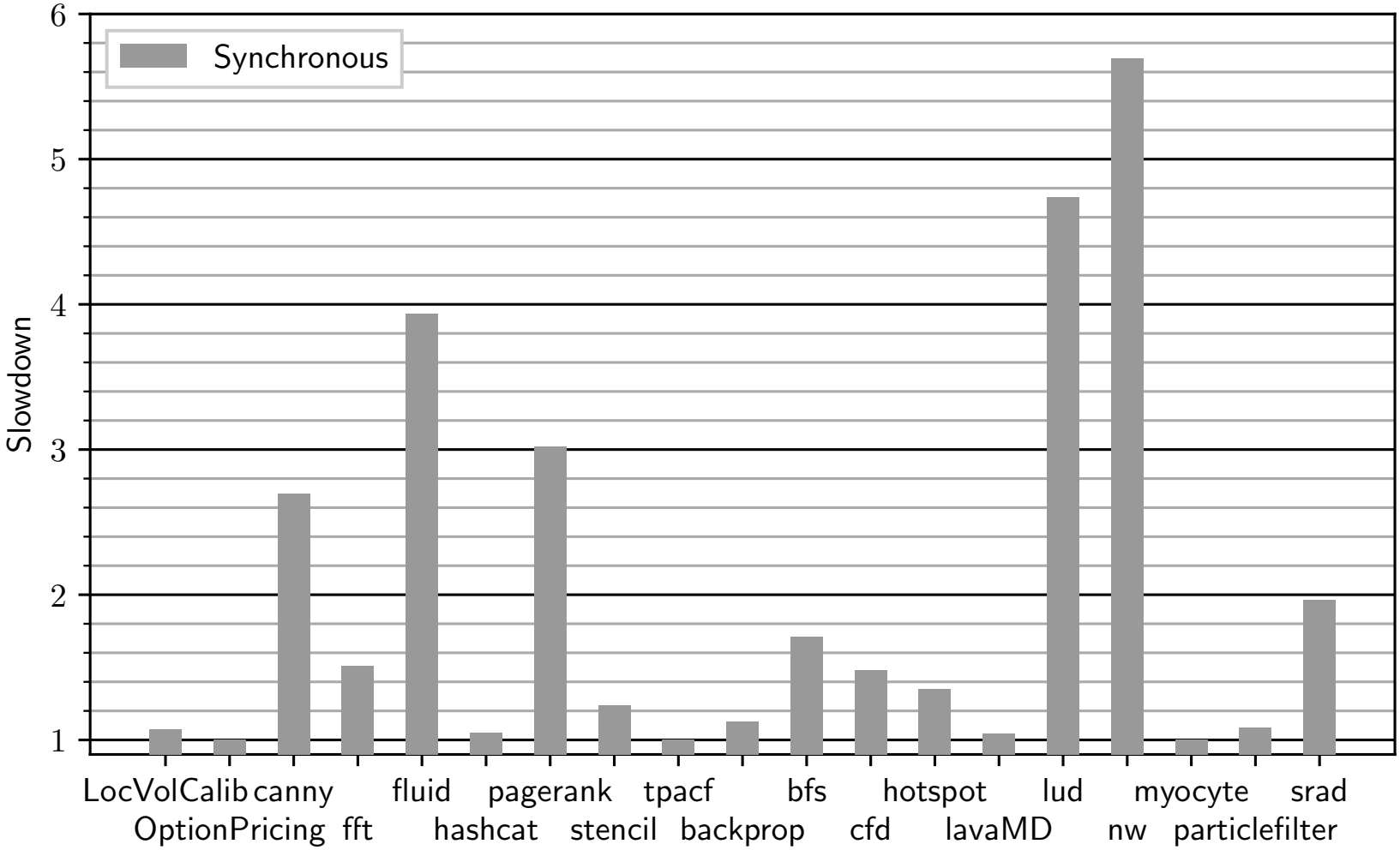
Experimental setup

- NVIDIA RTX 2080 Ti GPU (and an AMD GPU in the paper)
- Bounds checking implemented in the compiler for *Futhark*
- Bounds checking overhead measured on the Futhark benchmark suite

