Formal Analysis Techniques for GPU kernels

Nathan Chong (nyc04@imperial.ac.uk)

Leap Conference, 22 May 2013

Social Processes and Proofs of Theorems and Programs

Richard A. De Millo
Georgia Institute of Technology

Richard J. Lipton and Alan J. Perlis
Yale University

"It is argued that formal verifications of programs, no matter how obtained, will not play the same key role in the development of computer science and software engineering as proofs do in mathematics"

no matter how obtained, will not play the same key role in the development of computer science and software ... the absence of continuity, the inevitability of change, and the complexity of specification of significantly many real programs make the formal verification process difficult to justify and manage. It is felt that ease of formal verification should not dominate program language design.

Key Words and Phrases: formal mathematics, mathematical proofs, program verification, program specification

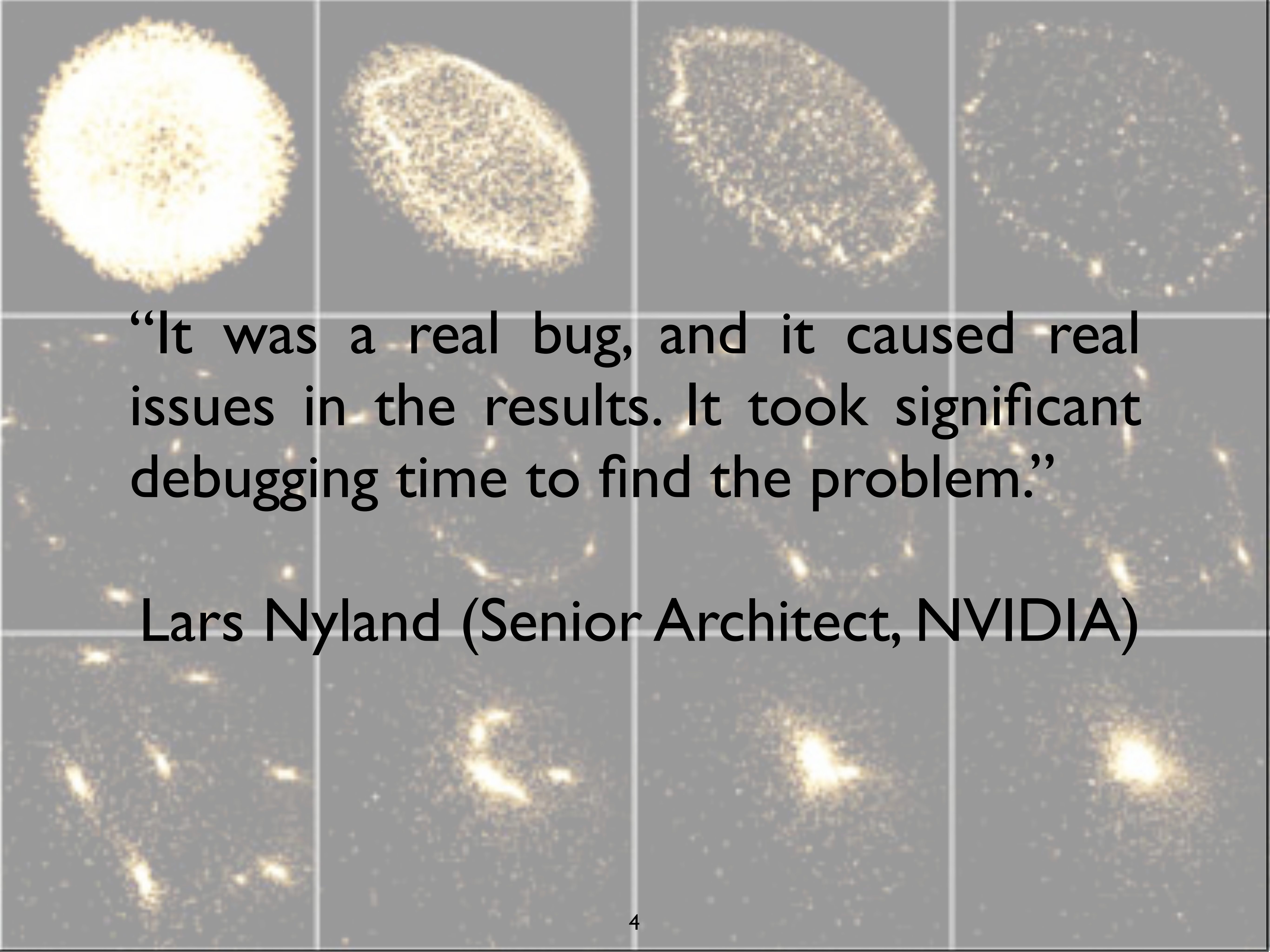
CR Categories: 2.10, 4.6, 5.24

... are proposing to give a precise definition of logical correctness which is to be the same as my vague intuitive feeling for logical ... the same? ... intuition is the final authority.

J. Barkley Rosser

Many people have argued that computer programming should strive to become more like mathematics. Maybe so, but not in the way they seem to think. The aim of program verification, an attempt to make programming more mathematics-like, is to increase dramatically one's confidence in the correct functioning of a piece of software, and the device that verifiers use to achieve this goal is a long chain of formal, deductive logic. In mathematics, the aim is to increase one's con-

Verification as a powerful and practical complement to Testing



"It was a real bug, and it caused real issues in the results. It took significant debugging time to find the problem."

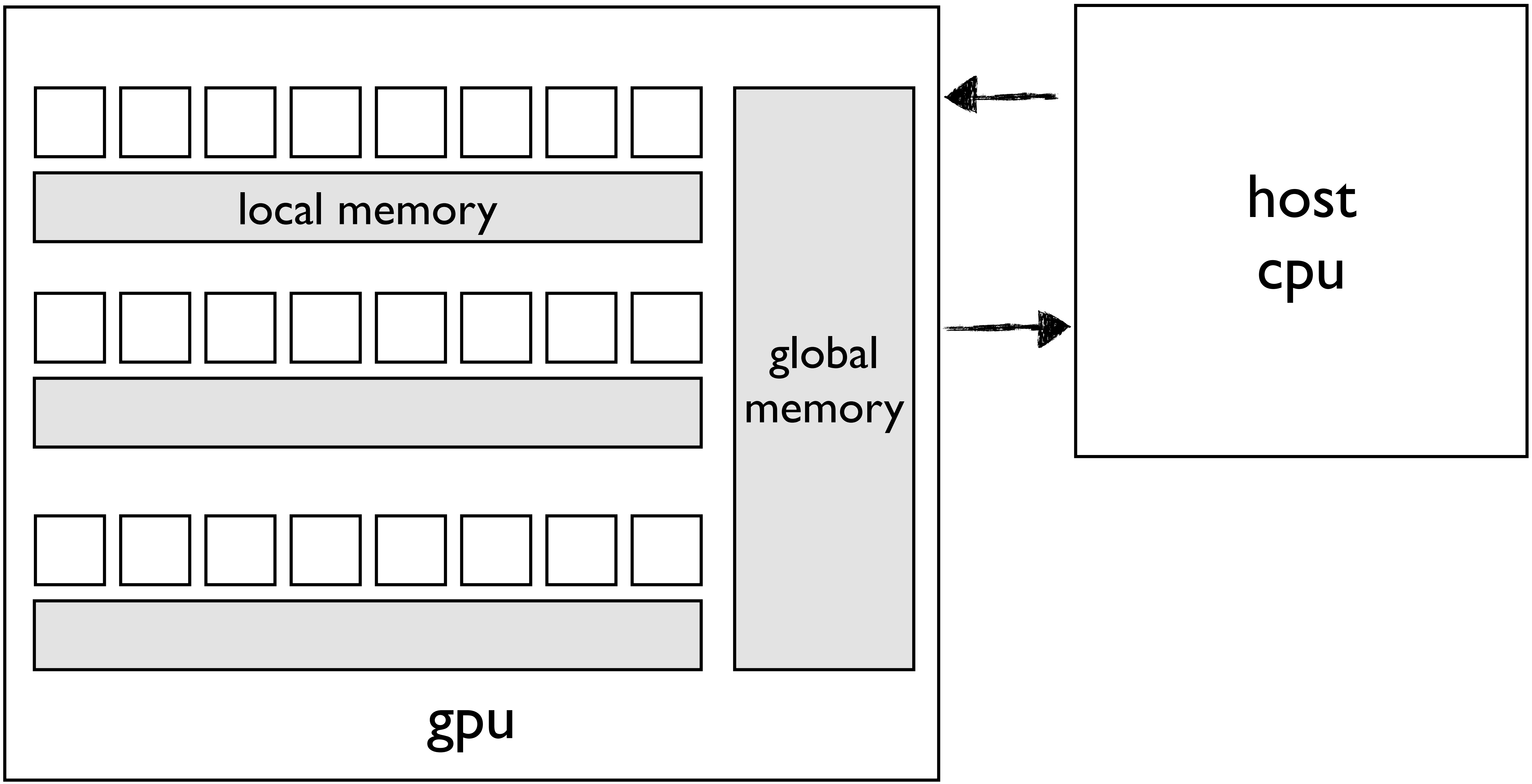
Lars Nyland (Senior Architect, NVIDIA)

