Semantics and Verification for GPU Kernels

Nathan Chong and Jeroen Ketema
Imperial College London
{nyc04, j.ketema}@imperial.ac.uk

Imperial College London

Show how the *theory, algorithms and tools* of the verification community can be brought to bear on an interesting and important domain

Schedule

- GPUs and their programming models
- The need for semantics and verification tools
- Building a static analyser with off-the-shelf parts
- Examining the GPUVerify verification method

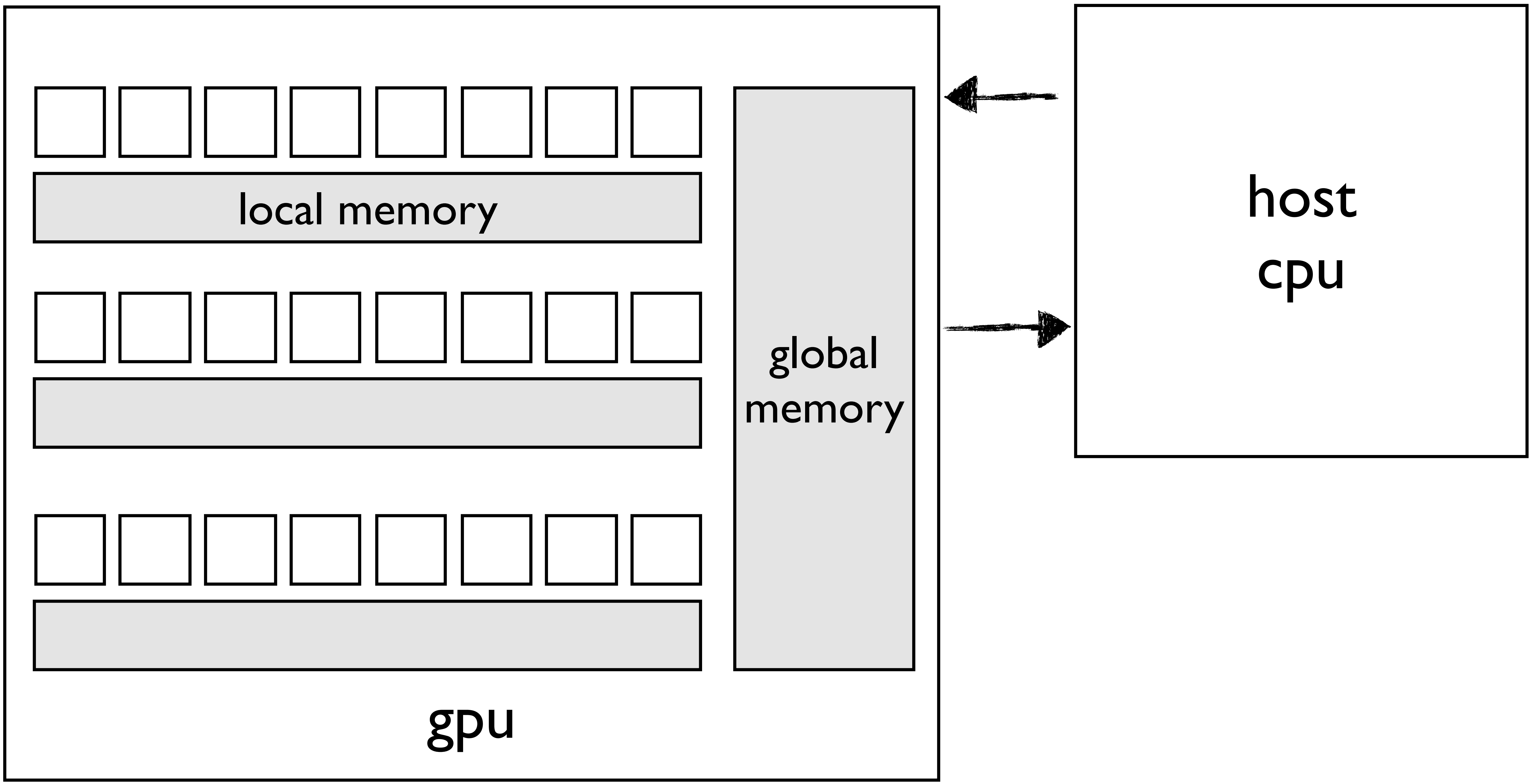
Organisation

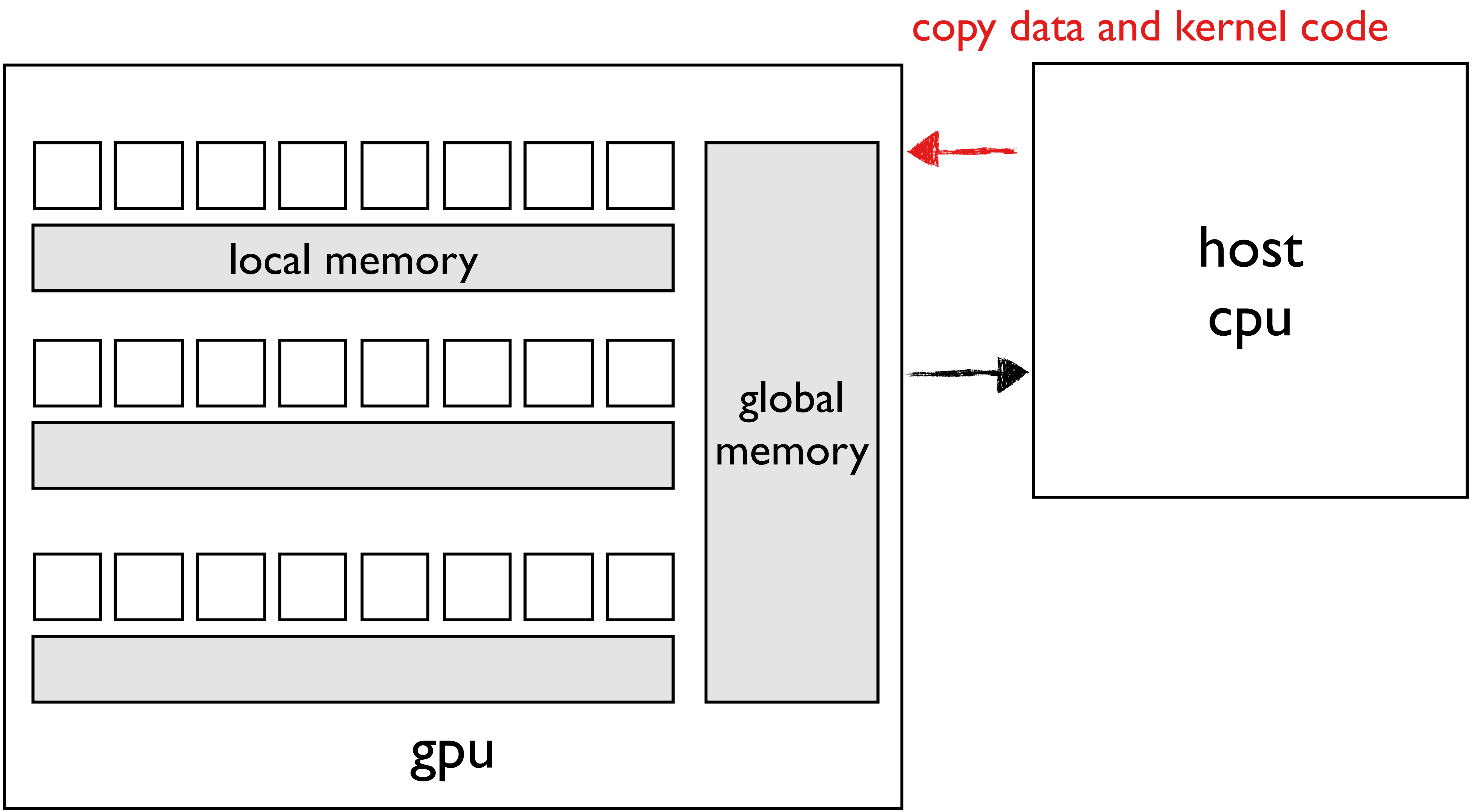
- Please ask questions!
- Material online
  http://multicore.doc.ic.ac.uk/tools/GPUVerify/tutorials/POPL14
- Try GPUVerify online
  http://rise4fun.com/GPUVerify-CUDA
  http://rise4fun.com/GPUVerify-OpenCL