But note that the technique is not Futhark-specific in any way—only the concrete implementation

- The Futhark benchmark suite contains 41 programs, of which 19 require bounds-checking in GPU code
- We will show overhead of bounds checking on these

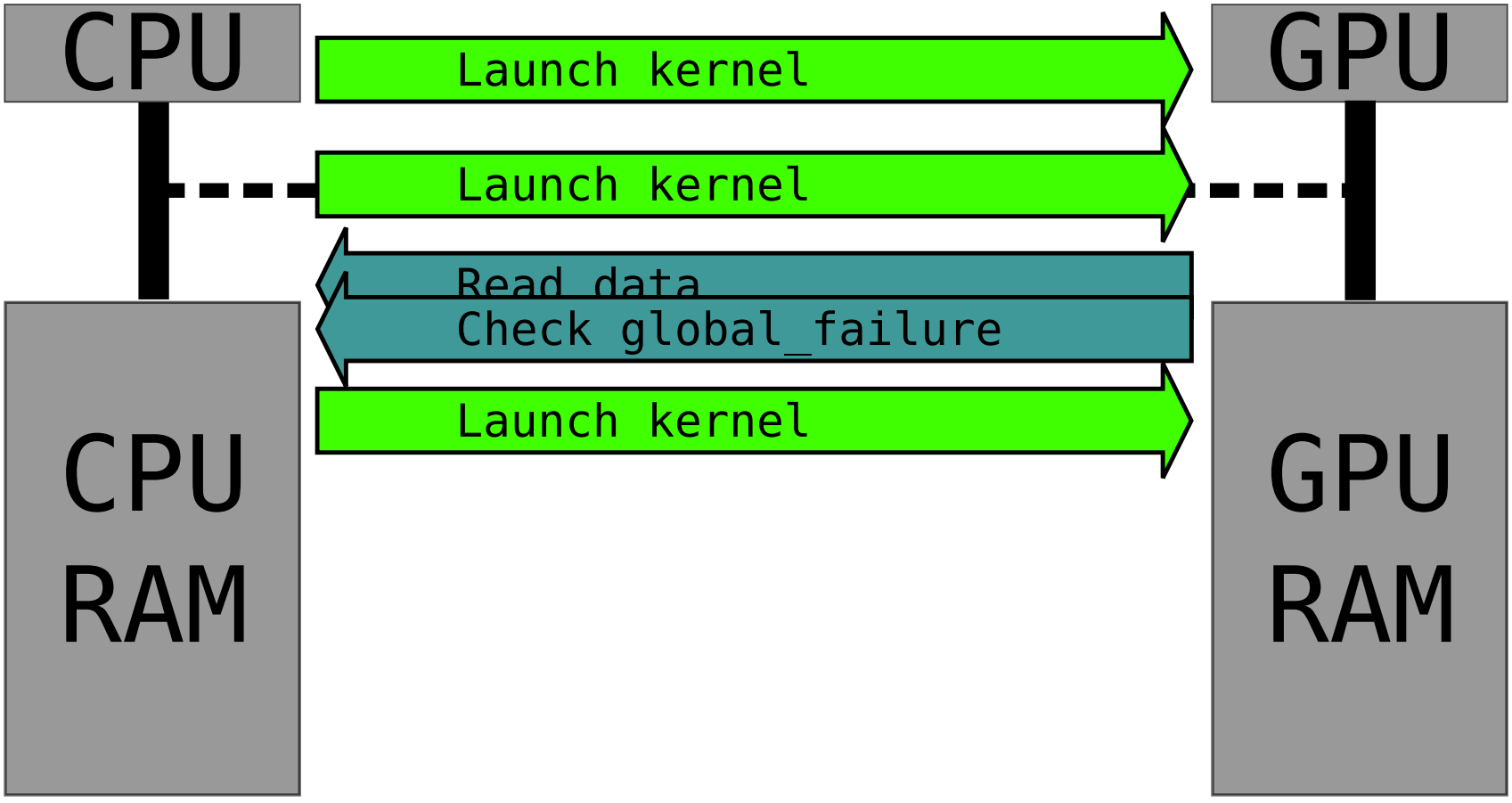
Overhead of fully synchronous bounds checking