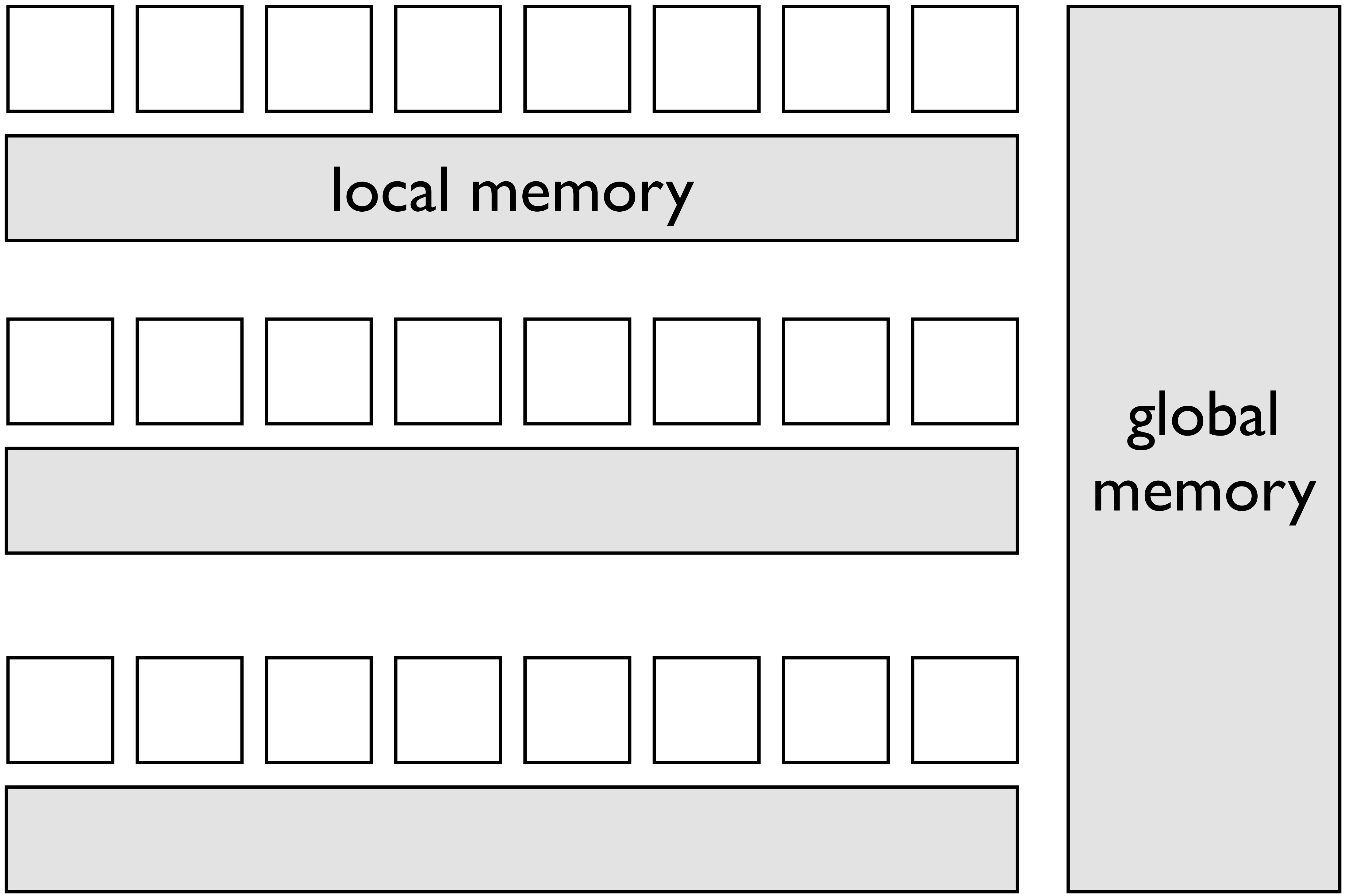
Schedule

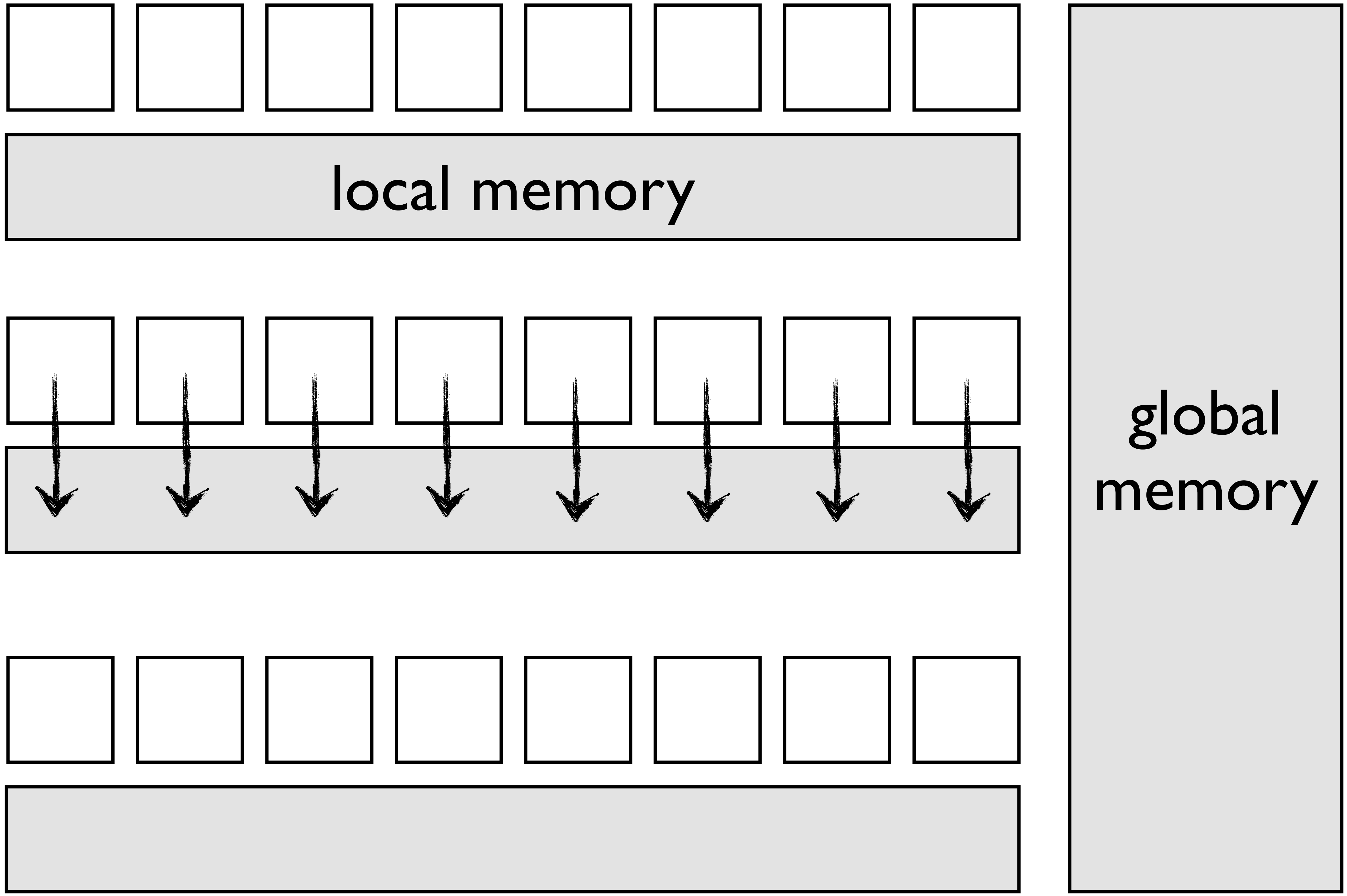
- Data races and Barrier Divergence
- Examples, Examples, Examples

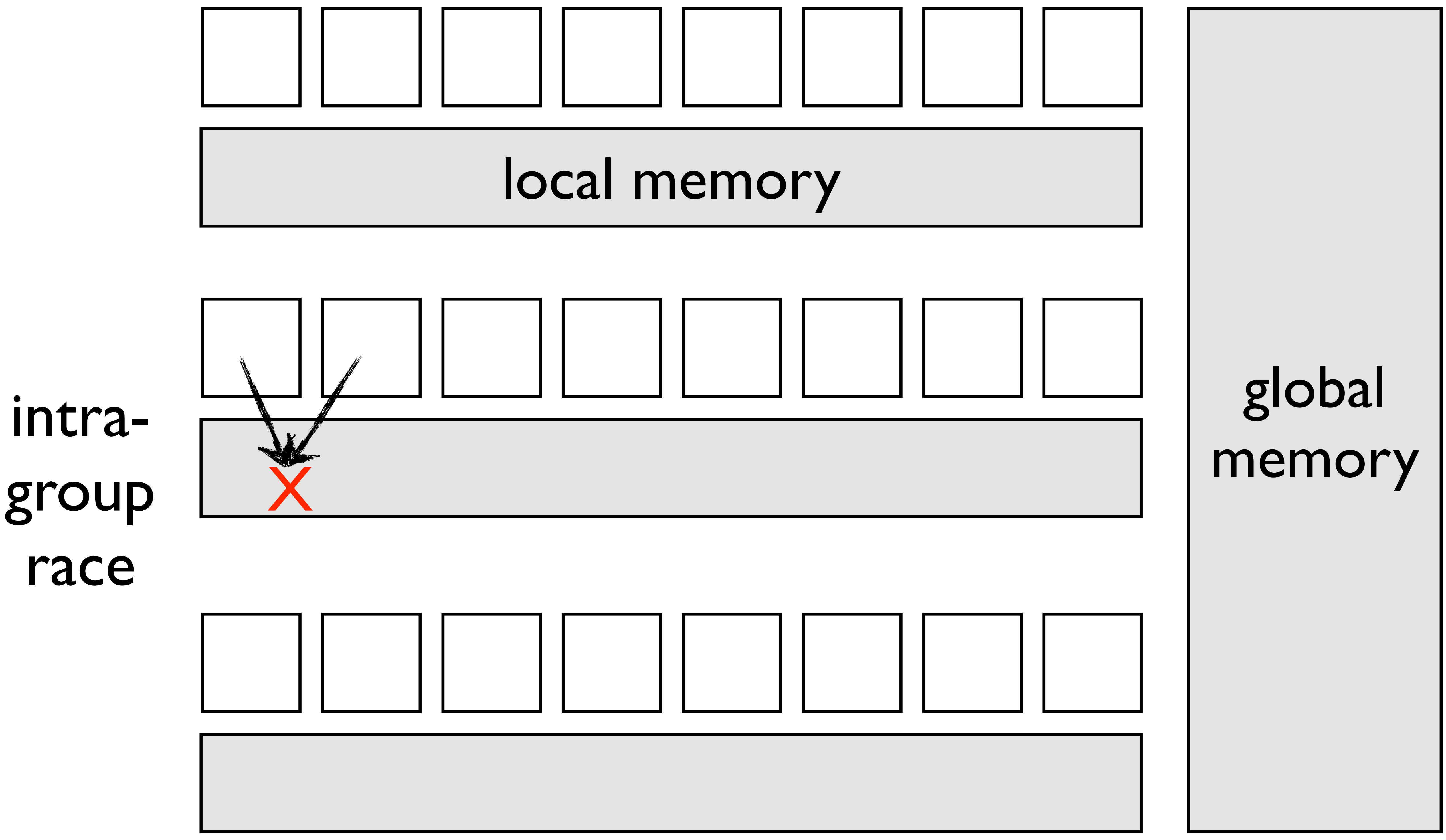
- Anatomy of GPUVerify
- Further Examples
- Close and Questions