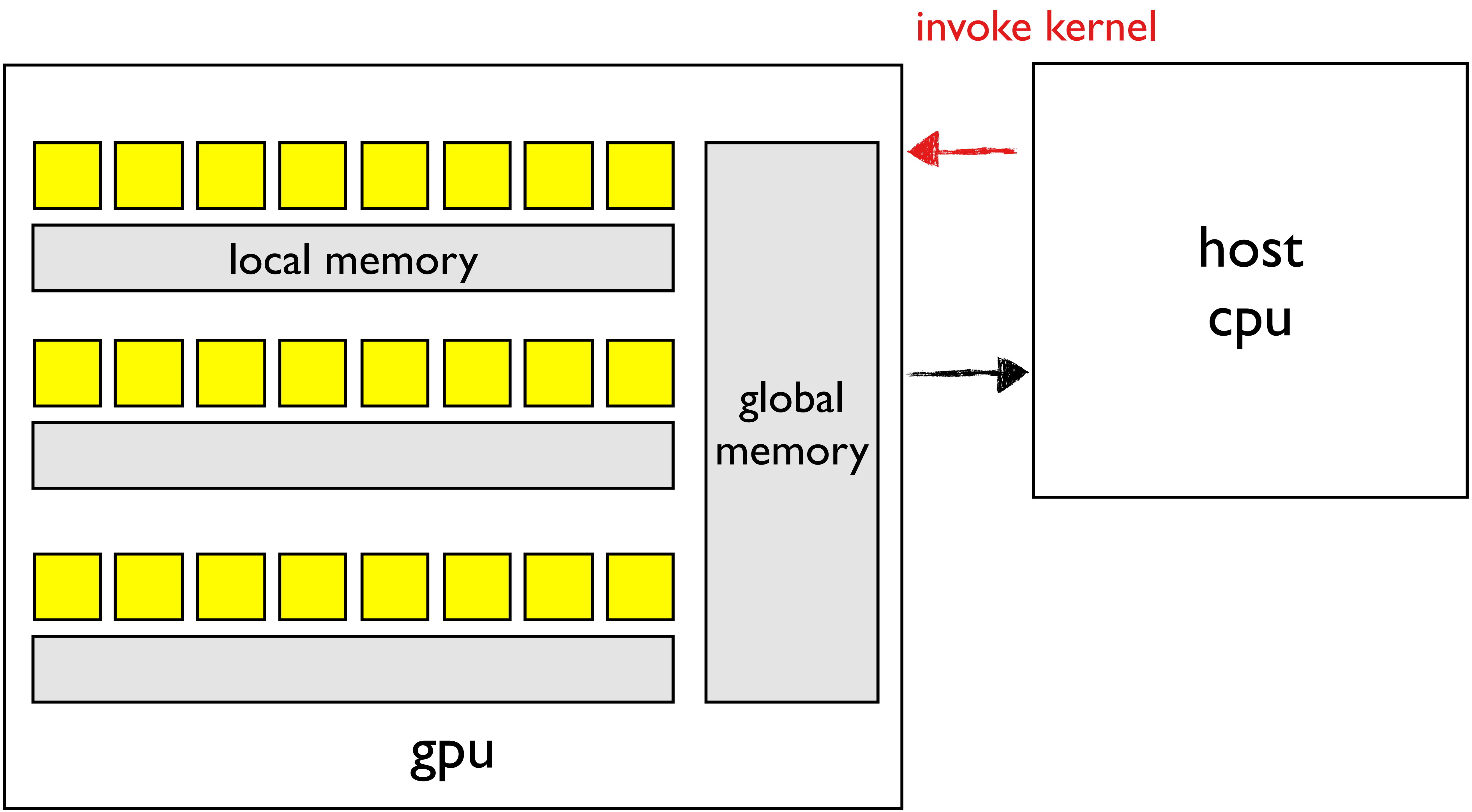
GPUs and their programming models

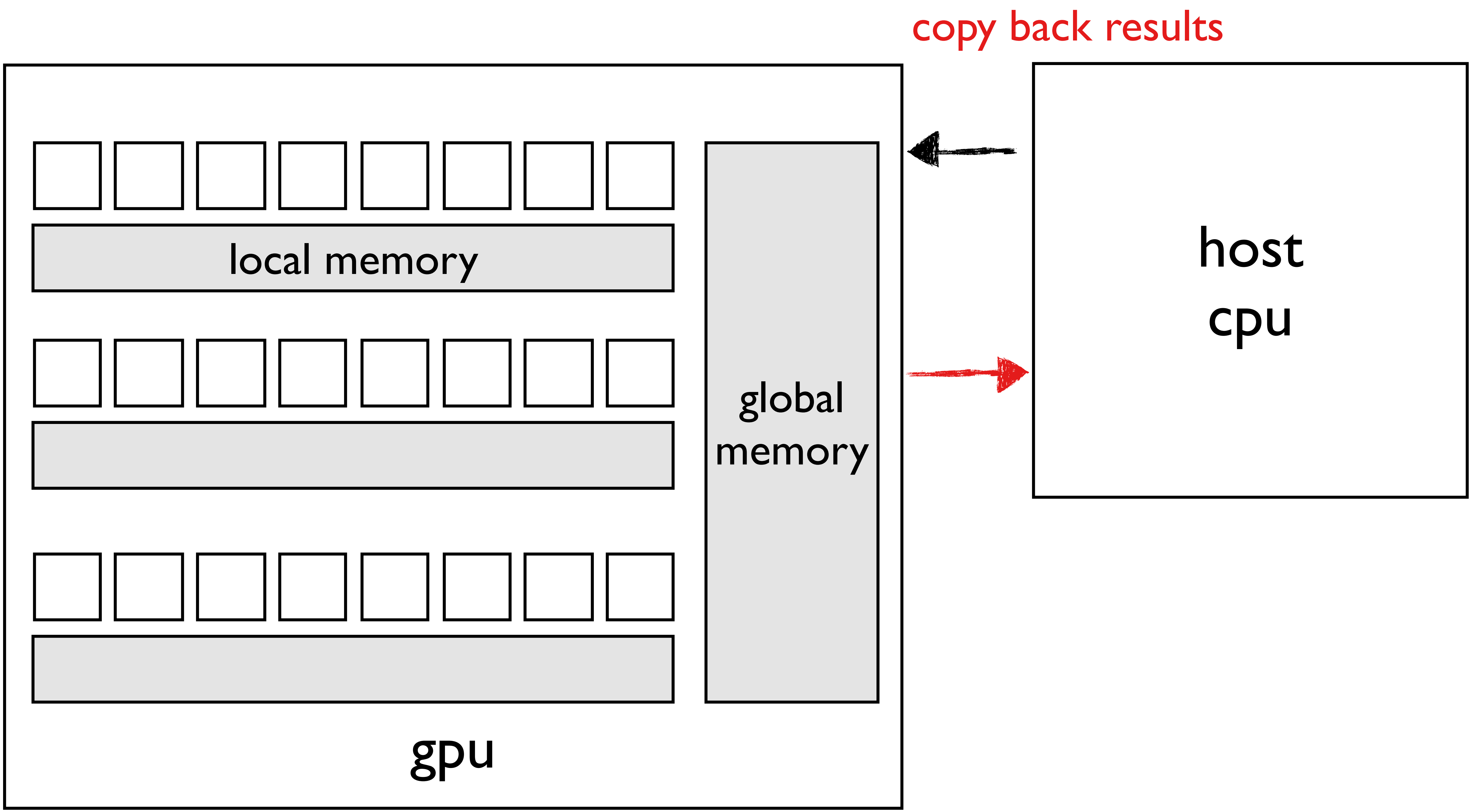
Graphics Processing Units

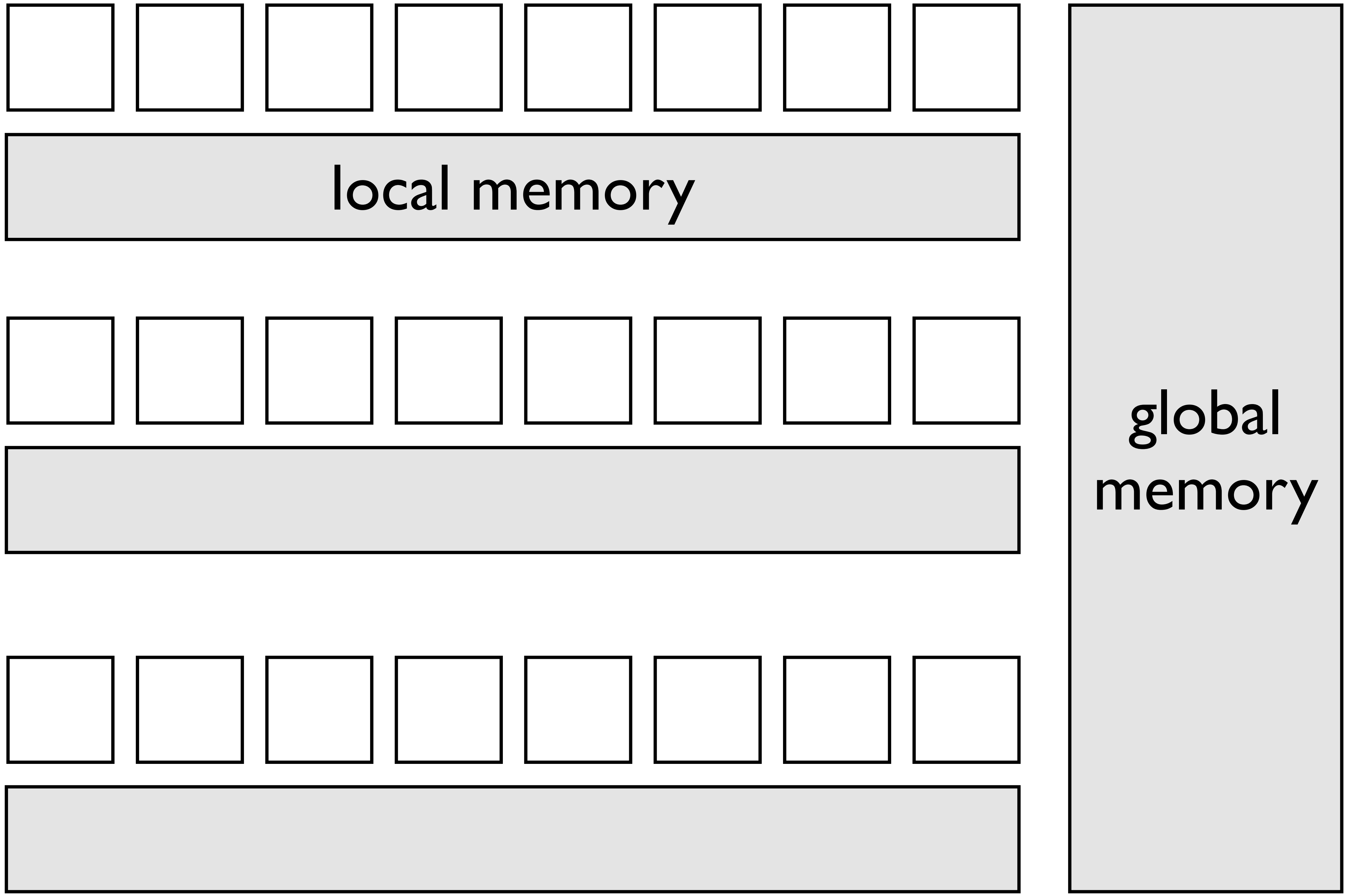
- Originally designed to accelerate graphics processing
- Operations on pixels are inherently parallel
- Early GPUs were specialised for graphics
- Modern GPUs are more powerful and general purpose
- Now widely used as accelerators for many domains