- Geomean overhead of 1.66 ×
- Max overhead exceeding 5 ×
- **Not viable**

The problem

- GPUs (viewed from the CPU) are **asynchronous machines** where we provide a *queue* of work
- Constantly halting the GPU to inspect a boolean is wasteful
- We should only check `global_failure` when we need to get data from the GPU anyway for other purposes


CPU
GPU
Launch kernel
Launch kernel
Read data
Check global_failure
Launch kernel
CPU
RAM
GPU
RAM

What if kernel $i + 1$ is *only* safe to run if kernel i ran succesfully?

How do we efficiently ensure that failures in one kernel prevent later kernels in the queue from running?

What if kernel $i + 1$ is *only* safe to run if kernel i ran succesfully?

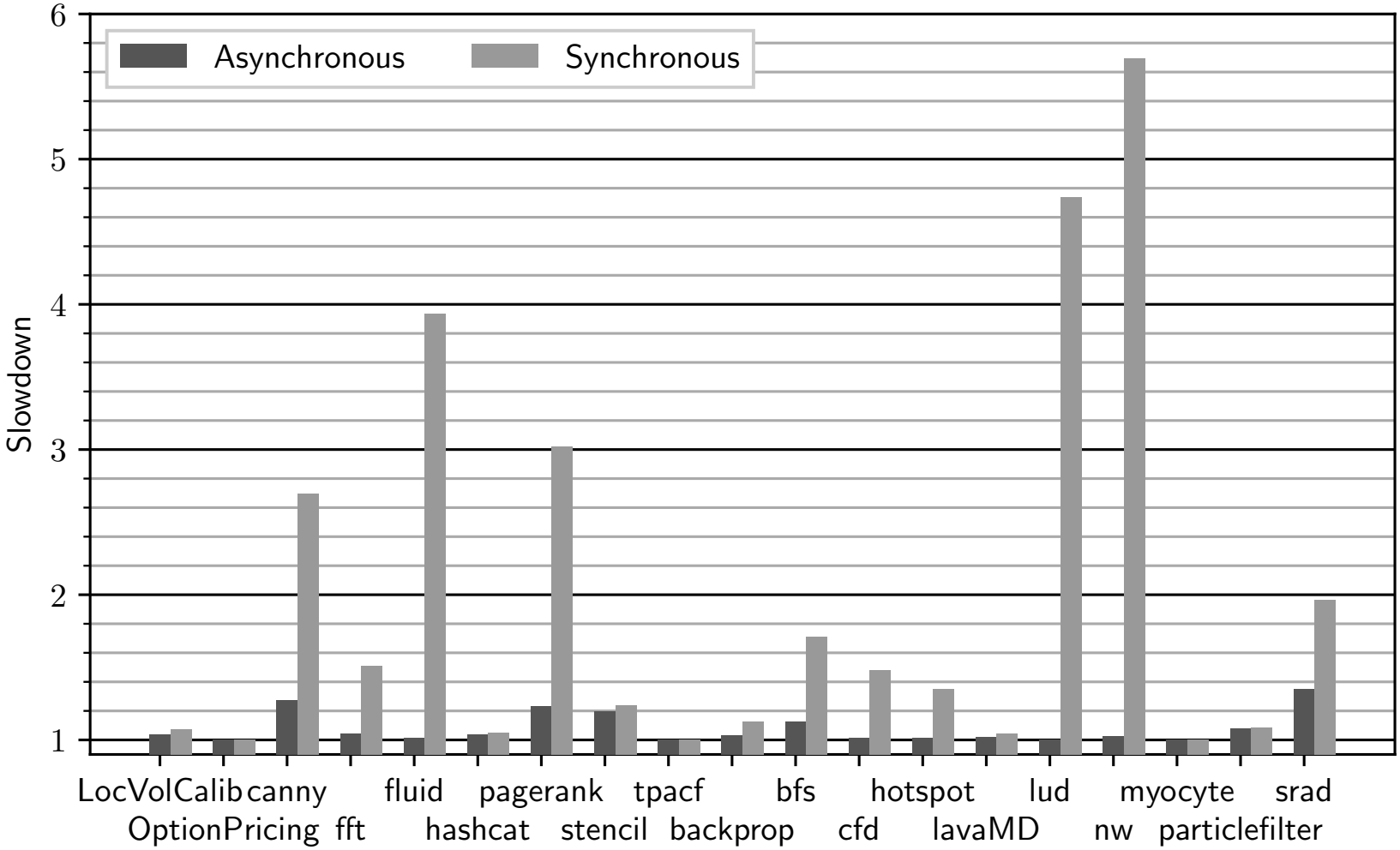
How do we efficiently ensure that failures in one kernel prevent later kernels in the queue from running?