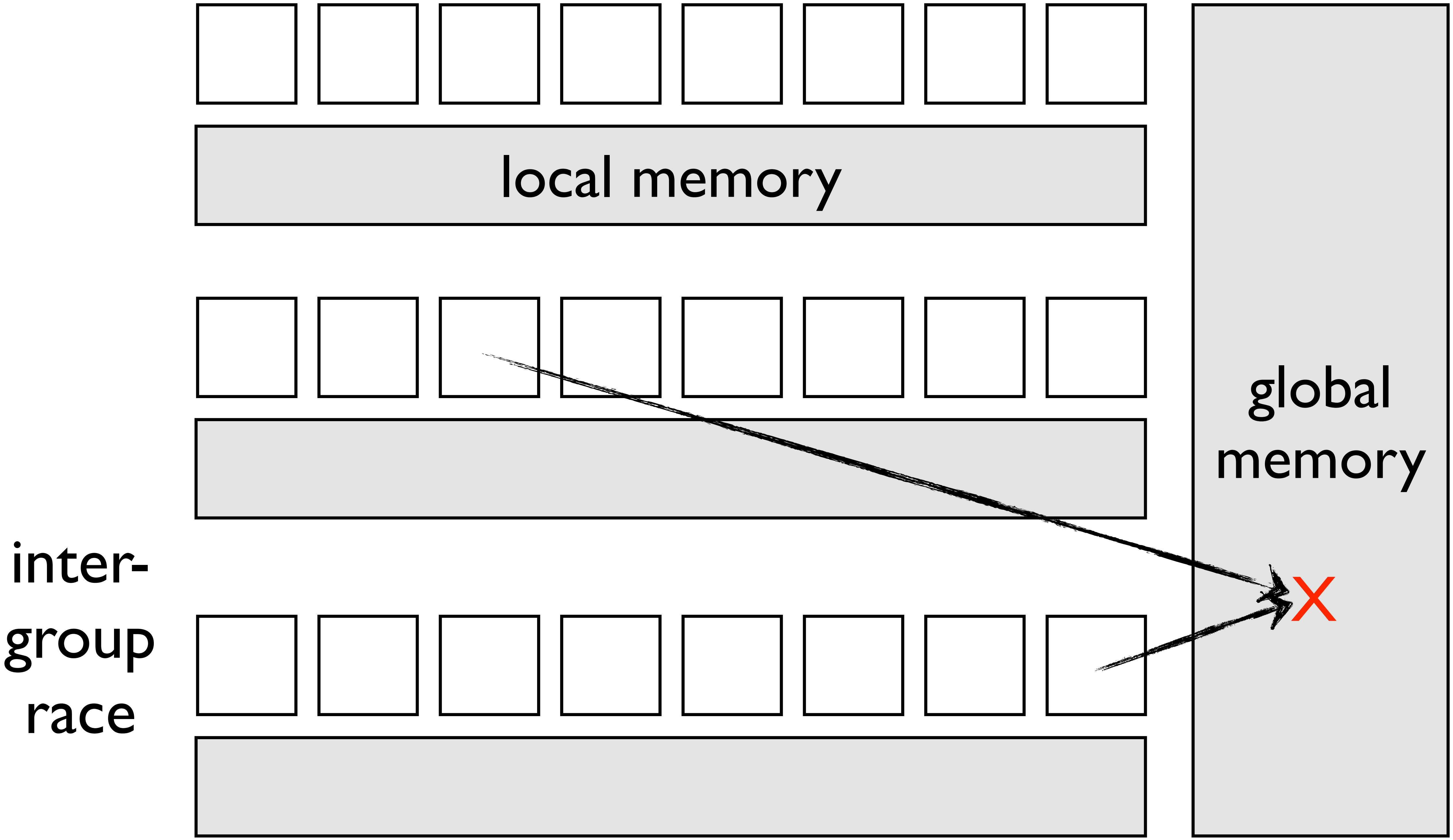
Data Races and Barrier Divergence


local memory
global
memory
gpu
host
cpu


local memory
global memory

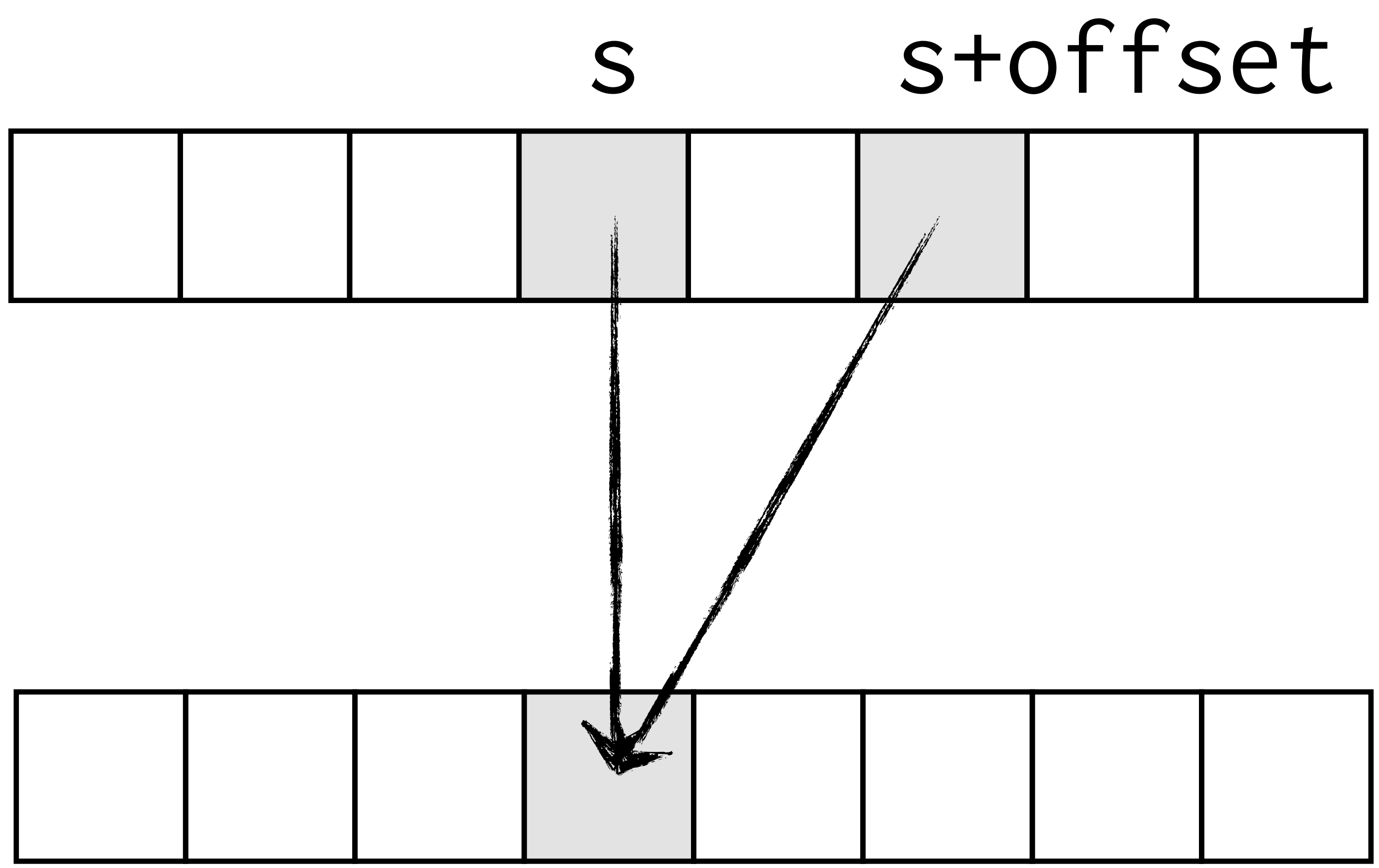

local memory
global memory


local memory
intra-
group
race
X
global
memory

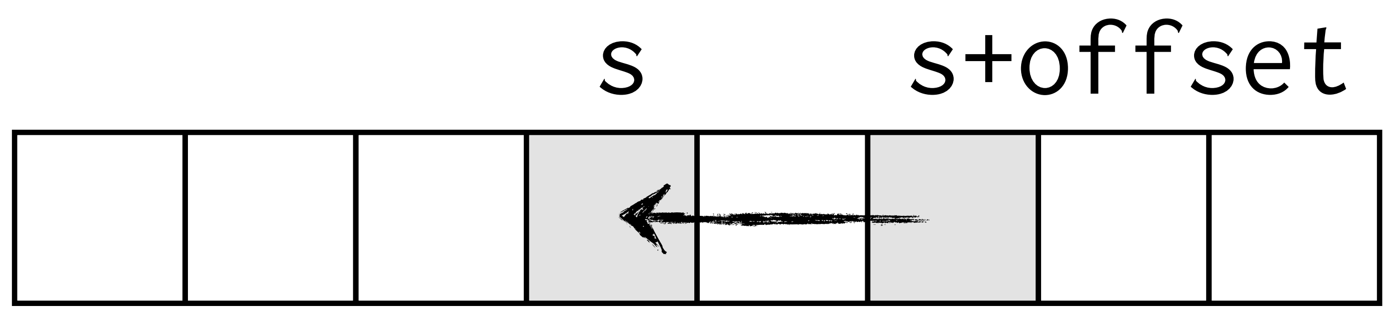

local memory
global memory
inter-group race
X

```
__kernel void
add_nbor(__local int *A, int offset) {
  int tid = get_local_id(0);
  A[tid] += A[tid+offset];
}
```

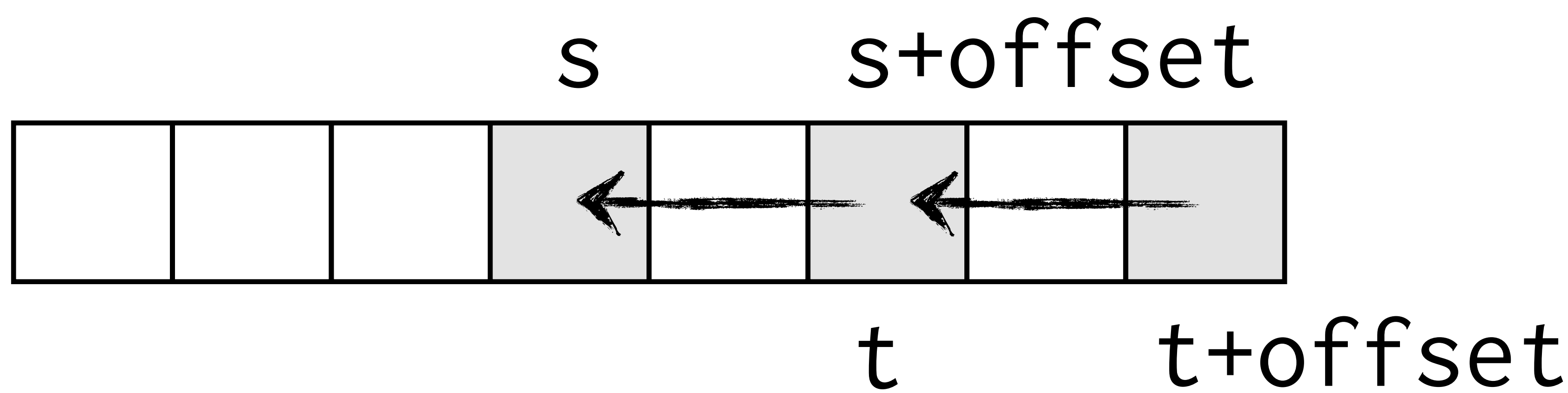
```
__kernel void
add_nbor(__local int *A, int offset) {
  int tid = get_local_id(0);
  A[tid] += A[tid+offset];
}
```



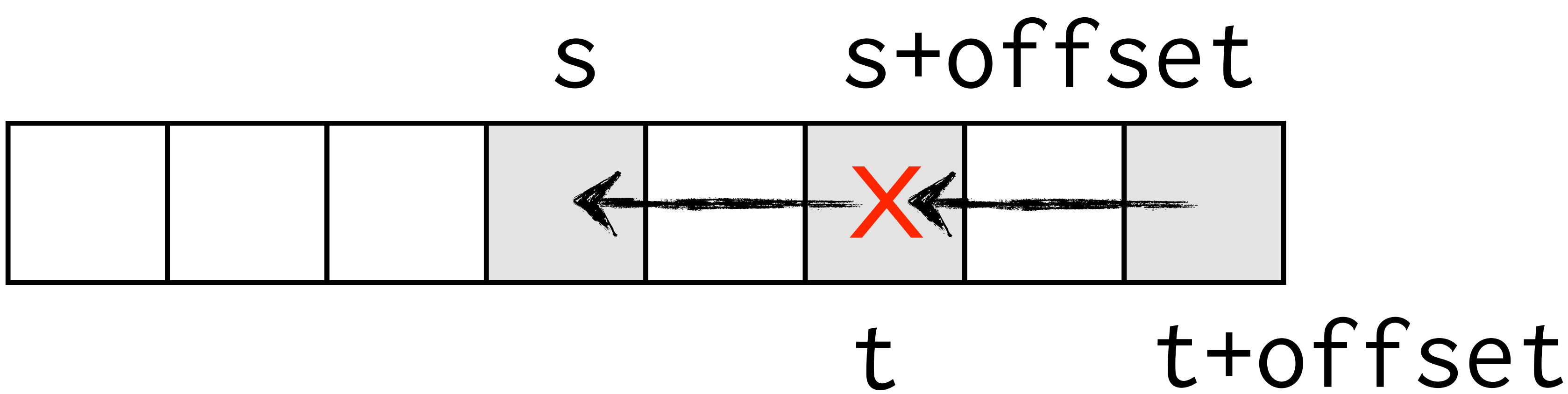
```
__kernel void
add_nbor(__local int *A, int offset) {
  int tid = get_local_id(0);
  A[tid] += A[tid+offset];
}
```



```
__kernel void
add_nbor(__local int *A, int offset) {
  int tid = get_local_id(0);
  A[tid] += A[tid+offset];
}
```



```
__kernel void
add_nbor(__local int *A, int offset) {
  int tid = get_local_id(0);
  A[tid] += A[tid+offset];
}
```