local memory
global memory
gpu
host
cpu

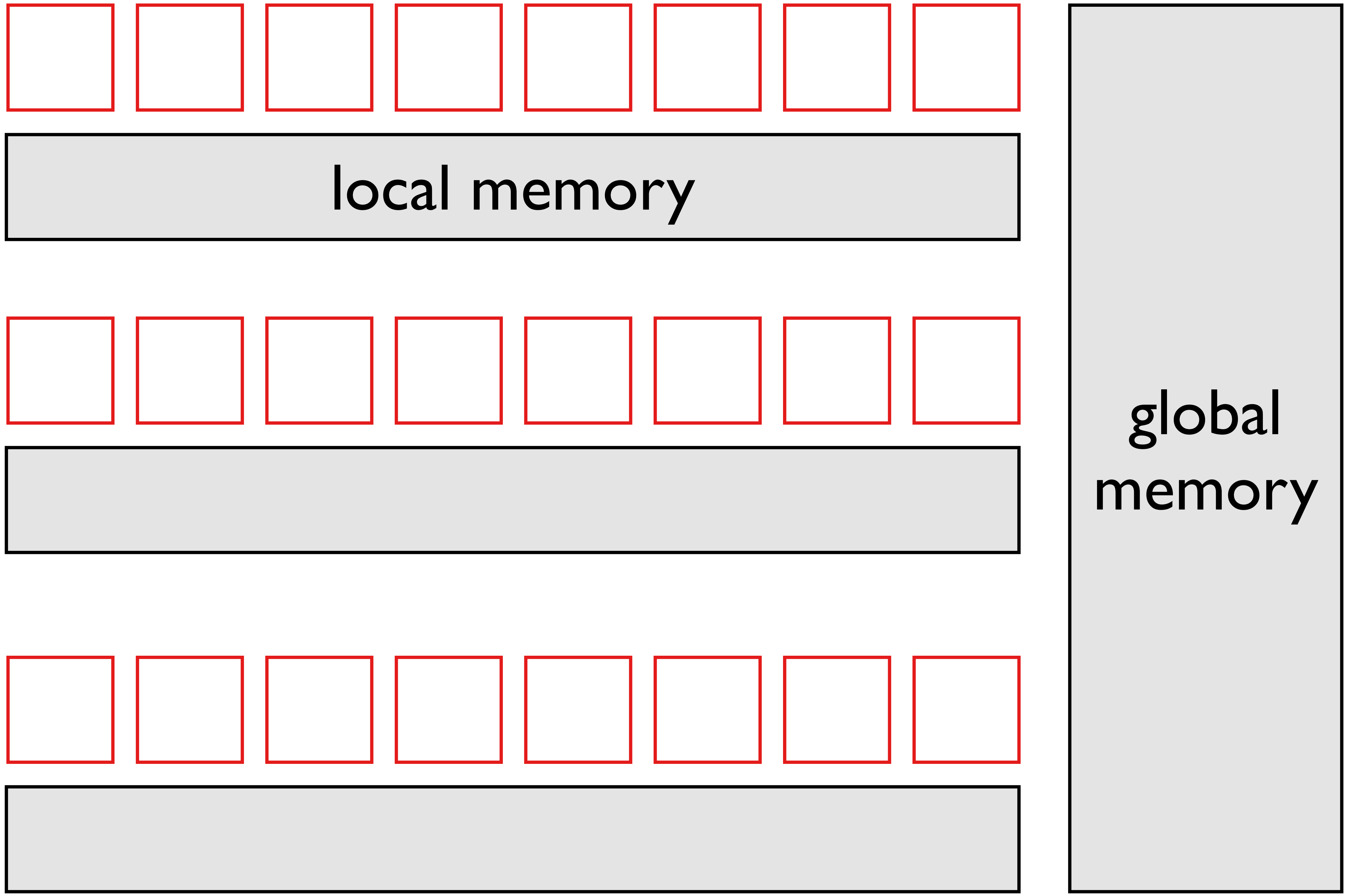

copy data and kernel code
local memory
global memory
host
cpu
gpu


invoke kernel
local memory
global memory
gpu
host
cpu

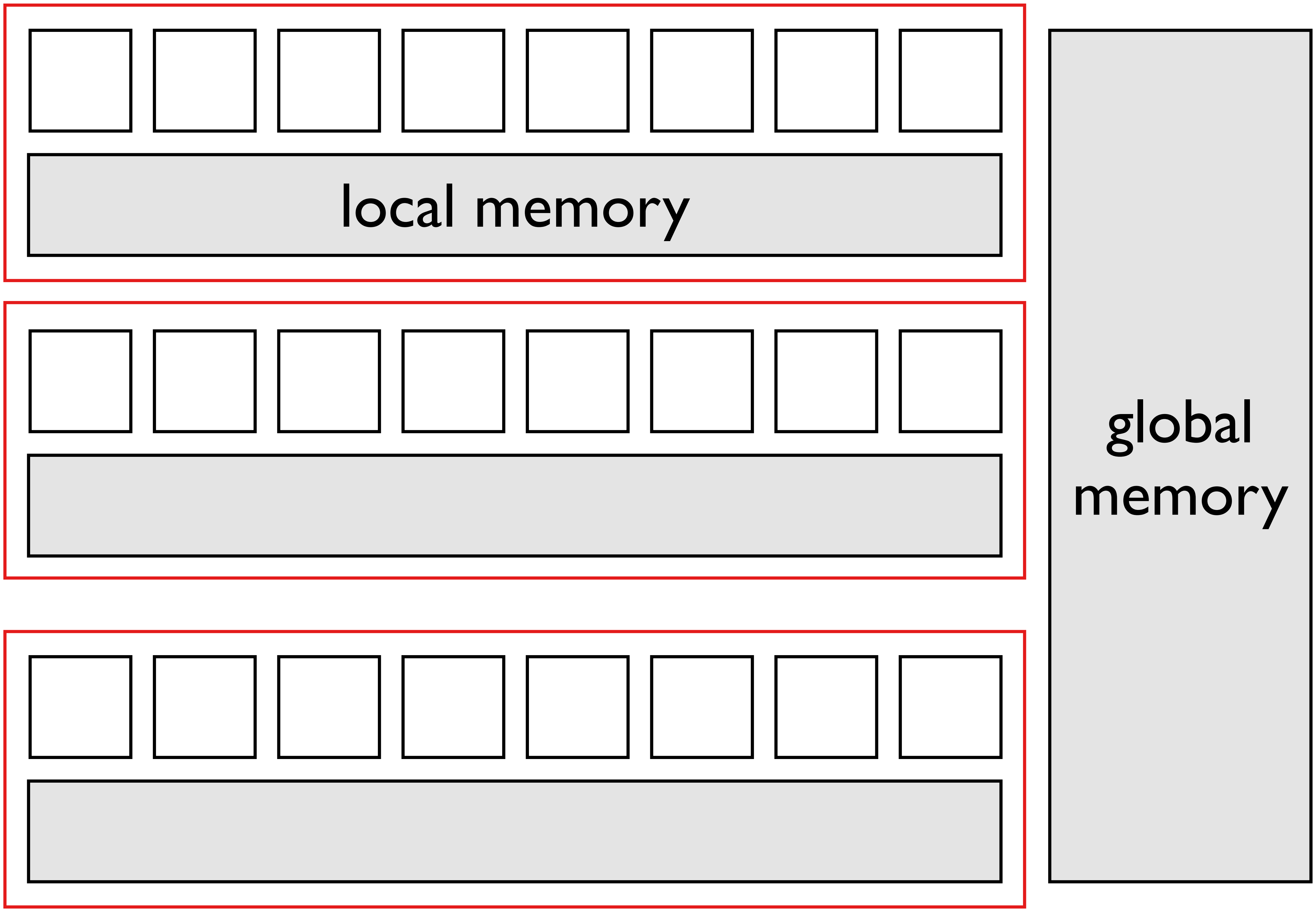

copy back results
local memory
global memory
host
cpu
gpu


local memory
global memory

processing elements or cores



multiprocessors



CUDA and OpenCL

thread / work-item

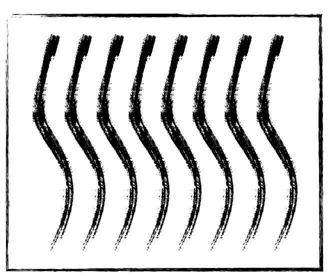
core

private

block / workgroup

multiprocessor

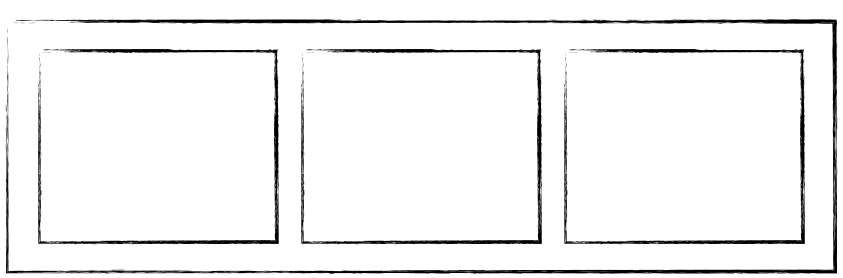
shared / local



grid / ndrange

gpu

global



```
__kernel void vadd(
  __local int *A, __local int *B,
  __local int *C) {

  int tid = get_local_id(0);

  C[tid] = A[tid] + B[tid];

}
```

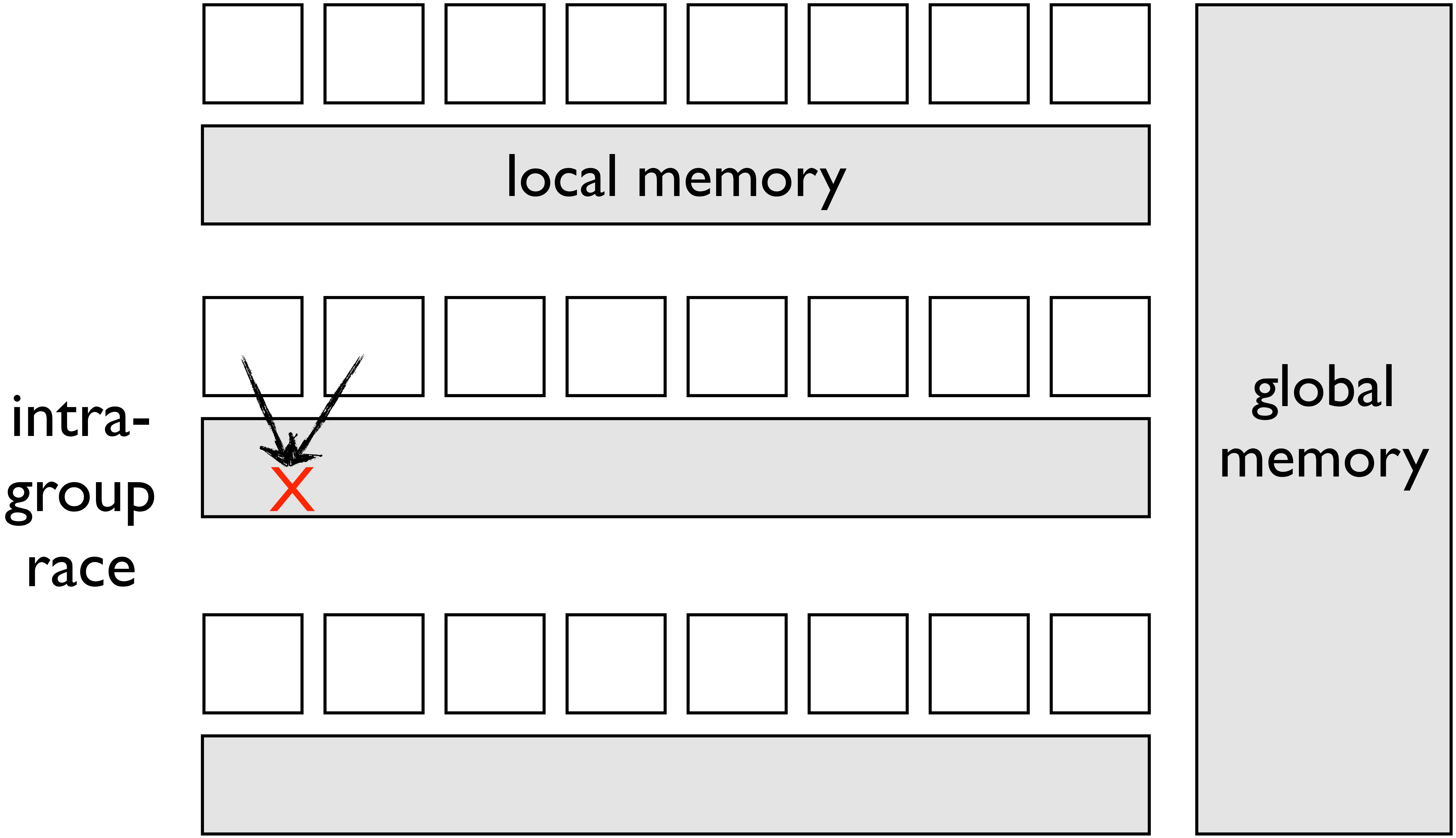
Summary

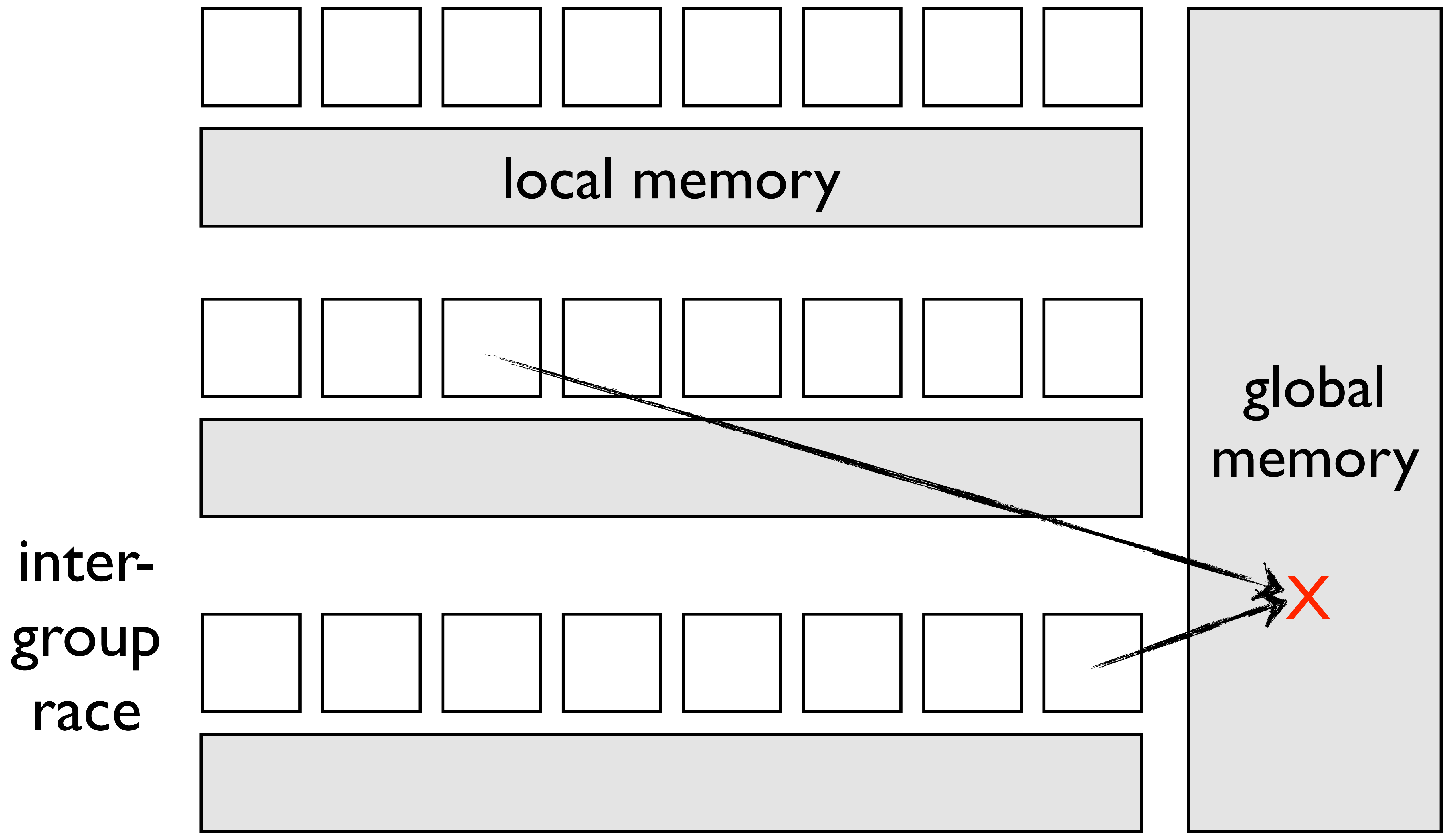
- Kernels are run on massively parallel GPUs
- Potentially hundreds of thousands of threads acting in concert over shared memory
- Wide variety of applications and domains