Answer: put a check into each GPU kernel itself!

```
// Prelude added to every GPU kernel
if (*global_failure) {
  return;
}
```

- To the GPU, the work queue is processed as normal
- But after the first failure, all kernels just exit immediately

Overhead of asynchronous bounds checking



- Geomean overhead of 1.07 × (vs 1.66 ×)
- Max overhead about 1.4 × (over 5 × before)
- **Viable**

What about error messages?

Error messages with 1 bit of information is not good enough.

But we cannot print from a GPU...

What about error messages?

Error messages with 1 bit of information is not good enough.

But we cannot print from a GPU...

Solution: at compile time, produce a table in CPU RAM mapping each distinct *failure point* to a `printf()` style format string.

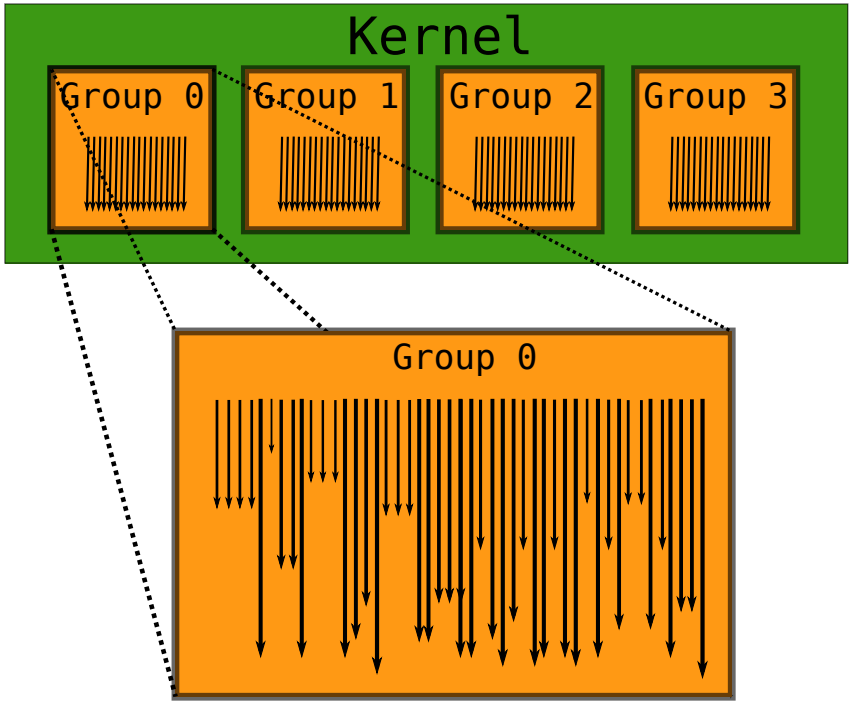
```
const char *errors[] =
  { "line 10: index %d invalid for array of size %d",
    "line 17: division by zero",
    "line 42: index %d invalid for array of size (%d,%d)" }
```

On failure, write arguments to GPU array.

```
if (i < 0 || i >= n) {
  if (atomic_cmpxchg(global_failure, -1, 0) < 0) {
    global_failure_args[0] = i;
    global_failure_args[1] = n;
  }
  return; // stop thread
}
```

Now `global_failure` is an index into an array of error messages.