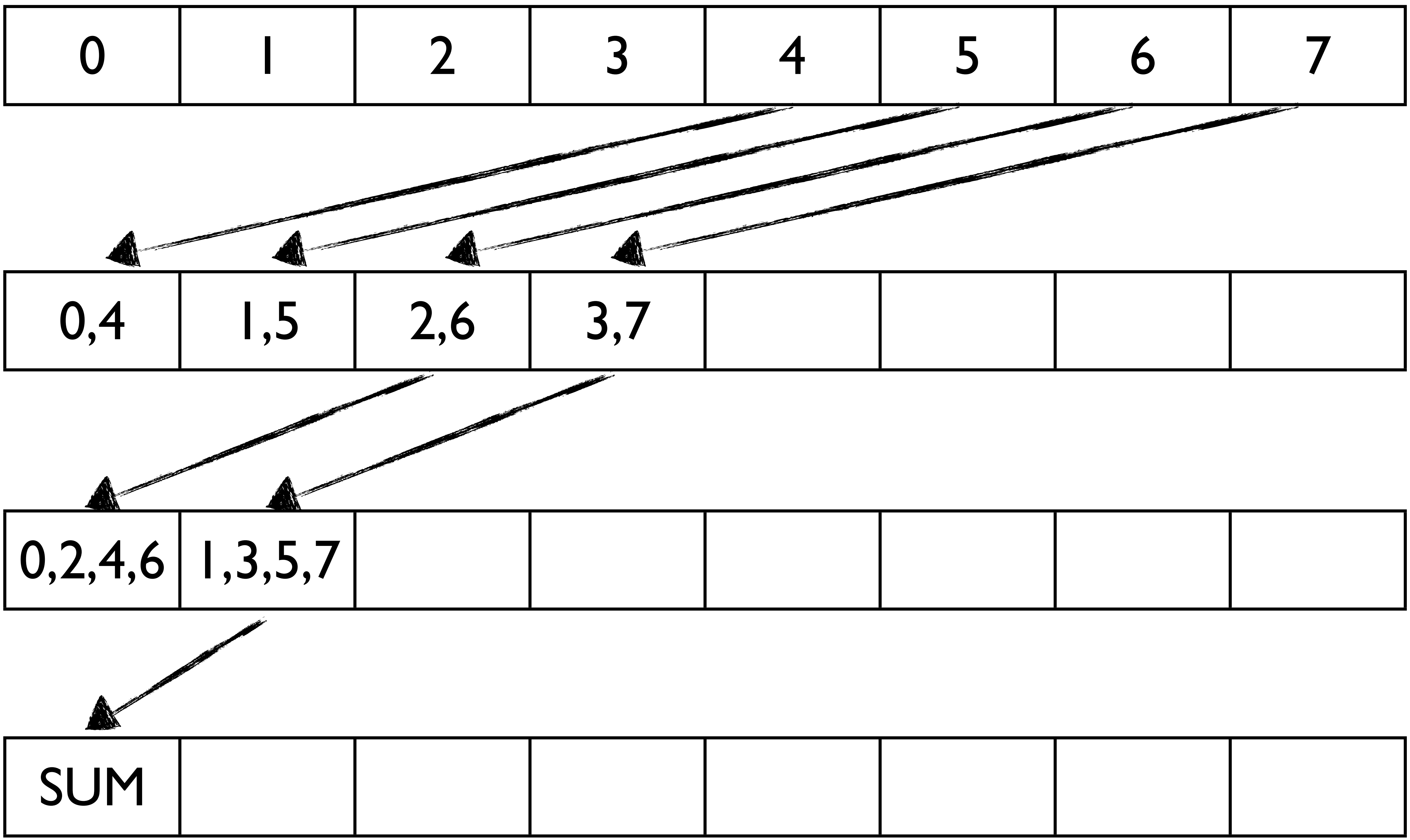
```
__kernel void diverge() {
  int tid = get_local_id(0);
  if (tid == 0) barrier();
  else barrier();
}
```

If barrier is inside a conditional statement, then all threads must enter the conditional if any thread enters the conditional statement and executes the barrier.

If barrier is inside a loop, all threads must execute the barrier for each iteration of the loop before any are allowed to continue execution beyond the barrier.

OpenCL Specification
(6.12.8 Synchronization Functions)

Reduction Demo



Examples, Examples, Examples

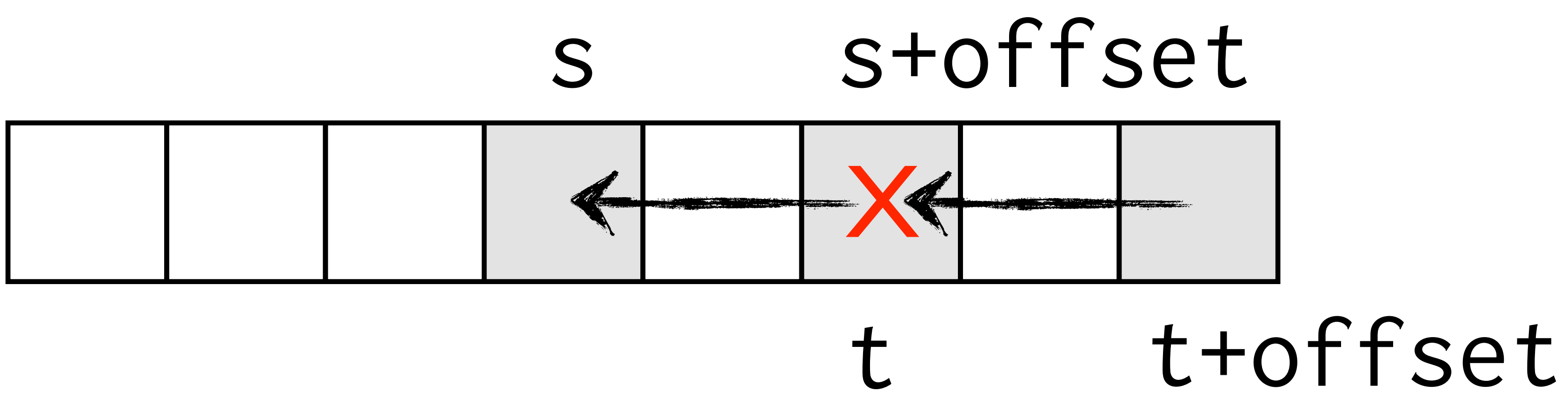
Be Skeptical

- Is the verification easier or harder than building a test harness?

- A common or rare type of bug?
- The impact of not catching this bug
- Limitations of technique

1 Races

```
__kernel void
add_nbor(__local int *A, int offset) {
  int tid = get_local_id(0);
  A[tid] += A[tid+offset];
}
```



- Run GPUVerify on nbor.cl

```
$ cd 1_simple_race
$ gpuverify --local_size=8 --num_groups=1 nbor.cl
```

- Can you fix the datarace?
- Does GPUVerify like your fix?
- Are there more problems with this kernel?

Lessons

- GPUVerify can find possible data races, giving a counterexample for you to evaluate
- By fixing bugs, you increase your confidence in the verification result
- But still, the verification is limited. For example, we don't prove absence of array-bounds or functional correctness

2 Benign Races

```
__kernel void
allsame(__local int *p, int val) {
  *p = val;
}
```

- Run GPUVerify on allsame.cl

```
$ cd 2_benign_race
$ gpuverify --local_size=8 --num_groups=1 allsame.cl
```

- Try adding "`--no-benign`" to the command
- Change "`val`" to "`get_local_id(0)`"
- Have a look at the example in find.cl

Lessons

- Benign data races do not lead to nondeterminism
- Use `--no-benign` flag to warn about benign data races

3 Barrier Divergence

```
__kernel void diverge() {
  int tid = get_local_id(0);
  if (tid == 0) barrier();
  else barrier();
}
```

```
__kernel void inloop() {

  int x = tid == 0 ? 4 : 1;
  int y = tid == 0 ? 1 : 4;

  int i = 0;
  while (i < x) {
    int j = 0;
    while (j < y) {
      barrier(); j++;
    }
    i++;
  }

}
```

- Run GPUVerify on these examples

```
$ cd 3_barrier_divergence
$ gpuverify --local_size=8 --num_groups=1 diverge.cl
$ gpuverify --local_size=8 --num_groups=1 inloop.cl
```

- Is the inloop kernel barrier divergent?
- What does the inloop kernel try to do?

If barrier is inside a conditional statement, then all threads must enter the conditional if any thread enters the conditional statement and executes the barrier.

If barrier is inside a loop, all threads must execute the barrier for each iteration of the loop before any are allowed to continue execution beyond the barrier.

OpenCL Specification
(6.12.8 Synchronization Functions)

$$A = \{\{0,1,2,3\},\{-,-,-,-\}\} \rightarrow \{\{0,1,2,3\},\{1,2,3,0\}\}$$
$$\rightarrow \{\{2,3,0,1\},\{1,2,3,0\}\} \rightarrow \{\{2,3,0,1\},\{3,0,1,2\}\}$$
$$\rightarrow \{\{0,1,2,3\},\{3,0,1,2\}\}$$

GPU	Final state of A
NVIDIA Tesla C2050	{{0,1,0,1},{1,0,1,0}}
AMD Tahiti	{{0,1,2,3},{1,2,3,0}}
ARM Mali-T600	{{0,1,2,3},{3,0,1,2}}
Intel Xeon X5650	{{*,*,*,1},{3,0,1,2}}

Lessons

- Barrier divergence results in undefined behaviour
- GPUVerify can detect such problems
- Arguably, this is a rare bug?

4 Asserts and Assumes

```
__kernel void simple(__local int *A) {
  A[tid] = tid;
  __assert(A[tid] == tid);
  __assert(A[tid] != get_local_size(0));
  __assert(__implies(
    __write(A),
    __write_offset(A)/sizeof(int) == tid));
}
```

- Run GPUVerify on these examples

```
$ cd 4_asserts_and_assumes
$ gpuverify --local_size=8 --num_groups=1 assert.cl
```

- Try writing your own assertions
- Have a look at vacuous.cl
- Does this surprise you?

Lessons

- Use asserts to state expected details of your kernel at a particular program point
- The dangers of inconsistent assumptions
- Use `__assert(false)` to test for inconsistency

5 Loops

```
__kernel void inc(int x) {

  int i = 0;
  while (i < x) {
    i = i + 1;
  }
  __assert(i == x);

}
```

```
__kernel void inc(int x) {
  __requires (0 < x);

  int i = 0;
  while (i < x) {
    i = i + 1;
  }
  __assert(i == x);

}
```

```
__kernel void inc(int x) {
  __requires (0 < x);

  int i = 0;
  while (__invariant(?), i < x) {
    i = i + 1;
  }
  __assert(i == x);

}
```

- Run GPUVerify on these examples

```
$ cd 5_loops
$ gpuverify --local_size=8 --num_groups=1 inc.cl
```

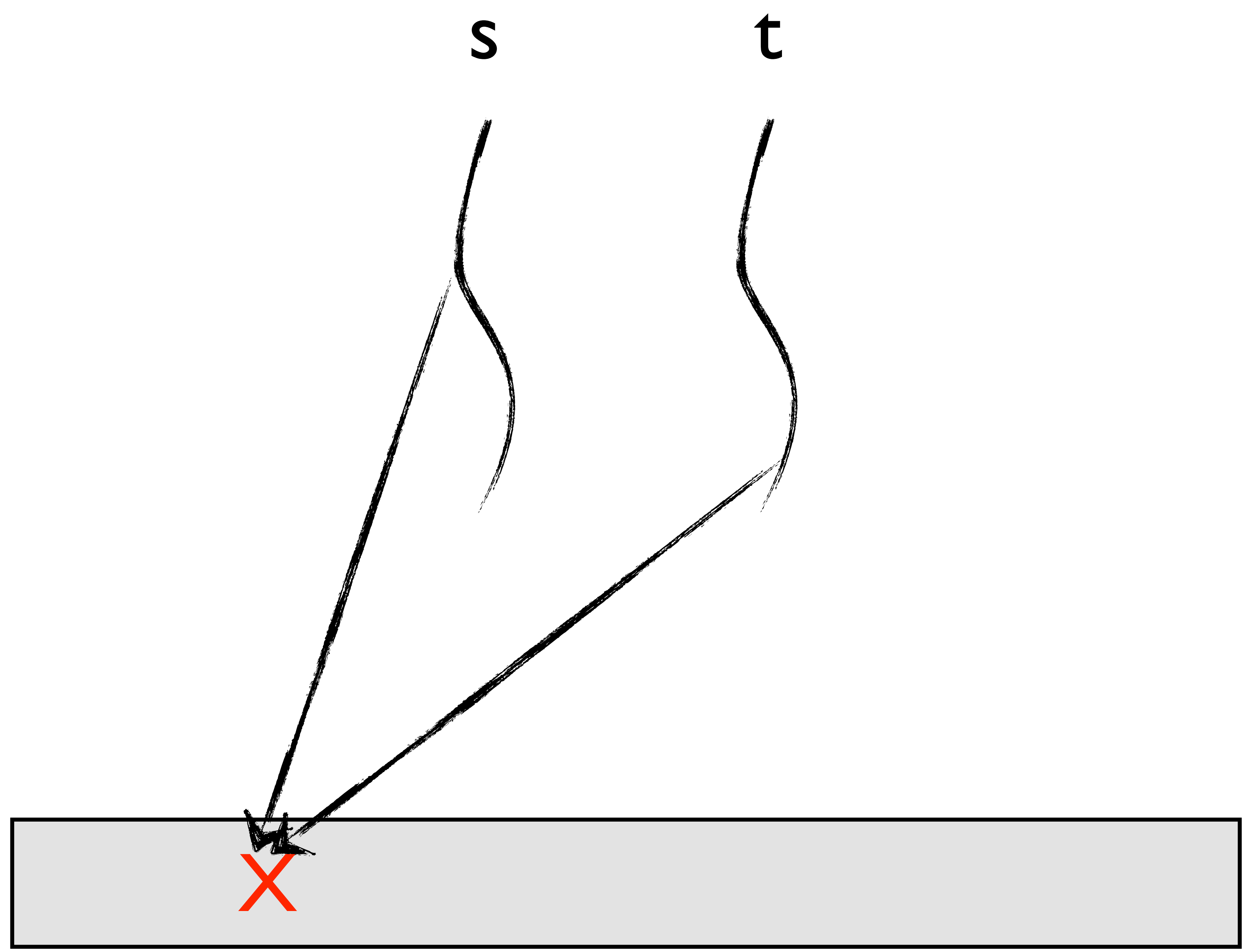
- Try running with the "`--findbugs`" flag
- Can you find an invariant for the loop?
- Take a look at stride.cl