The need for semantics and verification tools

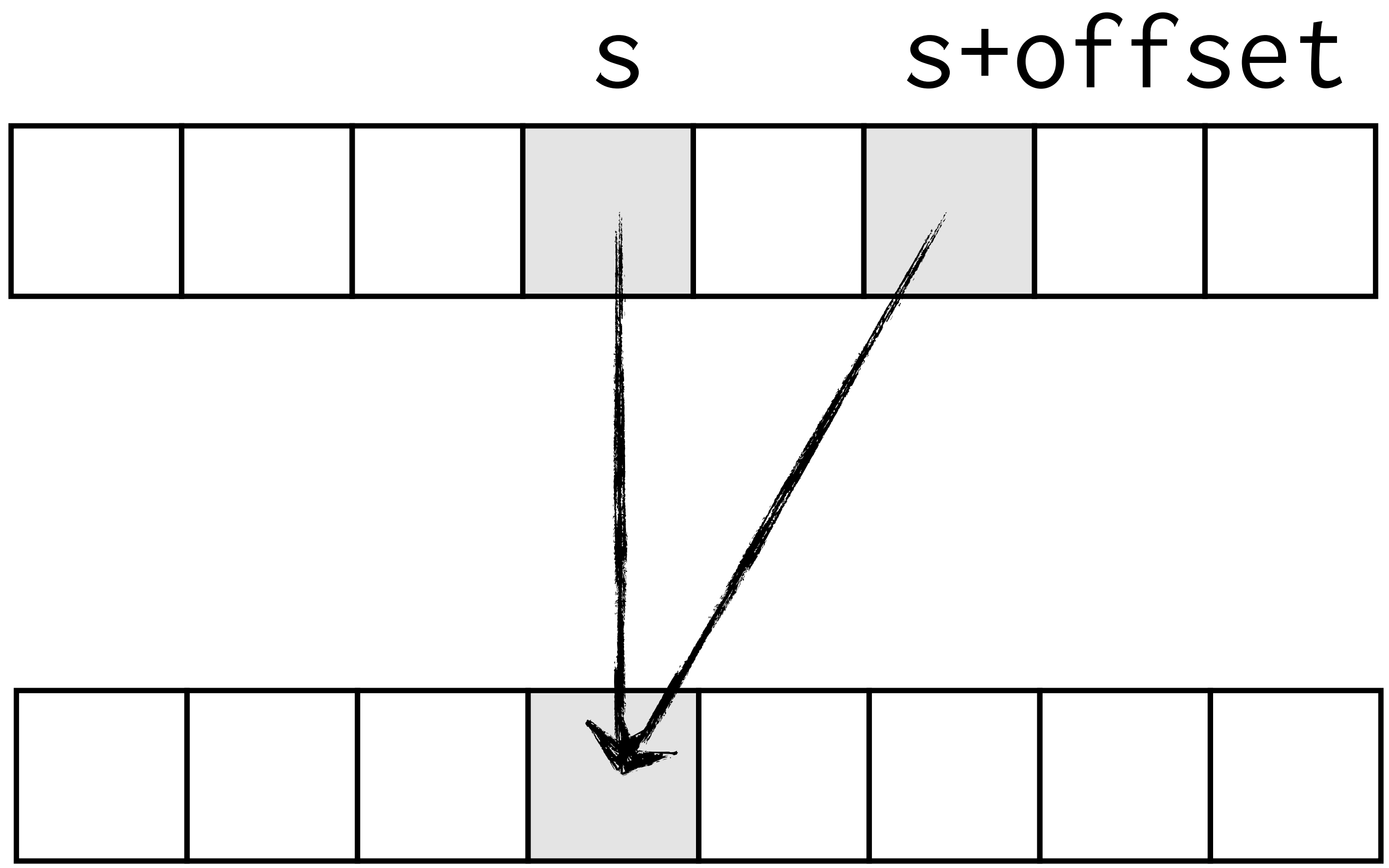
Data races

- Two distinct threads access the same memory location
- At least one of the accesses is a write
- The accesses are not separated by a barrier synchronisation operation