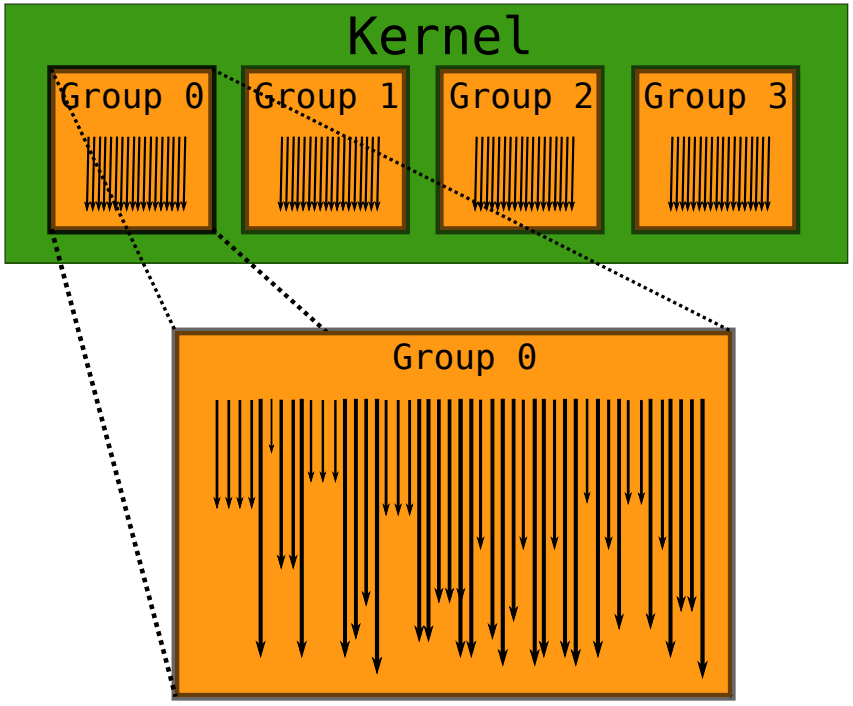
Problem: threads are not actually isolated



- Threads within a group can communicate through memory
- Synchronisation through **barriers**, which block the thread until all threads in the group have reached the barrier

Barrier rule: if *one* thread in a group reaches `barrier()`, then *all* threads must eventually reach it.

Problem: threads are not actually isolated



- Threads within a group can communicate through memory
- Synchronisation through **barriers**, which block the thread until all threads in the group have reached the barrier

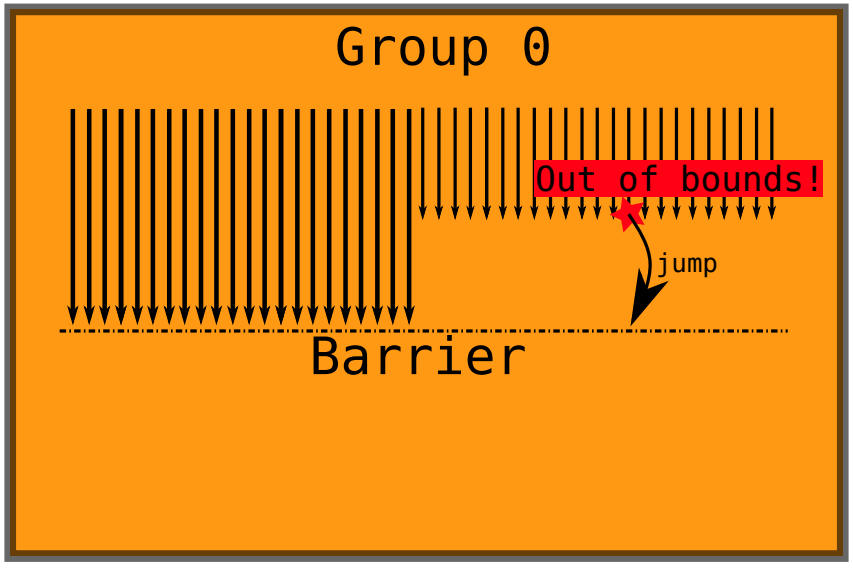
Barrier rule: if *one* thread in a group reaches `barrier()`, then *all* threads must eventually reach it.