Lessons

- Loop invariants are assertions that are true at every loop iteration
- GPUVerify attempts to guess invariants
- They may be necessary to strengthen verification to avoid false-positives
- Use `--findbugs` to do loop unwinding

Anatomy of GPUVerify

2-thread reduction



Arbitrary threads s and t

```
barrier() // b1







barrier() // b2
```

Arbitrary threads s and t

```
barrier() // b1
```

run s from b1 to b2
log all accesses

```
barrier() // b2
```

Arbitrary threads s and t

```
barrier() // b1
```

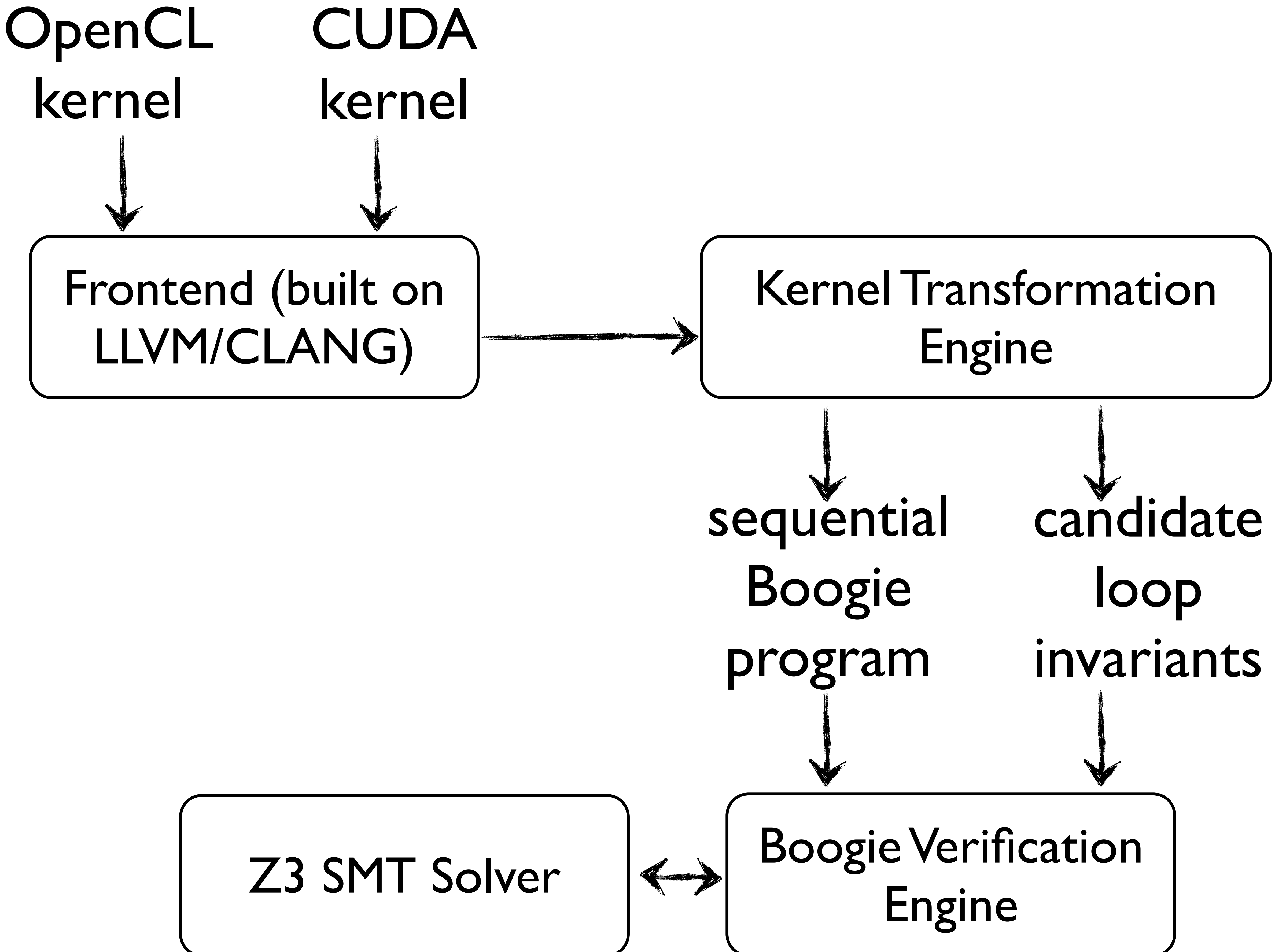
run s from b1 to b2
log all accesses

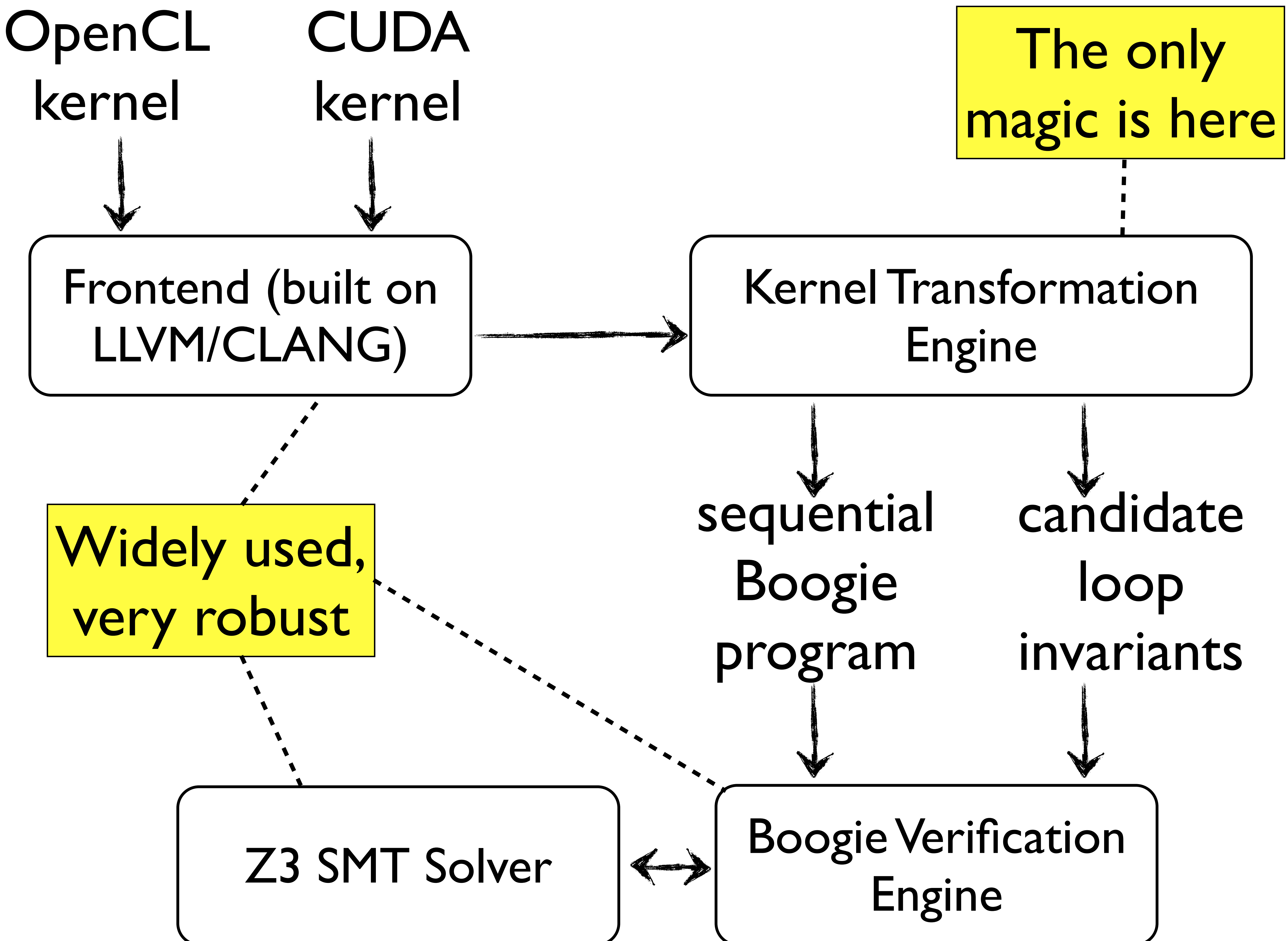
run t from b1 to b2
check all accesses against s
abort on race

```
barrier() // b2
```

2-thread reduction gives scalable verification

Translate parallel kernel K into sequential program P such that P correct implies K is race-free


OpenCL kernel
CUDA kernel
Frontend (built on LLVM/CLANG)
Kernel Transformation Engine
sequential Boogie program
candidate loop invariants
Z3 SMT Solver
Boogie Verification Engine


OpenCL kernel
CUDA kernel
The only magic is here
Frontend (built on LLVM/CLANG)
Kernel Transformation Engine
sequential Boogie program
candidate loop invariants
Widely used, very robust
Z3 SMT Solver
Boogie Verification Engine

Further Examples

```
__kernel void dbl_indirect(__local int *A) {

  A[tid] = tid;
  barrier();
  A[A[(tid+1)%N]] = tid;

}
```

0	1	2	3	4	5	6	7

```
__kernel void dbl_indirect(__local int *A) {

  A[tid] = tid;
  barrier();
  A[A[(tid+1)%N]] = tid;

}
```

7	0	1	2	3	4	5	6

```
barrier() // b1
```

run s from b1 to b2
log all accesses

run t from b1 to b2
check all accesses against s

```
barrier() // b2
```

run s from b2 to b3
log all accesses

run t from b2 to b3
check all accesses against s

```
barrier() // b3
```

```
barrier() // b1
```

run s from b1 to b2
log all accesses

run t from b1 to
check all ac s

```
barrier() /
```

ru to b3
log esses

run t from b2 to b3
check all accesses against s

```
barrier() // b3
```

unsound

havoc shared state

```
barrier() // b1
```

run s from b1 to b2
log all accesses

run t from b1 to b2
check all accesses against s

```
barrier() // b2
```

run s from b2 to b3
log all accesses

run t from b2 to b3
check all accesses against s

```
barrier() // b3
```

Shared state abstraction is necessary for soundness

GPUVerify: sound and scalable verification for GPU kernels

GPUVerify: A Verifier for GPU Kernels *

Adam Betts[1] Nathan Chong[1] Alastair F. Donaldson[1] Shaz Qadeer[2] Paul Thomson[1]

[1]Department of Computing, Imperial College London, UK [2]Microsoft Research, Redmond, USA

{abetts,nyc04,afd,pt1110}@imperial.ac.uk qadeer@microsoft.com

Abstract

We present a technique for verifying race- and divergence-freedom of GPU kernels that [illegible]re wr[illegible]ten in mainstream kernel p[illegible] [illegible]CL a[illegible]. Our a[illegible] [illegible] operational [illegible]mantics for GPU programming termed *synchronous, delayed visibility* (SDV) semantics. The SDV semantics provides a precise definition of barrier dive[illegible] in GP[illegible] and allows kernel verification to b[illegible] to a[illegible] of a sequential program, thereby comp[illegible] avoiding [illegible] need to reason about thread interleavings, and allowing existing modular techniques for program verification to be leveraged. We describe an efficient encoding for data race detection and propose a method for automatically inferring loop invariants required for verification. We have implemented these techniques as a practical verification tool, GPUVerify, which can be applied directly to OpenCL and CUDA source code. We evaluate GPUVerify with respect to a set of 163 kernels drawn from public and commercial sources. Our evaluation [illegible] GPUVerify is capable of efficient, auto[illegible] [illegible] a large number of real-world kernels.