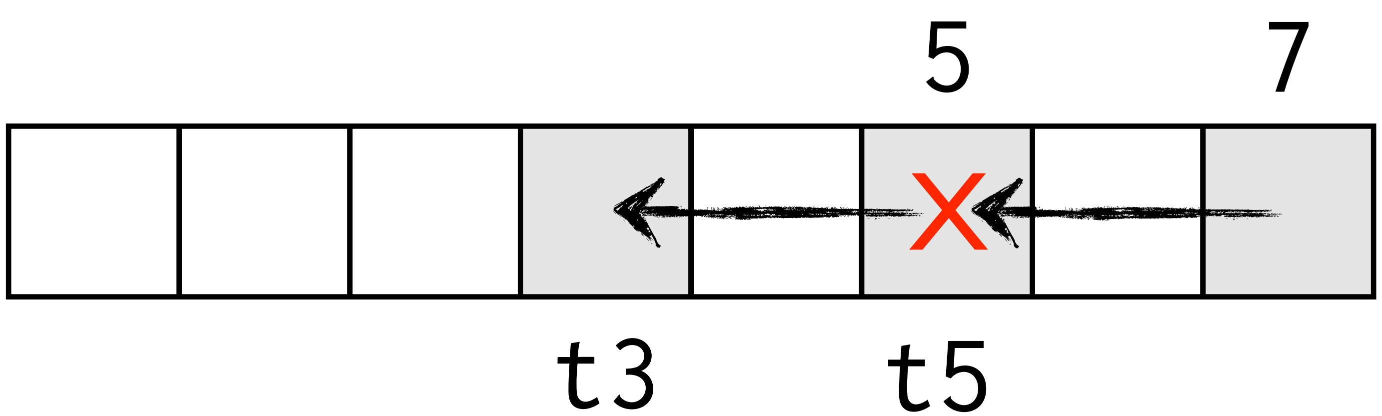

local memory
global memory
intra-
group
race
X

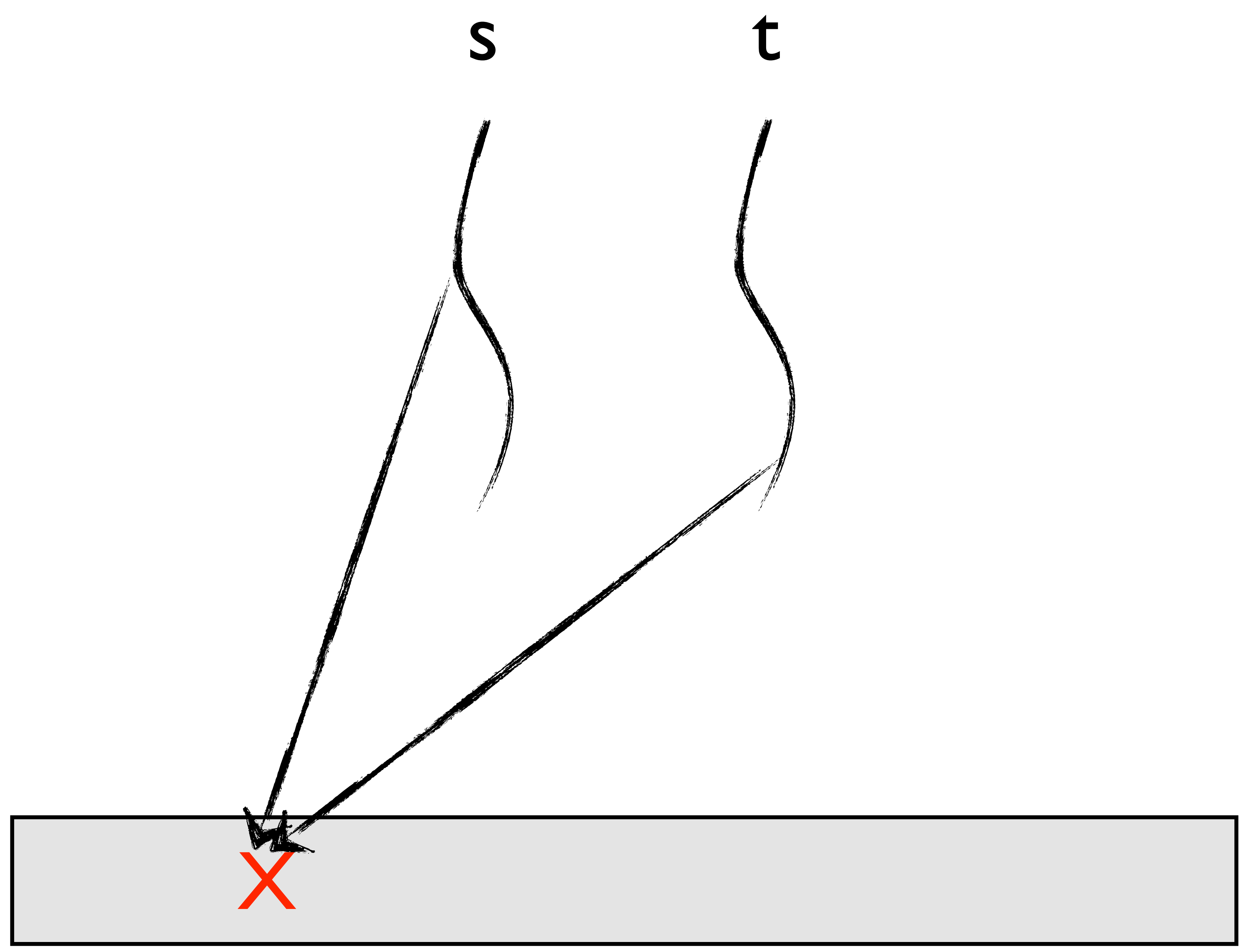

local memory
global memory
X
inter-
group
race

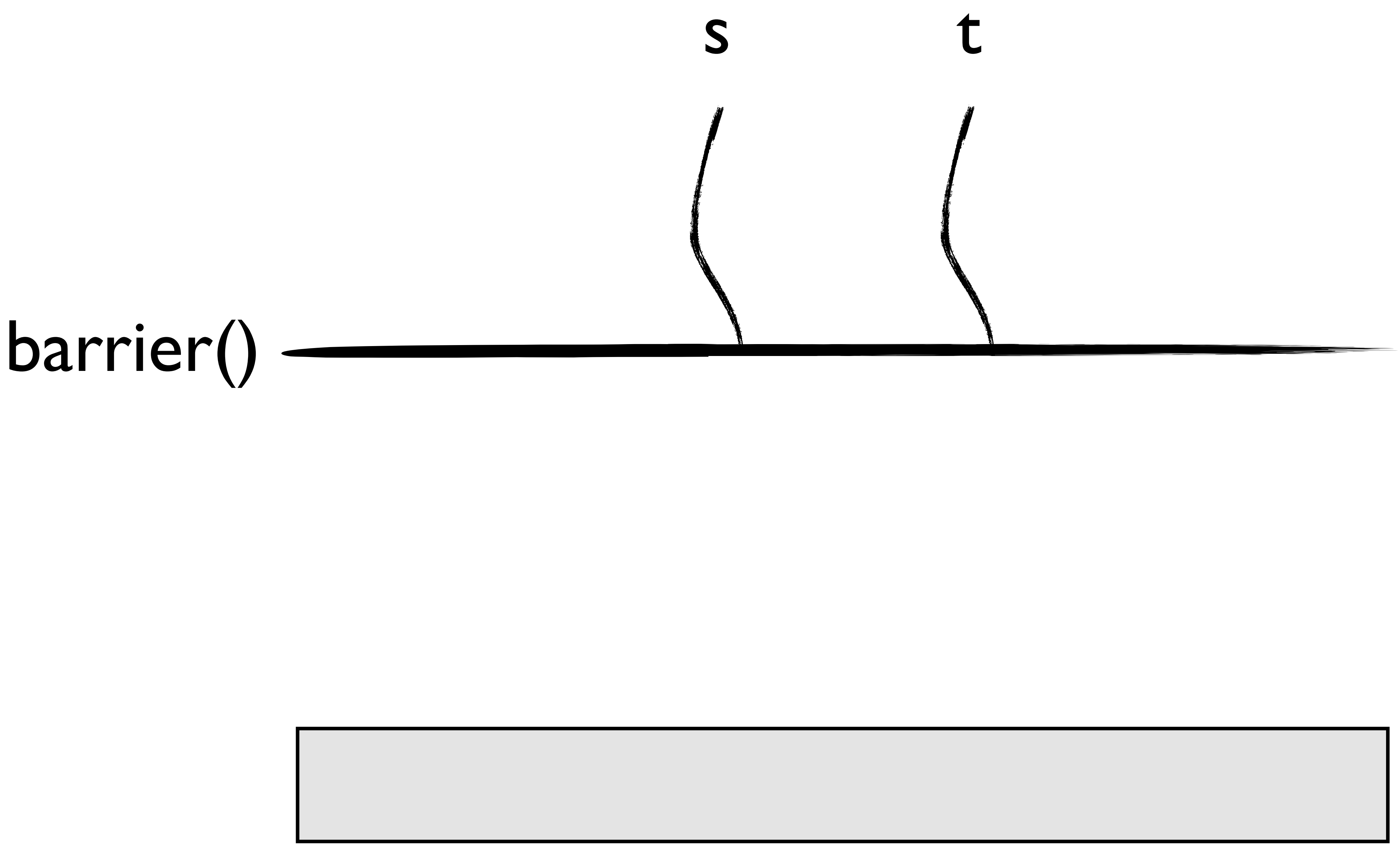
```
__kernel void
add_nbor(__local int *A, int offset) {
  int tid = get_local_id(0);
  A[tid] = A[tid] + A[tid+offset];
}
```

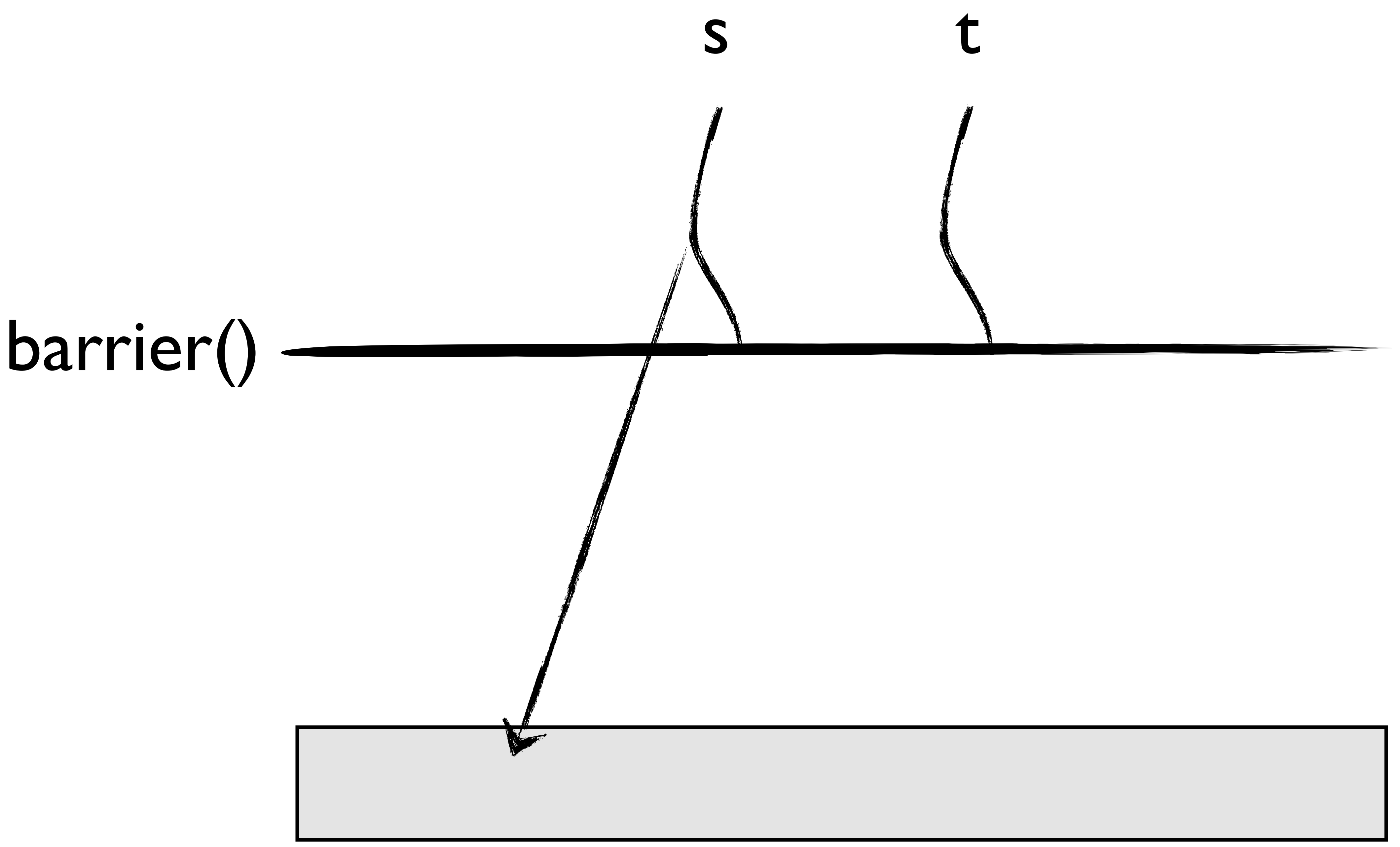


```
__kernel void
add_nbor(__local int *A, int offset) {
  int tid = get_local_id(0);
  A[tid] = A[tid] + A[tid+offset];
}
```

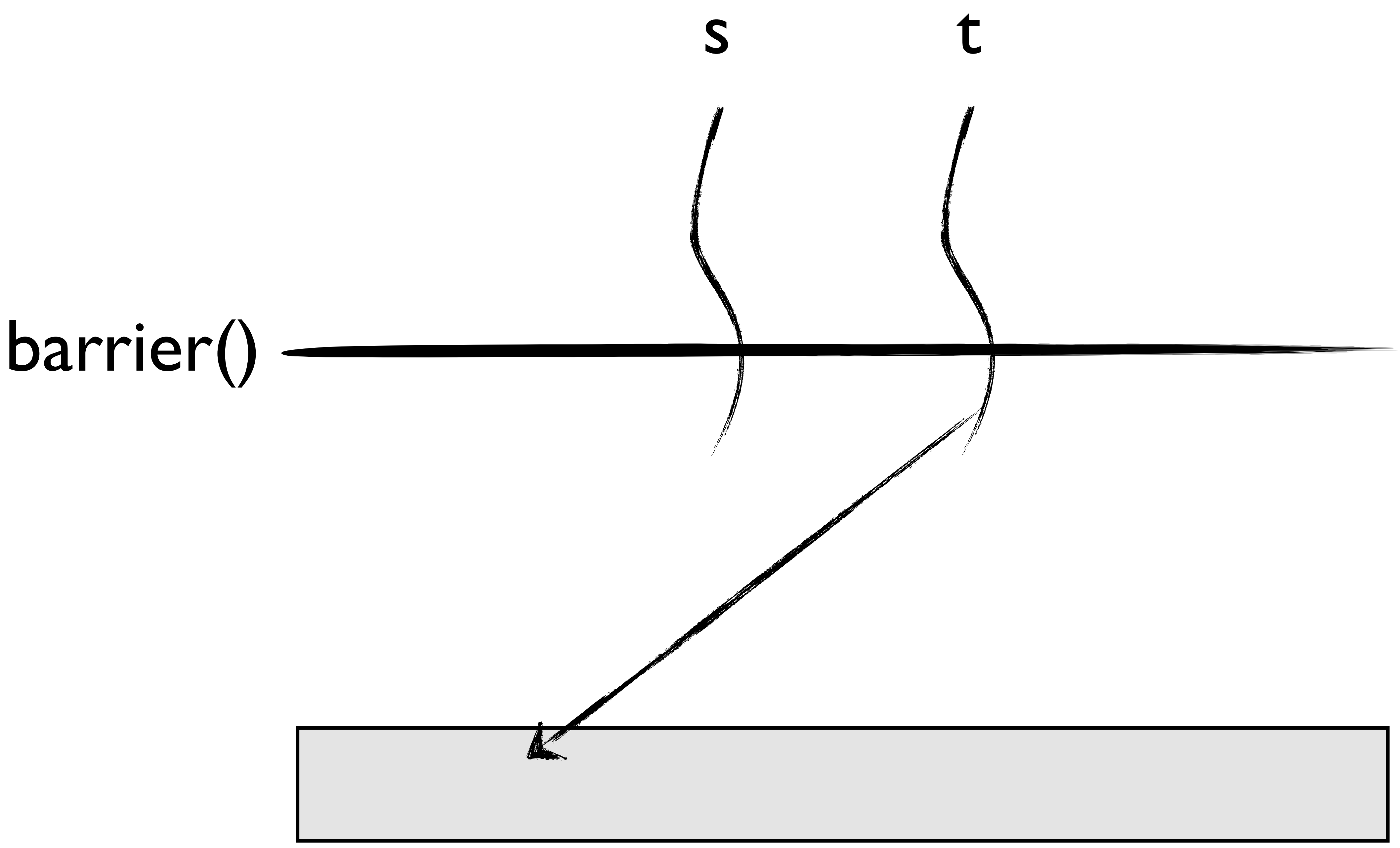



s
t
X


s
t
barrier()



s
t
barrier()


s
t
barrier()

barrier()