Consequence: a GPU thread cannot unilaterally just terminate itself—and prevailing GPU APIs provide *no* sound way for abnormally terminating a running kernel!

```
1  kernel sum(int *global_failure,
2             int n, int *js,
3             int m, int *vs,
4             ...) {
5    local int sums[GROUP_SIZE];
6    int i, gtid, ltid, k, acc; ...
7    for (i = gtid; i < n; i += k) {
8      int j = js[i];
9      if (j < 0 || j >= m) {
10       *global_failure = 1;
11       return; // What if another thread has reached the barrier?
12     }
13     acc += vs[j];
14   }
15
16   sums[ltid] = acc;
17
18
19   barrier();
20
21
22
23
24
25   // Perform parallel
26   // reduction of sums...
27 }
```

Avoiding barrier divergence

Instead of `return`ing from the kernel, a failing thread should jump to the next `barrier()`.

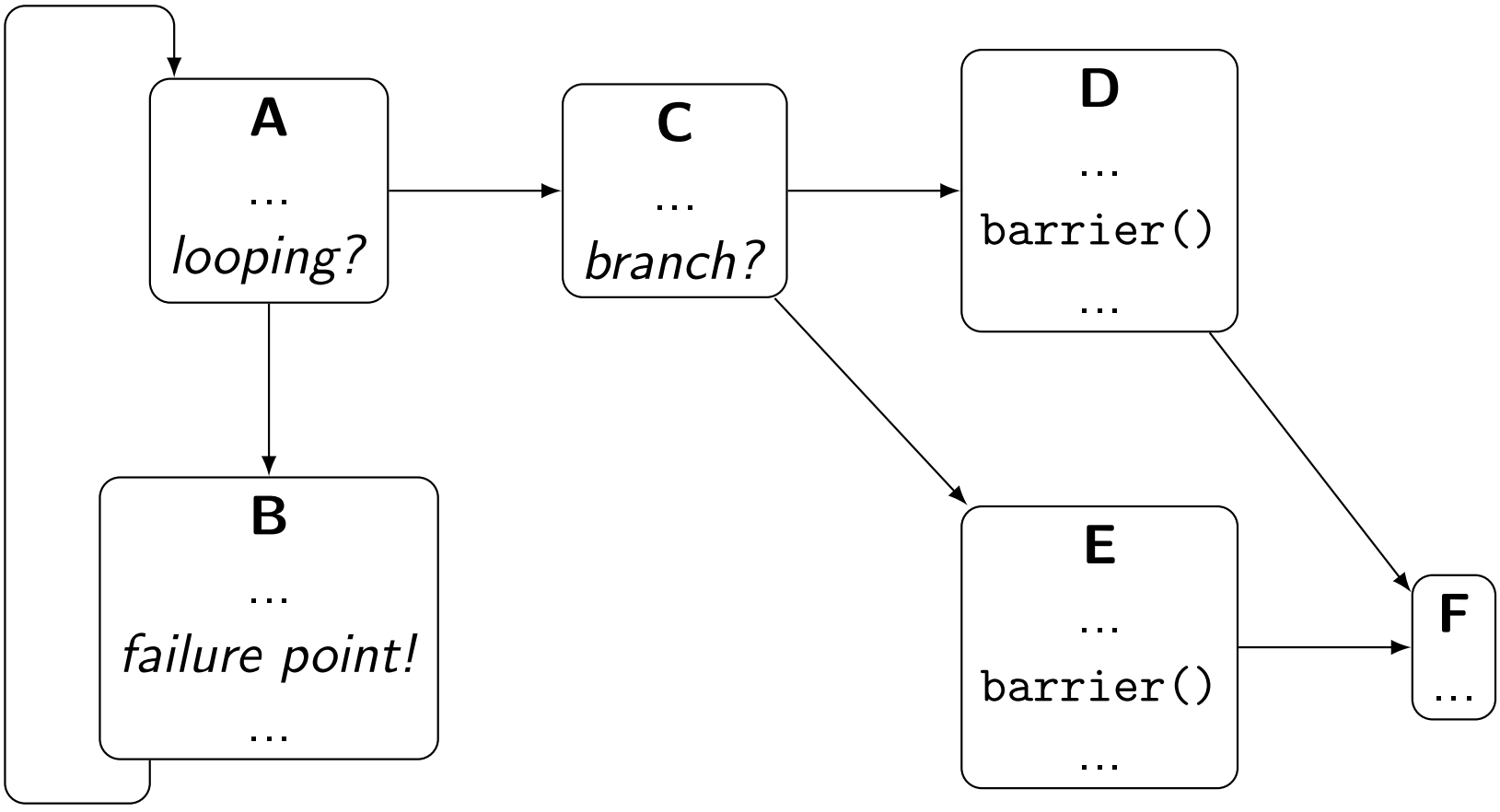


After the barrier, we then check whether any threads have failed.

We call this a *synchronisation point*.

```
1  kernel sum(int *global_failure,
2             int n, int *js,
3             int m, int *vs,
4             ...) {
5    local int sums[GROUP_SIZE];
6    int i, gtid, ltid, k, acc; ...
7    for (i=gtid; i<n; i+=k) {
8      int j = js[i];
9      if (j < 0 || j >= m) {
10       *global_failure = 1;
11       goto sync;
12     }
13     acc += vs[j];
14   }
15
16   sums[ltid] = acc;
17
18   sync:
19   barrier();
20   if (*global_failure != -1) {
21     return;
22   }
23   barrier();
24
25   // Perform parallel
26   // reduction of sums...
27 }
```

Viewed as a control-flow graph



- Compiler must insert a synchronisation point in node **C**