Categories and Subject Descriptors F3.1 [*Logics and*

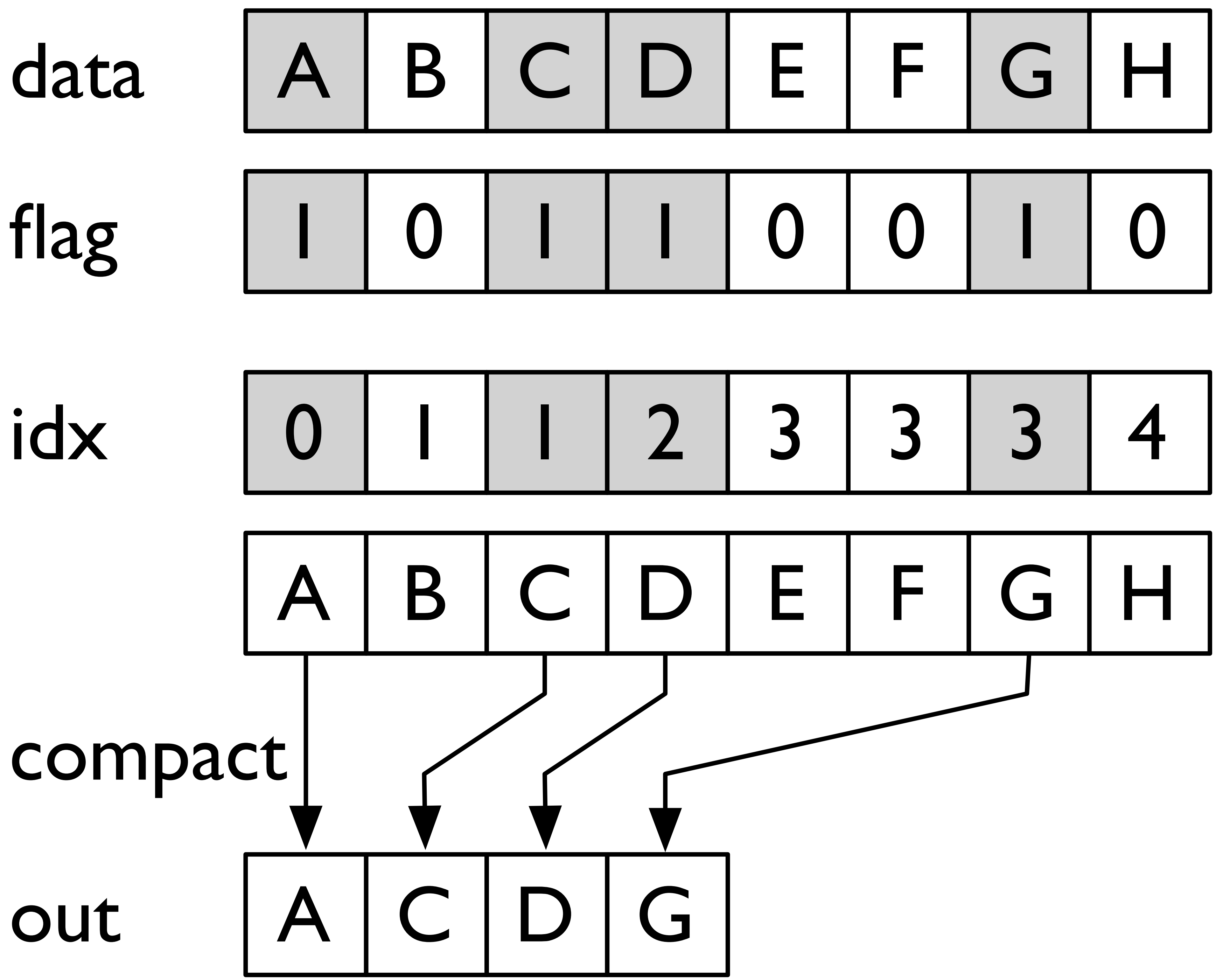
such as AMD and NVIDIA, have become widely available to end-users. Accelerators offer tremendous compute power at [illegible] cost, and tasks such as media processing, medical [illegible] be accelerated [illegible] GPU pe[illegible] magnitude.

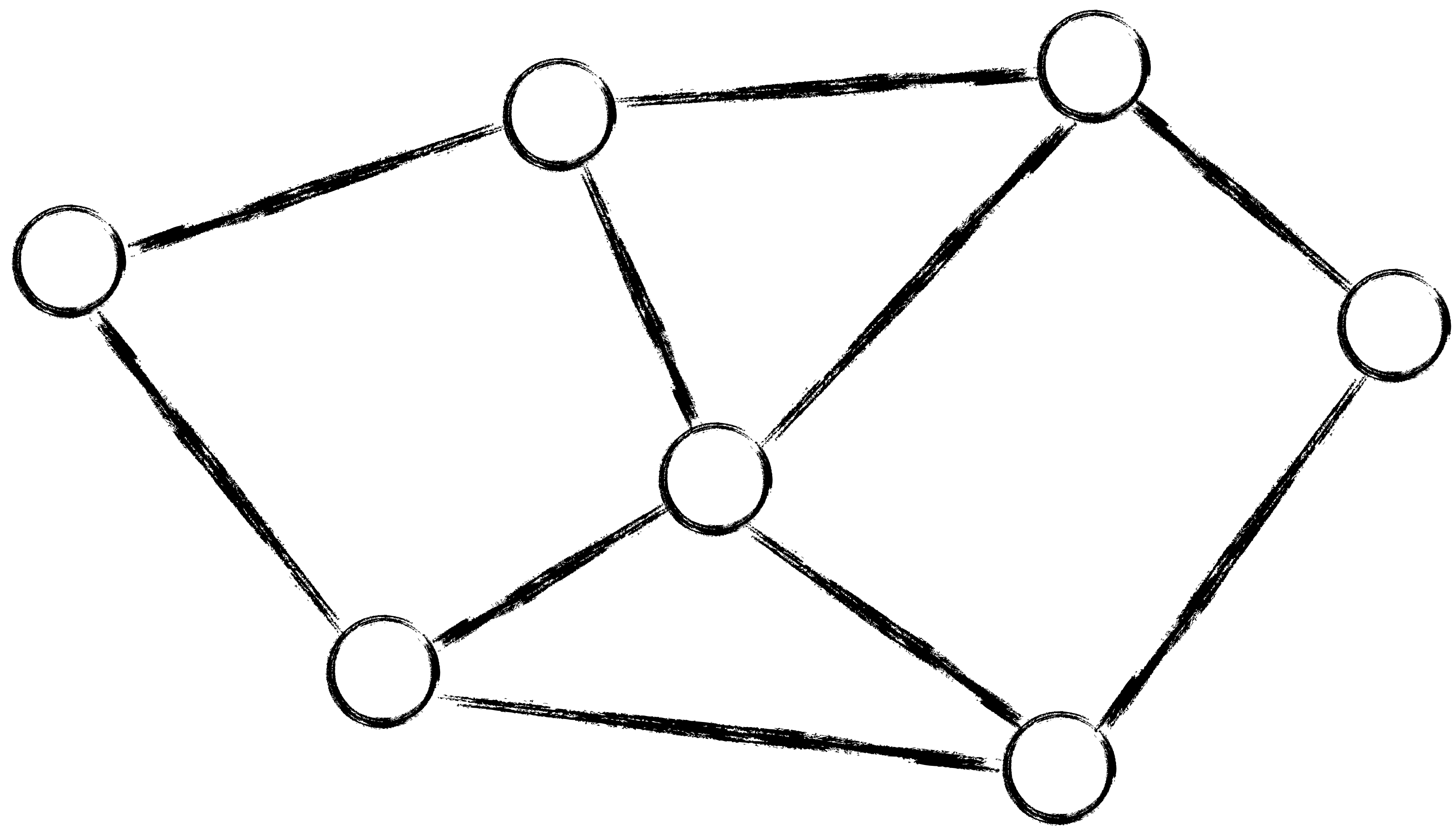
GPUs present a serious challenge for software developers. A system may contain one or more of the plethora of devices on the market, [illegible]th many more products anticipated [illegible] [illegible]cations must exhibit *portable* [illegible] operat[illegible] [illegible]ly on *any* GPU accelerator. Software bugs in media processing domains can have serious financial implications, and GPUs are being used increasingly in domains such as medical image processing [37] where safety is critical. Thus there is an urgent need for verification techniques to aid construction of correct GPU software.

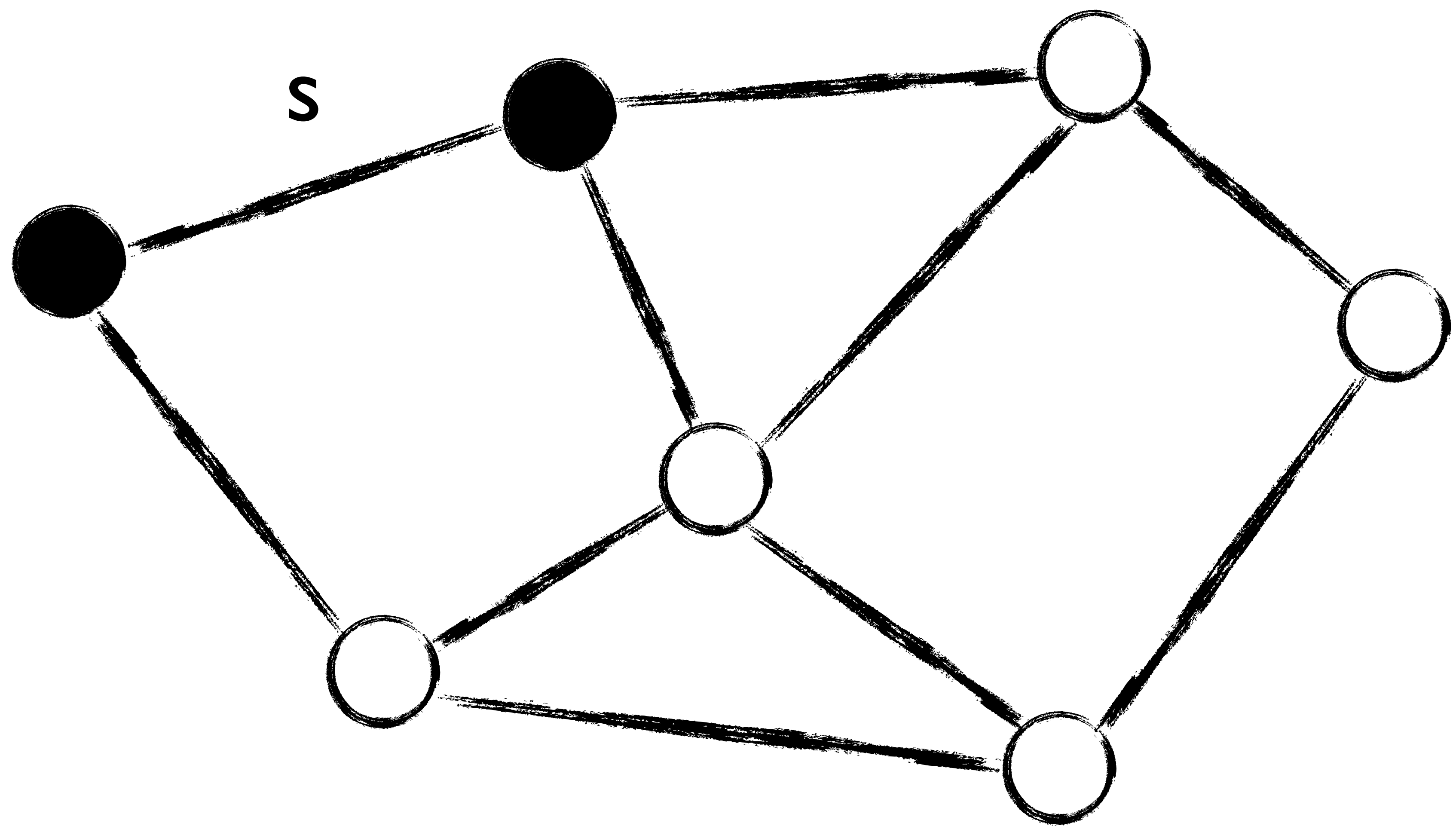
This paper addresses the problem of static verification of GPU kernels written in kernel programming languages such as OpenCL [17], CUDA [30] and C++ AMP [28]. We focus on two classes of bugs which make writing correct GPU kernels harder than writing correct sequential code: *data races* and *barrier divergence*.