=

synchronisation + consistency

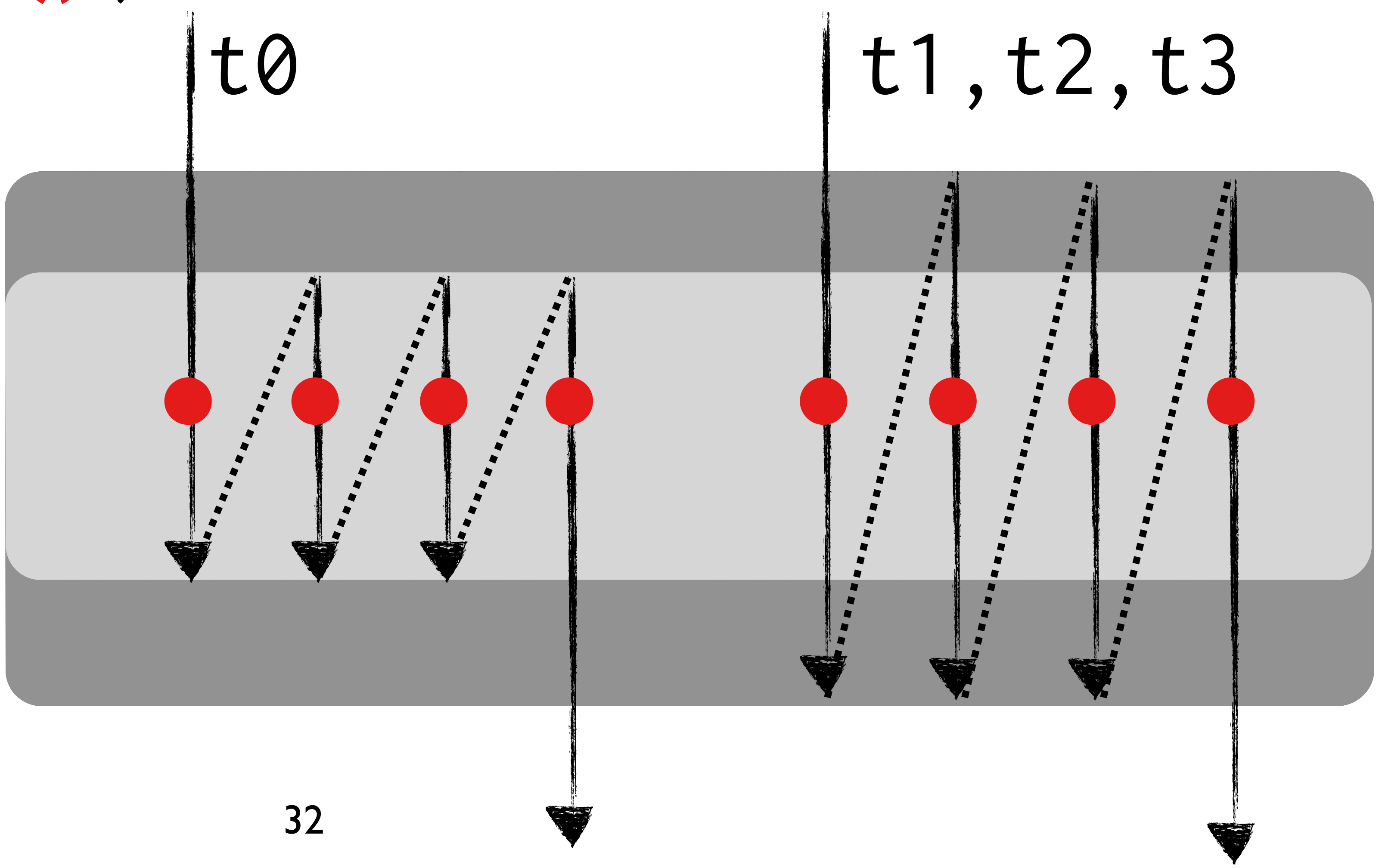
```
__kernel void
add_nbor(__local int *A, int offset) {
  int tid = get_local_id(0);



  A[tid] = A[tid] + A[tid + offset];
}
```

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

```
__kernel void diverge1() {
  int tid = get_local_id(0);
  if (tid == 0) barrier();
  else barrier();
}
```

```
__kernel void diverge2() {
  int x = get_local_id(0) == 0 ? 4 : 1;
  int y = get_local_id(0) == 0 ? 1 : 4;
  for (int i=0; i<x; i++)
    for (int j=0; j<y; j++)
      barrier();
}
```

```
__kernel void diverge2() {
  int x = get_local_id(0) == 0 ? 4 : 1;
  int y = get_local_id(0) == 0 ? 1 : 4;
  for (int i=0; i<x; i++)
    for (int j=0; j<y; j++)
      barrier();
}
```



Summary

- Data races result in non-determinism
- Barrier divergence leads to undefined behaviour
- Bugs of these kind are rarely intended and not benign

Techniques for data race detection

- Dynamic
  - Instrument + Execute
    - GRace
    - cuda-memcheck
  - Symbolic Execution
    - GKLEE
    - KLEE-CL
- Hybrid
  - Execute + Analyse
    - Test Amplification
- Static
  - Transform into SMT problem
    - GPUVerify
    - PUG

Building a static analyser

Essential idea

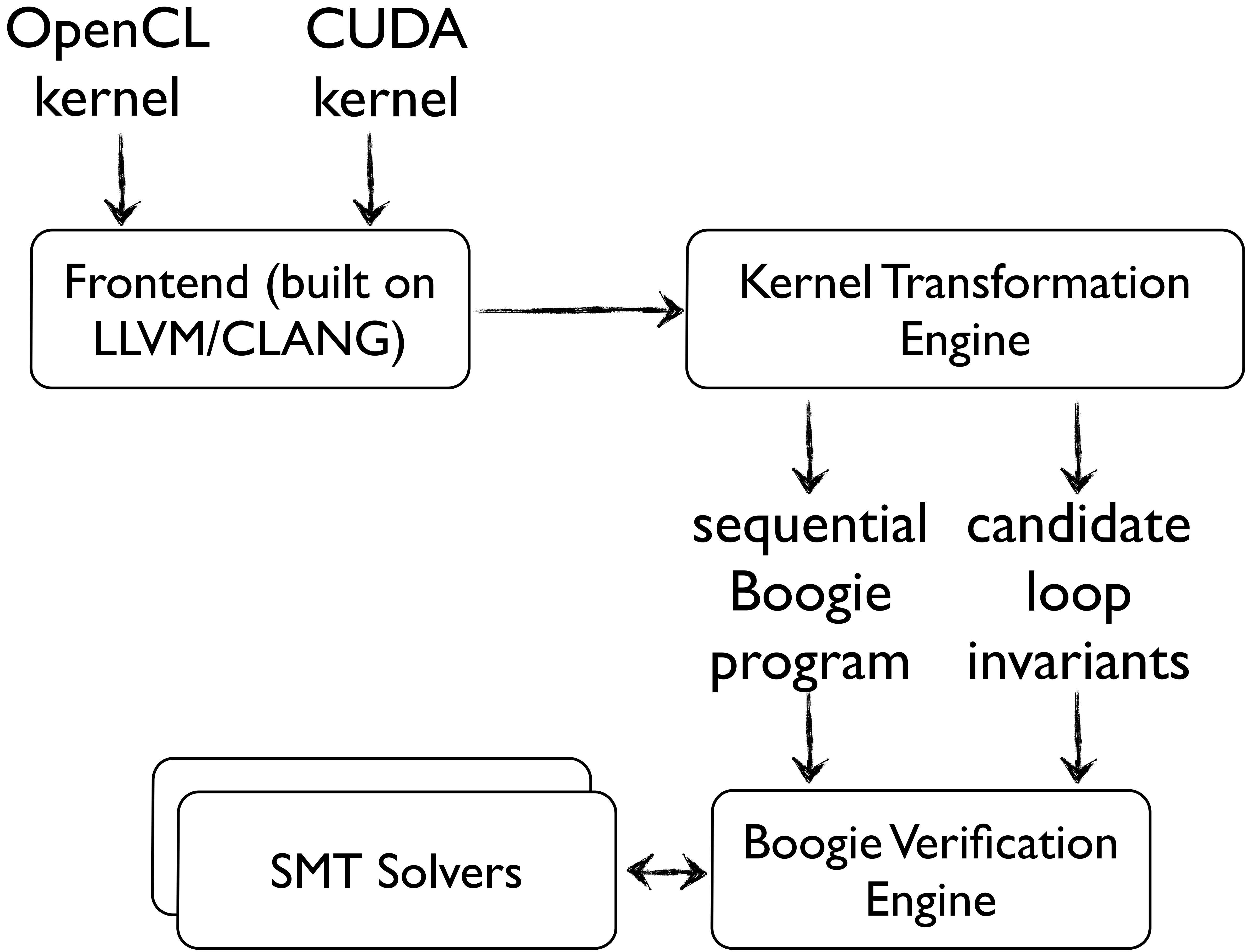
GPU kernel $\xrightarrow{\text{transform}}$ SMT problem

SAT: potential counterexample

UNSAT: could not find any bugs

Satisfiability Modulo Theories

- SMT = SAT solver + Theories
- Decision procedures for
  - Equality and uninterpreted functions
  - Bitvectors
  - Arrays
  - …


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

Boogie

- Intermediate verification language target
- Imperative language with mathematical components
- Generates verification conditions for a variety of solvers
- Used by many static analysers

Invariant inference

- Abstract interpretation
- Predicate abstraction
- Candidate generation and Houdini
- Interpolation
- Abduction

Houdini Algorithm

- A form of predicate abstraction implemented in Boogie
- Takes candidate invariants (guesses) and finds the maximal conjunction of true invariants
- Requires candidate invariant generation for domain
- More predictable than refinement strategies

Summary

- The work of the verification community enables the rapid design and implementation of new tools and techniques
- We live in interesting times!

“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)