- Straightforward when compiling a high-level language where programmer does not control thread communication

Correct error checking prelude

This prelude is wrong:

```
// Prelude added to every GPU kernel
if (*global_failure >= 0) {
  return;
}
```

Thread i may race ahead and fail (setting `global_failure`) before thread j has a chance to run.

Correct error checking prelude

This prelude is wrong:

```
// Prelude added to every GPU kernel
if (*global_failure >= 0) {
  return;
}
```

Thread *i* may race ahead and fail (setting `global_failure`) before thread *j* has a chance to run.

Fixed:

```
if (*global_failure >= 0) {
  return;
}
barrier();
```

Stops *all threads* or *no threads*.

More optimisations (details in paper)

- Do not protect *error-tolerant* kernels (e.g. copies, transpositions)

More optimisations (details in paper)

- Do not protect *error-tolerant* kernels (e.g. copies, transpositions)
- Simpler prelude for kernels without failure point

```
if (*global_failure >= 0) { return; }
```

More optimisations (details in paper)

- Do not protect *error-tolerant* kernels (e.g. copies, transpositions)
- Simpler prelude for kernels without failure point

```
if (*global_failure >= 0) { return; }
```

- Cache failure information in GPU *local memory*

```
local int local_failure;
```

  Failure points write to both `global_failure` and `local_failure`, but synchronisation points read only from `local_failure`

More optimisations (details in paper)

- Do not protect *error-tolerant* kernels (e.g. copies, transpositions)
- Simpler prelude for kernels without failure point