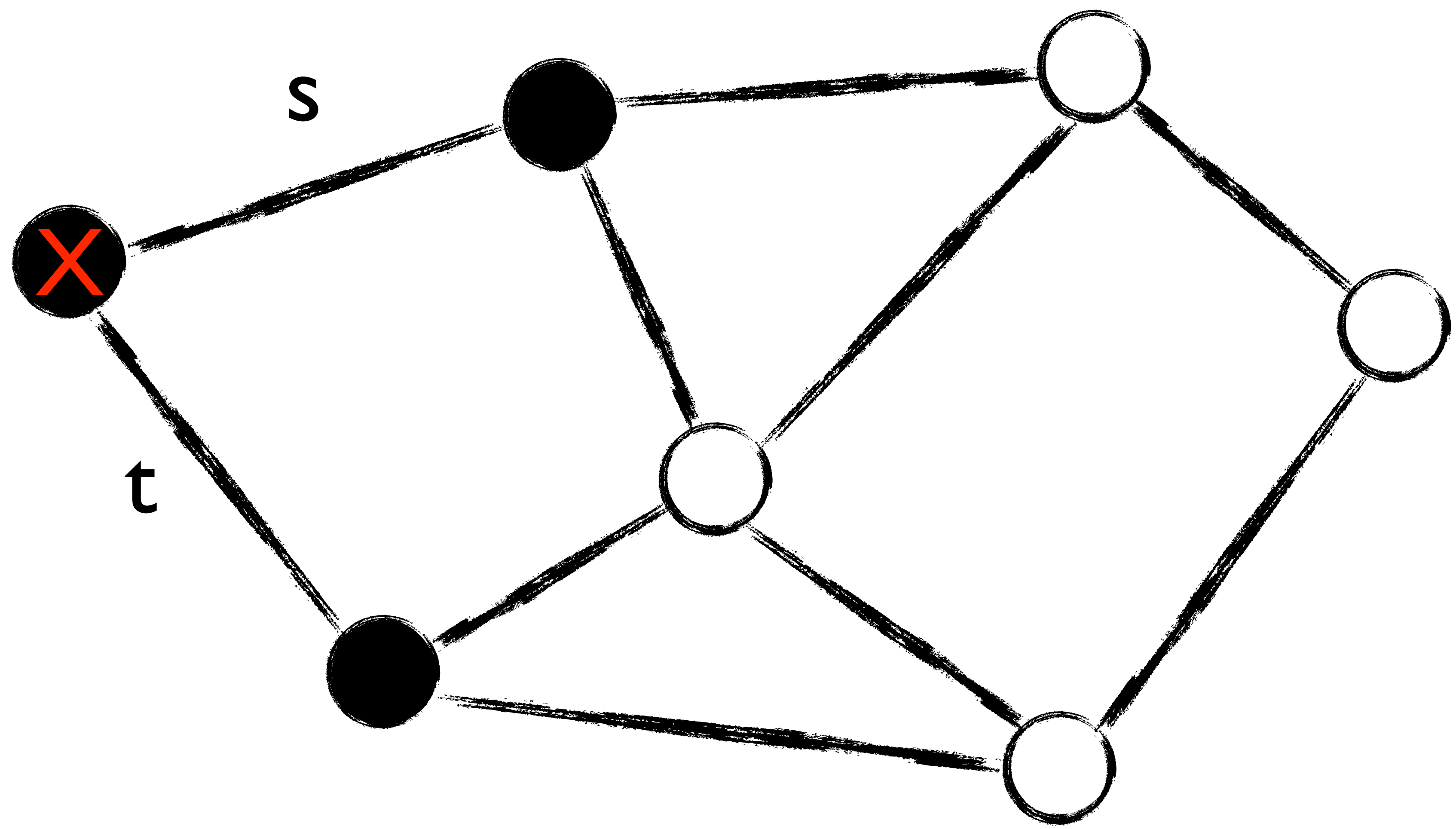
In contrast to the well-understood notion of data races, there does not appear to be a formal definition of barrier di-

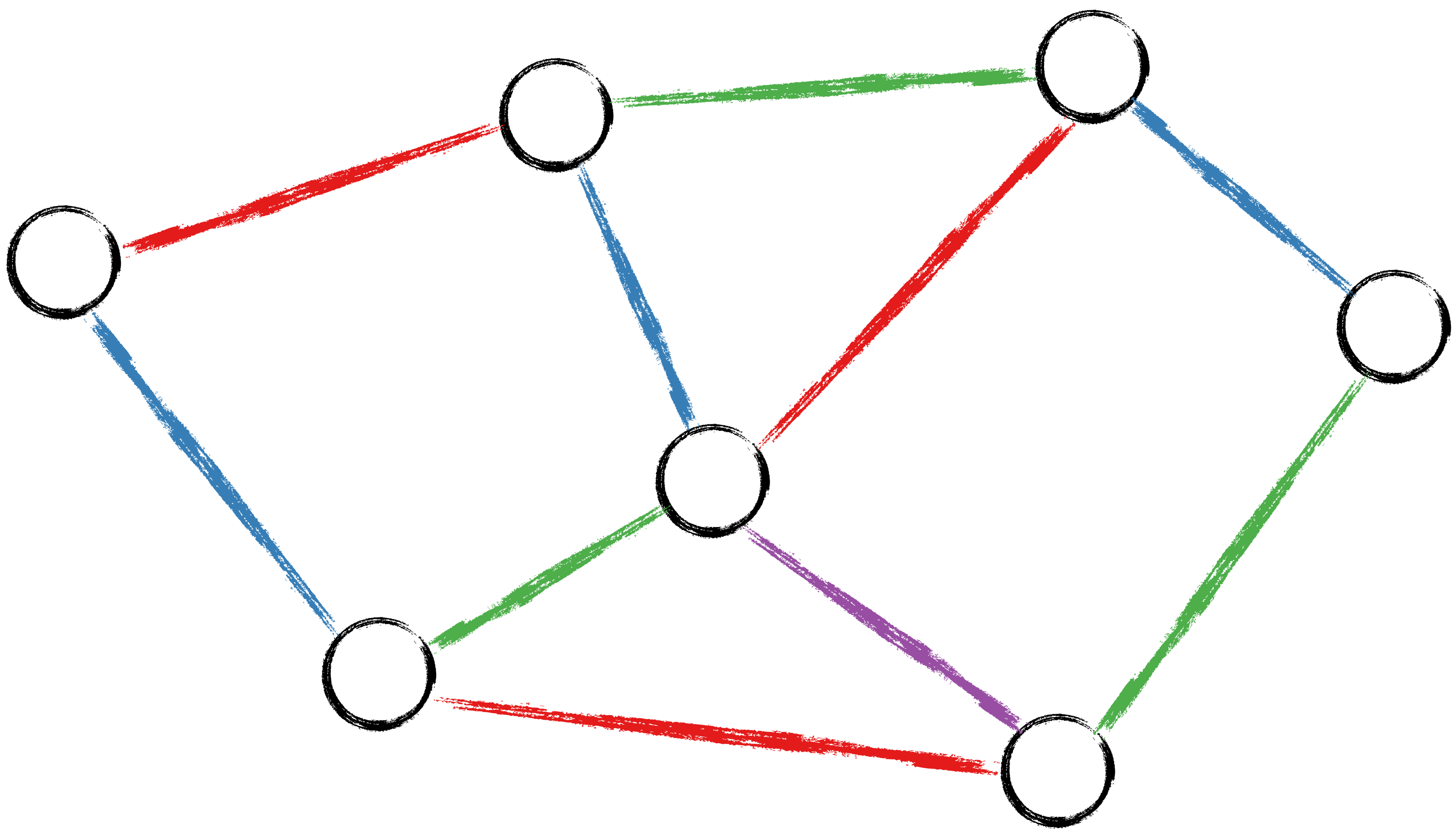
In OOPSLA'12


data
A B C D E F G H
flag
1 0 1 1 0 0 1 0
idx
0 1 1 2 3 3 3 4
A B C D E F G H
compact
out
A C D G