Some important observations

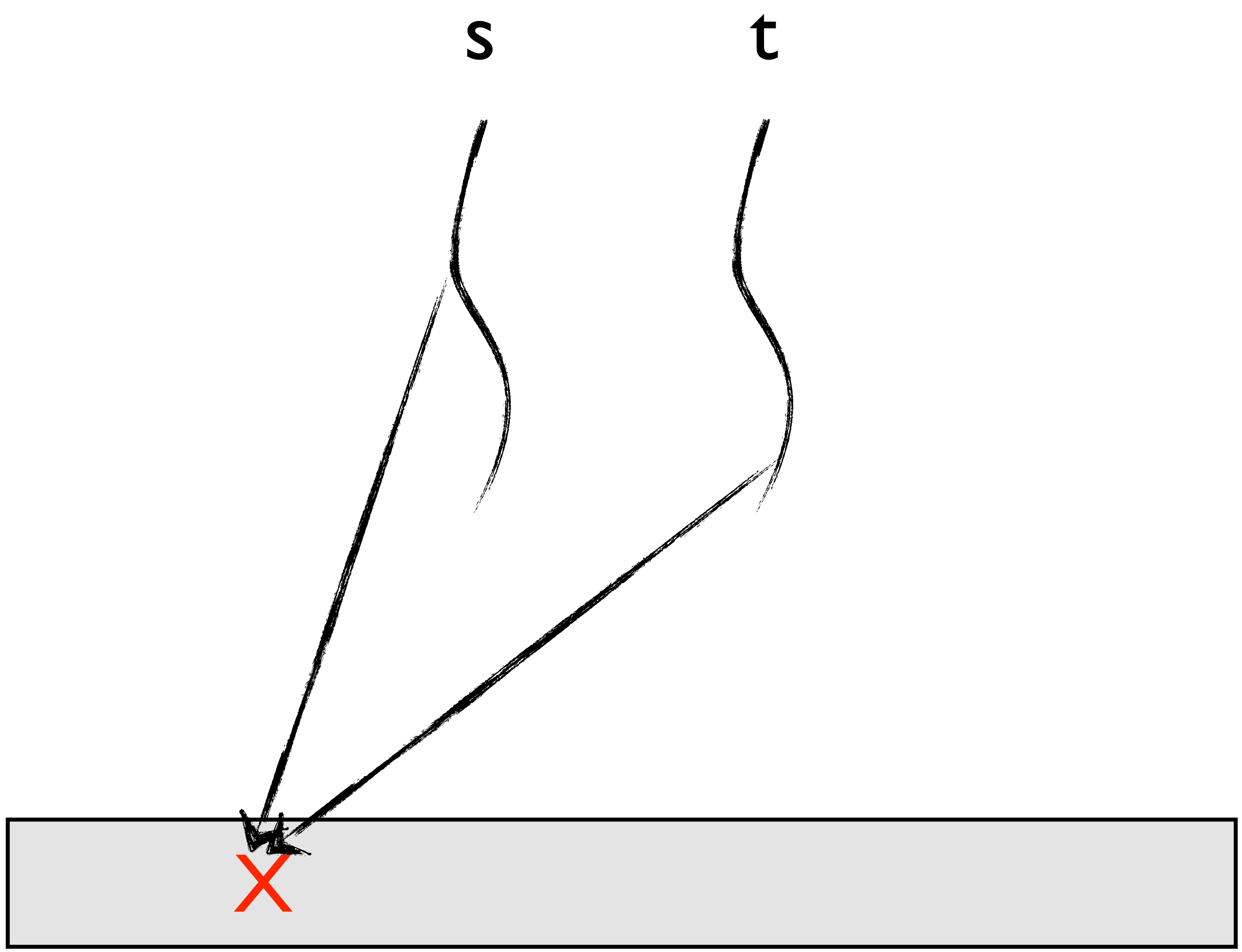
translate

Parallel Kernel K

Sequential Program P

race-freedom

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

```
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

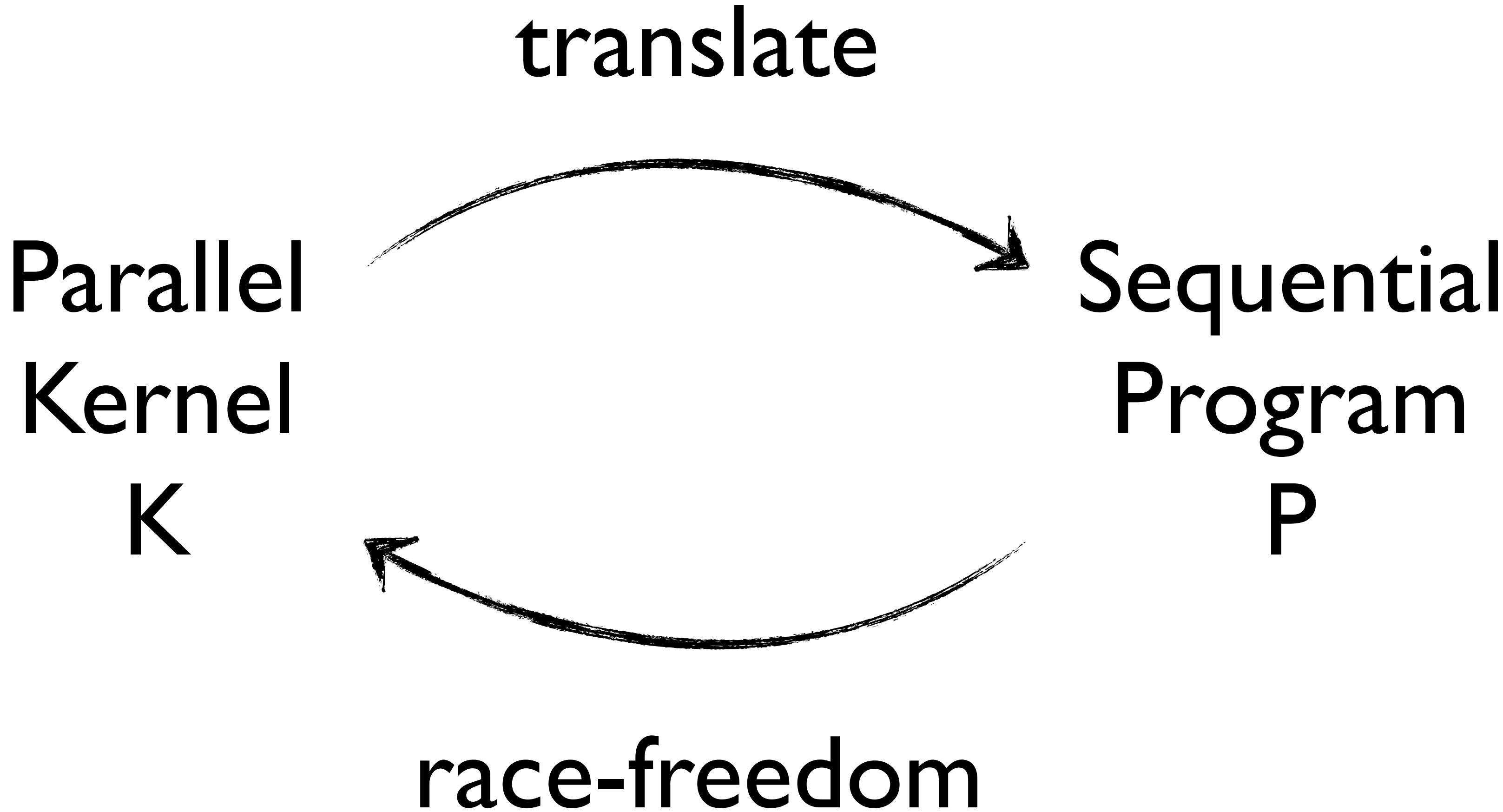
run t from b2 to b3
check all accesses against s

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

Shared state abstraction is necessary for soundness

Break

Kernel transformation for straight-line code


translate
Parallel Kernel K
Sequential Program P
race-freedom

Straight-line code

```
__kernel void
foo( <parameters, including __local arrays> ) {
  <local variable declarations>
  S1;
  S2;
  …
  Sk;
}
```

Form of each statement:

```
x = e
x = A[e]
A[e] = x
barrier()
```

`x` is a local variable

`e` is an expression over local variables and `tid`

`A` is a `__local` array parameter

2 arbitrary threads

The kernel has implicit variable `tid`

Suppose `N` is the total number of threads

Model `tid` by introducing two global variables

```
int tid$1;
int tid$2;
```

and preconditions:

```
\requires 0 <= tid$1 < N;
\requires 0 <= tid$2 < N;
\requires tid$1 != tid$2;
```

Threads in range

Threads different

...but otherwise the threads are arbitrary

Race checking instrumentation

For each `__local` array A introduce four variables

```
bool READ_HAS_OCCURRED_A;
bool WRITE_HAS_OCCURRED_A;
int  READ_OFFSET_A;
int  WRITE_OFFSET_A;
```

and four procedures

```
void LOG_READ_A(int offset);
void LOG_WRITE_A(int offset);
void CHECK_READ_A(int offset);
void CHECK_WRITE_A(int offset);
```

and do not represent A in the sequential program

Example

Kernel

```
__kernel void
foo(__local int *A, int idx) { … }
```

Sequential program

```
int tid$1; int tid$2;
```

2 thread ids

```
bool READ_HAS_OCCURRED_A;
bool WRITE_HAS_OCCURRED_A;
int READ_OFFSET_A;
int WRITE_OFFSET_A;
```

Instrumentation variables for array A

```
// \requires 0 <= tid$1 < N;
// \requires 0 <= tid$2 < N;
// \requires tid$1 != tid$2;
```

Constraining `tid$1` and `tid$2` to be arbitrary, distinct threads

```
void foo(int idx) { … }
```

array A is gone

Dualise local variables

`int x` is duplicated to become:

```
int x$1;
int x$2;
```

This reflects fact that each thread has a copy of `x`

Each non-array parameter `x` is duplicated similarly
Initially equal for threads

```
\requires x$1 == x$2
```

More generally

For an expression e let

`e$1` denotes e with every local variable x replaced by `x$1` and similarly for `e$2`

For example: if e is `a + tid - x` then

`e$2` is `a$2 + tid$2 - x$2`

Translating statements

Stmt	translate(Stmt)
`x = e;`	`x$1 = e$1;` `x$2 = e$2;`

`x = A[e];`	`LOG_READ_A(e$1);`
	`CHECK_READ_A(e$2);`
	`havoc(x$1);`
	`havoc(x$2);`

Log location from which first thread reads

Check read by second thread does not conflict with any prior write by first thread

Over-approximate effect of read by making assigned variables arbitrary

We removed the array A, thus we over-approximate reading from A using havoc. That is, we make no assumptions about the contents of A

Translating statements (2)

Stmt	translate(Stmt)

`A[e] = x;`	`LOG_WRITE_A(e$1);`
	`CHECK_WRITE_A(e$2);`
	`// nothing`

`barrier();`	`barrier();`
`S;` `T;`	`translate(S);` `translate(T);`

Log location to which the first thread writes

Check write by the second thread does not conflict with any prior read or write by the first thread

The write itself has no effect because the array `A` has been removed

We shall give `barrier()` a special meaning in the translated program

Example

```
__kernel void
foo(__local int *A,
    int idx) {
  int x;
  int y;

  x = A[tid + idx];

  y = A[tid];

  A[tid] = x + y;

}
```

```
// \requires 0 <= tid$1 < N;
// \requires 0 <= tid$2 < N;
// \requires tid$1 != tid$2;
// \requires idx$1 == idx$2;
__kernel void
foo(
    int idx$1; int idx$2;) {
  int x$1; int x$2;
  int y$1; int y$2;

  LOG_READ_A(tid$1 + idx$1);
  CHECK_READ_A(tid$2 + idx$2);
  havoc(x$1); havoc(x$2);

  LOG_READ_A(tid$1);
  CHECK_READ_A(tid$2);
  havoc(y$1); havoc(y$2);

  LOG_WRITE_A(tid$1);
  CHECK_WRITE_A(tid$2);
}
```

LOG_READ_A

```
void LOG_READ_A(int offset) {
  if (*) {
    READ_HAS_OCCURRED_A = true;
    READ_OFFSET_A = offset;
  }
}
```

LOG_WRITE_A

```
void LOG_WRITE_A(int offset) {
  if (*) {
    WRITE_HAS_OCCURRED_A = true;
    WRITE_OFFSET_A = offset;
  }
}
```

CHECK_READ_A

```
void CHECK_READ_A(int offset) {
  assert(WRITE_HAS_OCCURRED_A =>
           WRITE_OFFSET_A != offset);
}
```

CHECK_WRITE_A

```
void CHECK_WRITE_A(int offset) {
  assert(WRITE_HAS_OCCURRED_A =>
         WRITE_OFFSET_A != offset);
  assert(READ_HAS_OCCURRED_A =>
         READ_OFFSET_A != offset);
}
```

Precondition

```
// \requires 0 <= tid$1 < N;
// \requires 0 <= tid$2 < N;
// \requires tid$1 != tid$2;
// \requires idx$1 == idx$2;
// \requires !READ_HAS_OCCURRED_A;
// \requires !WRITE_HAS_OCCURRED_A;
void foo(int idx$1, int idx$2) {
  int x$1; int x$2;
  int y$1; int y$2;
  LOG_READ_A(tid$1 + idx$1);
  CHECK_READ_A(tid$2 + idx$2);
  havoc(x$1); havoc(x$2);
  LOG_READ_A(tid$1);
  CHECK_READ_A(tid$2);
```

...