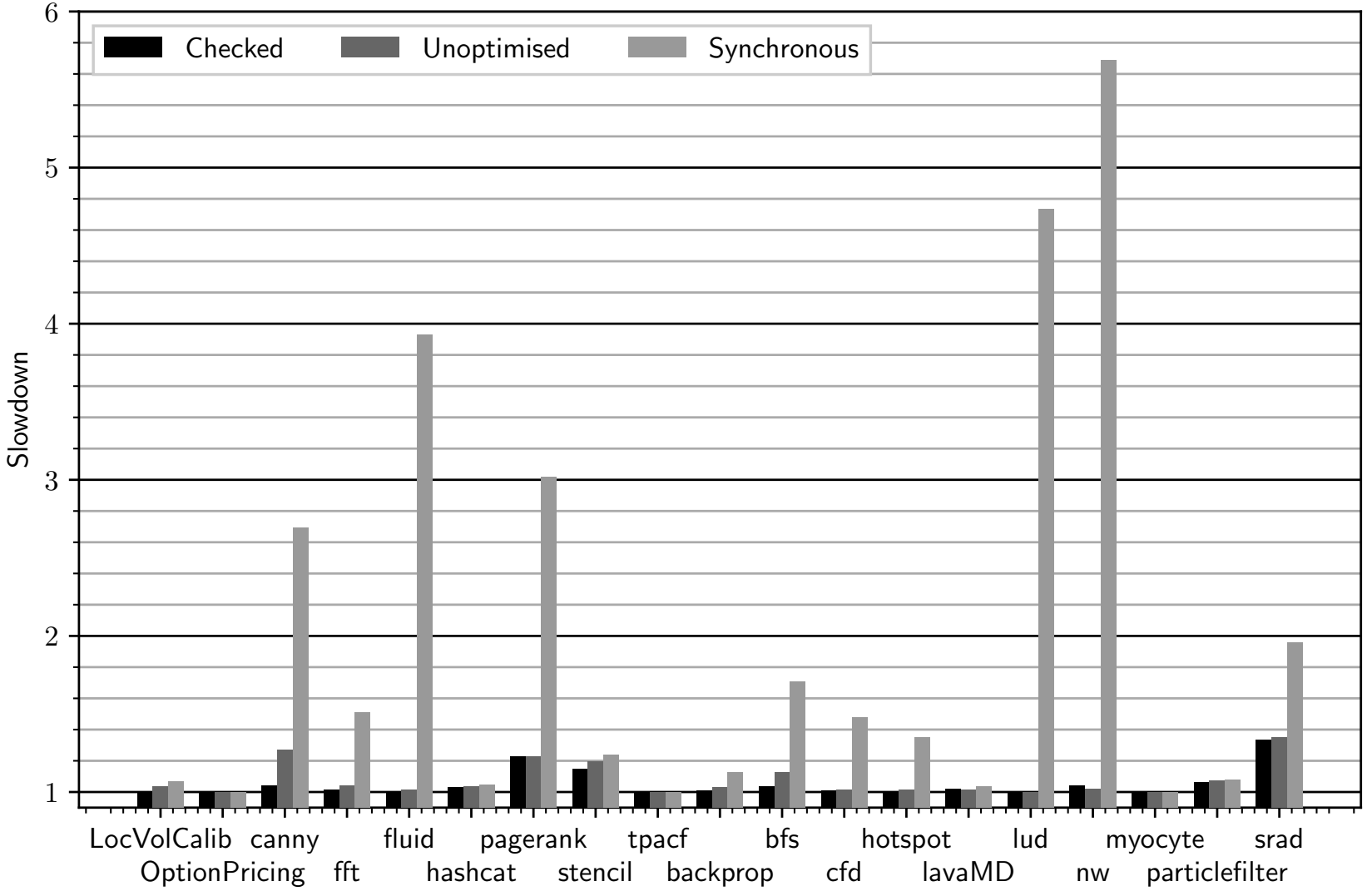
```
if (*global_failure >= 0) { return; }
```

- Cache failure information in GPU *local memory*

```
local int local_failure;
```

Failure points write to both `global_failure` and `local_failure`, but synchronisation points read only from `local_failure`

These optimisations have relatively small impact



- Geomean overhead of 1.04 × (1.07 × without optimisations)
- Max overhead about 1.4 ×
- **Fast enough to be default behaviour**

Conclusions

- Bounds checking on GPU can be implemented non-invasively in the code generator for a high-level language
- Overhead low enough to do by default
- Asynchronous checking is crucial for low overhead
- Care must be taken to work around API limitations
- Other optimisations are nice to have, but not crucial

`https://futhark-lang.org`