S

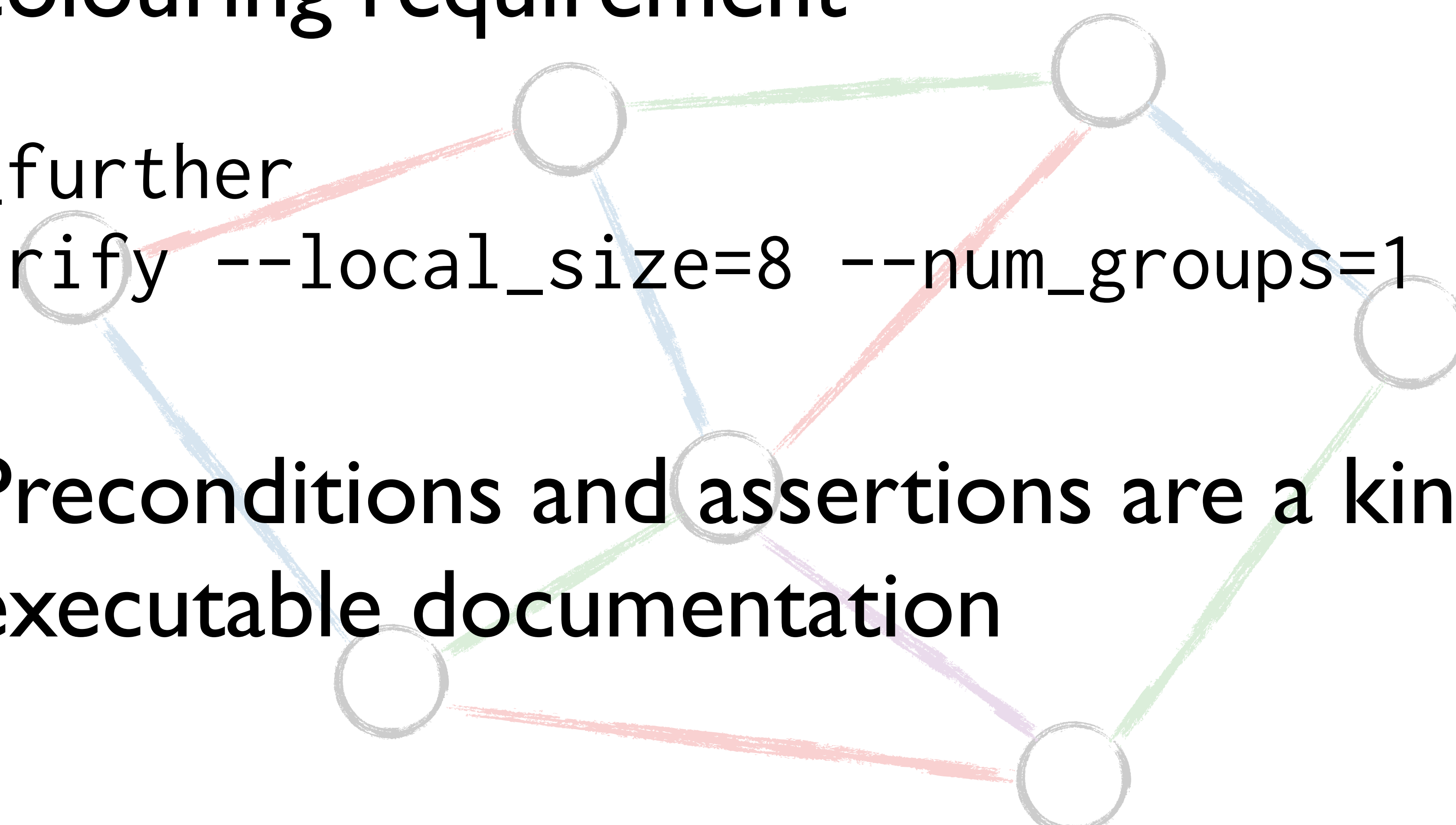

s
X
t



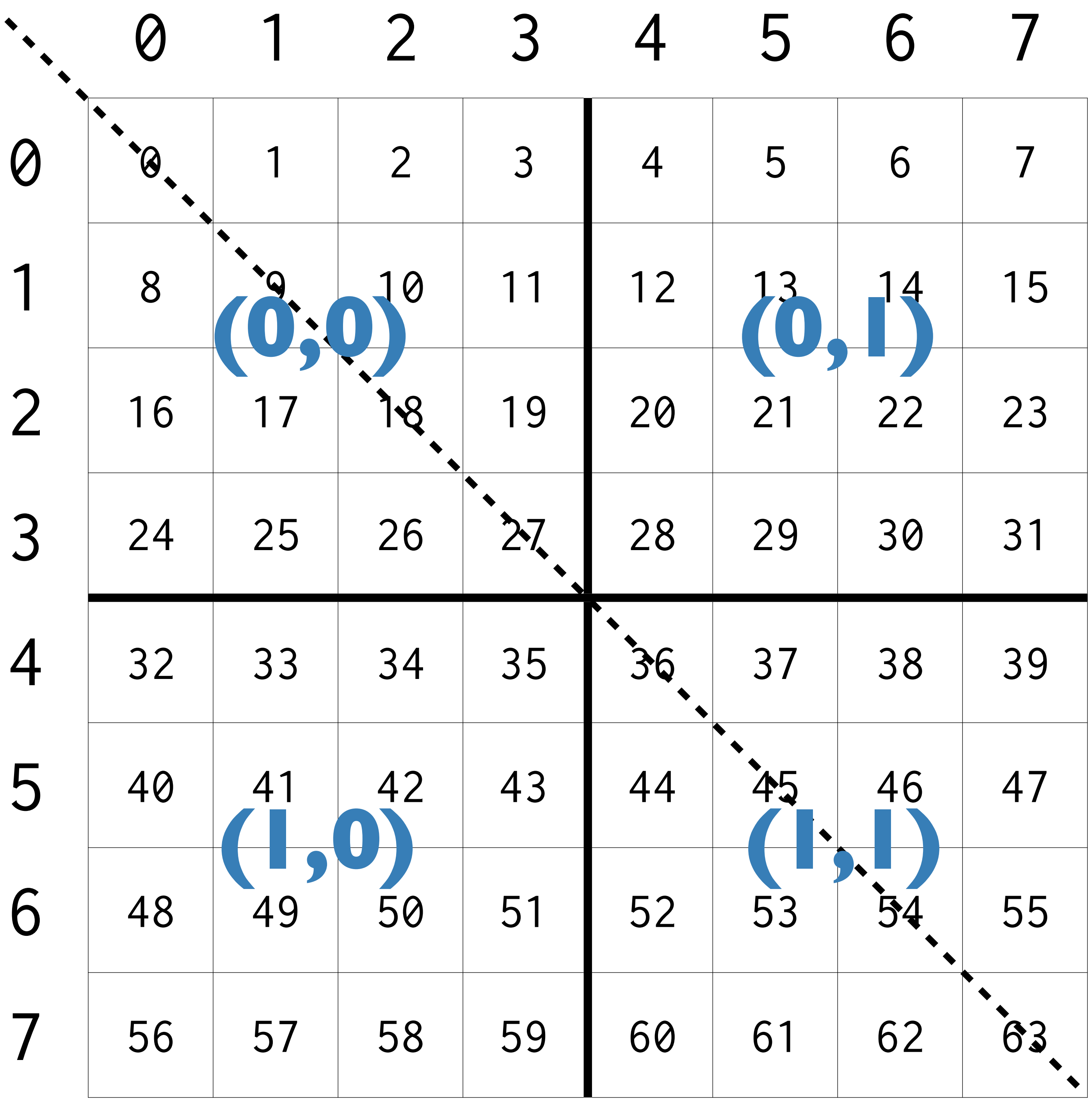
```
__kernel void iterall_edges(
  __local uint2 *edges,
  __local uint  *edgecolour,
  __local float *node_val
) {

  __requires(?);

  for (uint c=0; c < MAX_COLOUR; c++) {
    if (c == edgecolour[tid]) {
      node_val[edges[tid].lo] = ...;
      node_val[edges[tid].hi] = ...;
    }
    barrier();
  }

}
```

- Write a precondition that satisfies the colouring requirement

```
$ cd 6_further
$ gpuverify --local_size=8 --num_groups=1 graph.cl
```



- Preconditions and assertions are a kind of executable documentation



width = 8

height = 8

Row Major

A_{ij} stored at

`i + (width*j)`

- Check out transpose.cu

```
$ cd 6_further
$ gpuverify --blockDim=[4,2] --gridDim=[2,2]
-DWIDTH=8 -DHEIGHT=8 -DTILE_DIM=4 -DBLOCK_ROWS=2
transpose.cu
```

- Involves tricky loop invariants for reasoning about data accesses of individual threads
- More invariants than lines of code!

Lessons

- Valuable to know the limitations of the tools you use
- Discovering loop invariants can be time-consuming (but rewarding!)
- It is possible to reason about complicated kernels if the engineering investment is worthwhile

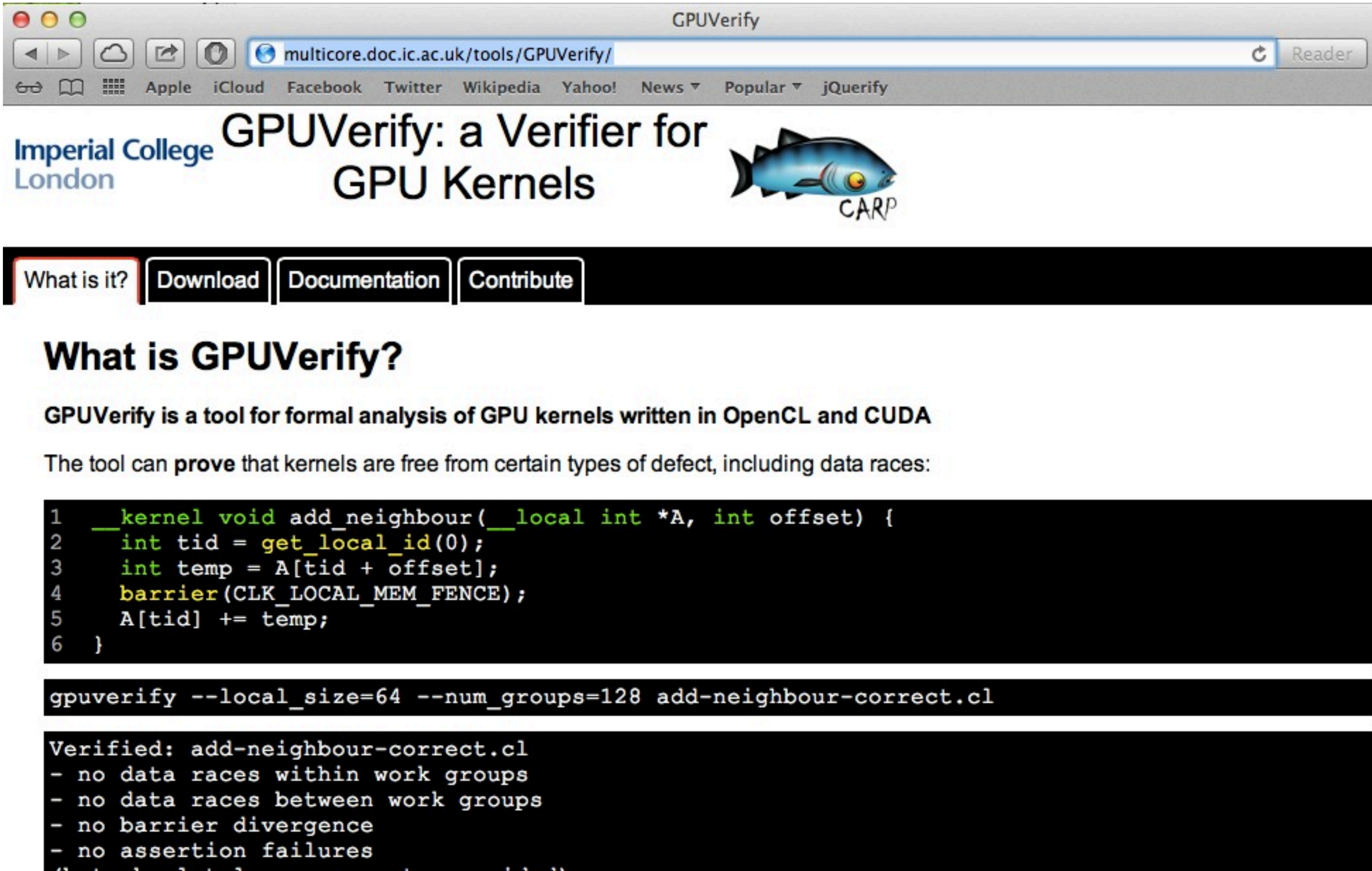
Closing

Verification as a powerful and practical complement to Testing

Formal reasoning as a valuable discipline

Search ‘GPUVerify’ on YouTube

http://multicore.doc.ic.ac.uk/tools/GPUVerify



What is GPUVerify?

GPUVerify is a tool for formal analysis of GPU kernels written in OpenCL and CUDA

The tool can **prove** that kernels are free from certain types of defect, including data races:

```
__kernel void add_neighbour(__local int *A, int offset) {
  int tid = get_local_id(0);
  int temp = A[tid + offset];
  barrier(CLK_LOCAL_MEM_FENCE);
  A[tid] += temp;
}
```

```
gpuverify --local_size=64 --num_groups=128 add-neighbour-correct.cl
```

```
Verified: add-neighbour-correct.cl
- no data races within work groups
- no data races between work groups
- no barrier divergence
- no assertion failures
```

Alastair Donaldson

Microsoft Research	Shaz Qadeer
Frontend	Adam Betts Peter Collingbourne
Semantics heavy lifting	Jeroen Ketema
PhD students	Paul Thomson Nathan Chong Dan Liew
UROP students	Egor Kyshtymov Cassie Epps

Work supported by EU FP7 STREP project CARP (project number 287767) and EPSRC PSL project (EP/I006761/1).