```
// \requires 0 <= tid$1 < N;
// \requires 0 <= tid$2 < N;
// \requires tid$1 != tid$2;
// \requires idx$1 == idx$2;
// \requires !READ_HAS_OCCURRED_A;
// \requires !WRITE_HAS_OCCURRED_A;
__kernel void
foo(int x$1; int x$2;) {
  int x$1; int x$2;
  int y$1; int y$2;

  LOG_READ_A(tid$1 + idx$1);
  CHECK_READ_A(tid$2 + idx$2);
  havoc(x$1); havoc(x$2);

  LOG_READ_A(tid$1);
  CHECK_READ_A(tid$2);
  havoc(y$1); havoc(y$2);

  LOG_WRITE_A(tid$1);
  CHECK_WRITE_A(tid$2);
}
```

```
// \requires 0 <= tid$1 < N;
// \requires 0 <= tid$2 < N;
// \requires tid$1 != tid$2;
// \requires idx$1 == idx$2;
// \requires !READ_HAS_OCCURRED_A;
// \requires !WRITE_HAS_OCCURRED_A;
void foo(int idx$1, int idx$2) {
  LOG_READ_A(tid$1 + idx$1);
  CHECK_READ_A(tid$2 + idx$2);
  LOG_READ_A(tid$1);
  CHECK_READ_A(tid$2);
  LOG_WRITE_A(tid$1);
  CHECK_WRITE_A(tid$2);
}
```

```
// other preconditions same as before
// \requires !READ_HAS_OCCURRED_A;
// \requires !WRITE_HAS_OCCURRED_A;
void foo(int idx$1, int idx$2) {
  // LOG_READ_A(tid$1 + idx$1);
  if (*) { READ_HAS_OCCURRED_A = true;
           READ_OFFSET_A = tid$1 + idx$1; }
  // CHECK_READ_A(tid$2 + idx$2);
  assert(WRITE_HAS_OCCURRED_A => WRIT
  // LOG_READ_A(tid$1);
  if (*) { READ_HAS
  
  
          RRED_A => WRITE_OFFSET_A != tid$2);
         _A(tid$1);
      ) { WRITE_HAS_OCCURRED_A = true;
           WRITE_OFFSET_A = tid$1; }
  // CHECK_WRITE_A(tid$2);
  assert(WRITE_HAS_OCCURRED_A => WRITE_OFFSET_A != tid$2);
  assert(READ_HAS_OCCURRED_A => READ_OFFSET_A != tid$2);
}
```

Nondeterminism ensures that some program execution checks every pair of potentially racing operations

Possible race checked by choosing to log the read, then executing the assert. In this case there is no race

```
// other preconditions same as before
// \requires !READ_HAS_OCCURRED_A;
// \requires !WRITE_HAS_OCCURRED_A;
void foo(int idx$1, int idx$2) {
  // LOG_READ_A(tid$1 + idx$1);
  if (*) { READ_HAS_OCCURRED_A = true;
           READ_OFFSET_A = tid$1 + idx$1; }
  // CHECK_READ_A(tid$2 + idx$2);
  assert(WRITE_HAS_OCCURRED_A => WRITE_OFFSET_A != tid$2 + idx$2);
  // LOG_READ_A(tid$1);
  if (*) { READ_HAS_OCCURRED_A = true;
           READ_OFFSET_A = tid$1; }
  // CHECK_READ_A(tid$2);
  assert(WRITE_HAS_OCCURRED_A => WRITE_OFFSET_A != tid$2);
  // LOG_WRITE_A(tid$1);
  if (*) { WRITE_HAS_OCCURRED_A = true;
           WRITE_OFFSET_A = tid$1; }
  // CHECK_WRITE_A(tid$2);
  assert(WRITE_HAS_OCCURRED_A => WRITE_OFFSET_A != tid$2);
  assert(READ_HAS_OCCURRED_A => READ_OFFSET_A != tid$2);
}
```

In this case a potential race is detected

```
// other preconditions same as before
// \requires !READ_HAS_OCCURRED_A;
// \requires !WRITE_HAS_OCCURRED_A;
void foo(int idx$1, int idx$2) {
  // LOG_READ_A(tid$1 + idx$1);
  if (*) { READ_HAS_OCCURRED_A = true;
           READ_OFFSET_A = tid$1 + idx$1; }
  // CHECK_READ_A(tid$2 + idx$2);
  assert(WRITE_HAS_OCCURRED_A => WRITE_OFFSET_A != tid$2 + idx$2);
  // LOG_READ_A(tid$1);
  if (*) { READ_HAS_OCCURRED_A = true;
           READ_OFFSET_A = tid$1; }
  // CHECK_READ_A(tid$2);
  assert(WRITE_HAS_OCCURRED_A => WRITE_OFFSET_A != tid$2);
  // LOG_WRITE_A(tid$1);
  if (*) { WRITE_HAS_OCCURRED_A = true;
           WRITE_OFFSET_A = tid$1; }
  // CHECK_WRITE_A(tid$2);
  assert(WRITE_HAS_OCCURRED_A => WRITE_OFFSET_A != tid$2);
  assert(READ_HAS_OCCURRED_A => READ_OFFSET_A != tid$2);
}
```

Barrier

```
void barrier() {
  assume(!READ_HAS_OCCURRED_A);
  assume(!WRITE_HAS_OCCURRED_A);
  // Do this for every array
}
```

Summary

For each array A

- Introduce instrumentation variables
- Generate log and check procedures
- Remove array parameter

For each statement

- Generate corresponding statements
- Interleave two arbitrary threads using round-robin schedule

Loops and conditionals

Predicated execution

```
if (x < 100) {
  x = x + 1;
} else {
  y = y + 1;
}
```

predicated →

```
P = (x < 100);
Q = !(x < 100);

x = (P ? x + 1 : x);
y = (Q ? y + 1 : y);
```

Encoding loops and conditionals

- Apply predication so that every execution point has a predicate determining whether a thread is enabled
- Add predication to logging and checking procedures and barrier

Statements with predication

Stmt	translate(Stmt, P)
`x = e;`	`x$1 = P$1 ? e$1 : x$1;` `x$2 = P$2 ? e$2 : x$2;`

`x = A[e];`	`LOG_READ_A(P$1, e$1);`
	`CHECK_READ_A(P$2, e$2);`
	`x$1 = P$1 ? * : x$1;`
	`x$2 = P$2 ? * : x$2;`

`A[e] = x;`	`LOG_WRITE_A(P$1, e$1);` `CHECK_WRITE_A(P$2, e$2);`

LOG and CHECK calls take predicate as parameter

We only havoc `x$1` and `x$2` if `P$1` and `P$2` are true, respectively

LOG and CHECK calls take predicate as parameter

Generating predicates

Stmt	translate(Stmt, P)

`if (e) {`
`  S;`
`} else {`
`  T;`
`}`
	`Q$1 = P$1 && e$1;`
	`Q$2 = P$2 && e$2;`
	`R$1 = P$1 && !e$1;`
	`R$2 = P$2 && !e$2;`
	`translate(S, Q);`
	`translate(T, R);`

`while (e) {`
`  S;`
`}`
	`Q$1 = P$1 && e$1;`
	`Q$2 = P$2 && e$2;`
	`while (Q$1 || Q$2) {`
	`  translate(S, Q);`
	`  Q$1 = Q$1 && e$1;`
	`  Q$2 = Q$2 && e$2;`
	`}`

`Q` and `R` are fresh

Code for both threads becomes predicated

Threads compute loop guard into predicate

Loop until both threads are done

Translate loop body using loop predicate

Re-evaluate loop guard

Statements with Predicates

Stmt	translate(Stmt, P)
`S;` `T;`	`translate(S, P);` `translate(T, P);`
`barrier();`	`barrier(P$1, P$2);`

barrier now takes parameters determining whether the threads are enabled

Predicated LOG_READ_A

```
void LOG_READ_A(bool enabled, int offset) {
  if(enabled) {
    if(*) {
      READ_HAS_OCCURRED_A = true;
      READ_OFFSET_A = offset;
    }
  }
}
```

Predicated CHECK_WRITE_A

```
void CHECK_WRITE_A(bool enabled, int offset) {
  assert(enabled && WRITE_HAS_OCCURRED_A =>
         WRITE_OFFSET_A != offset);
  assert(enabled && READ_HAS_OCCURRED_A =>
         READ_OFFSET_A != offset);
}
```

Predicated barrier

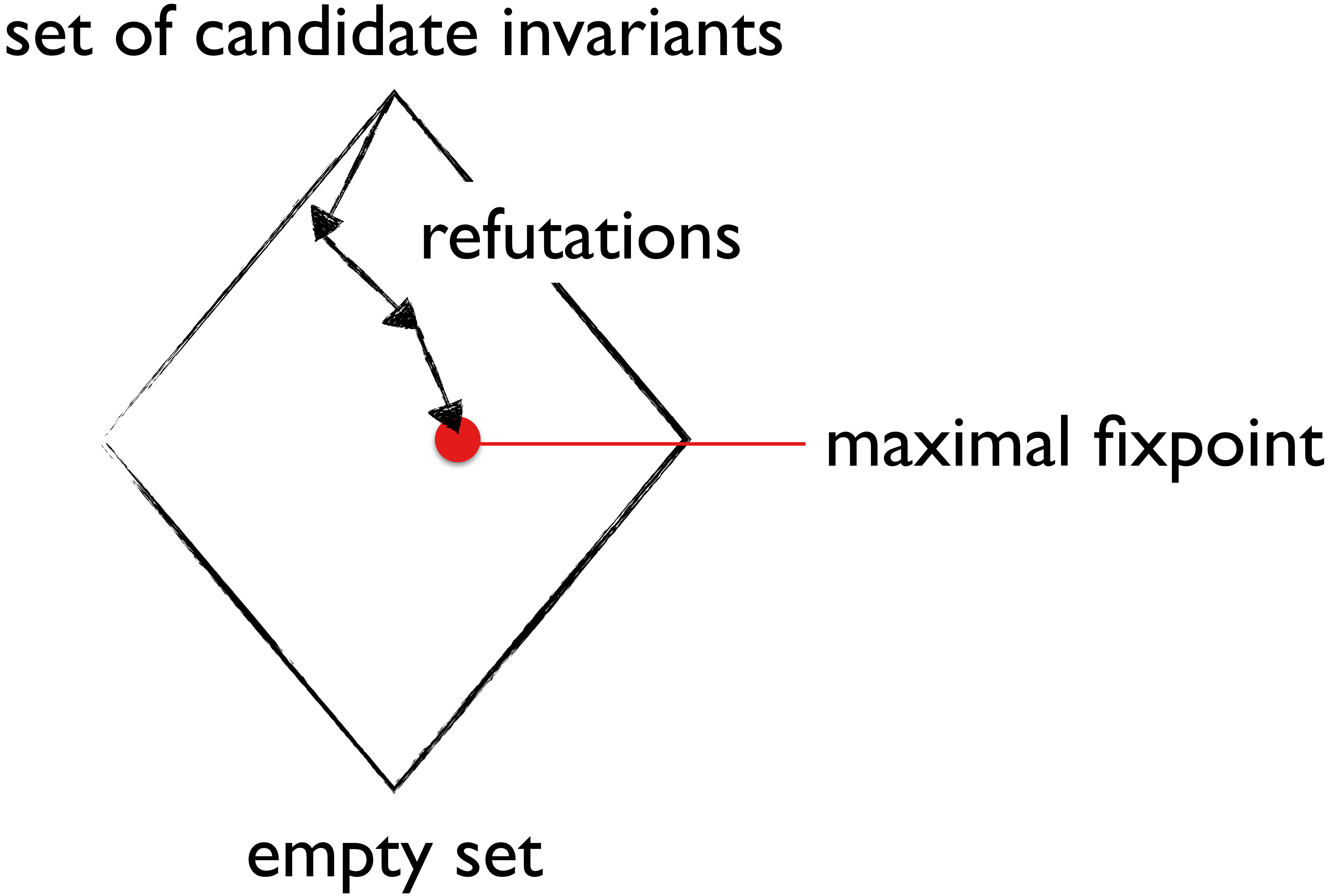
```
void barrier(bool enabled$1, bool enabled$2) {

  assert(enabled$1 == enabled$2);

  if(!enabled$1) {
    return;
  }

  // As before:
  assume(!READ_HAS_OCCURRED_A);
  assume(!WRITE_HAS_OCCURRED_A);
  // Do this for every array
}
```

The threads must agree on whether they are enabled – otherwise we have barrier divergence

barrier does nothing if the threads are not enabled

Loop invariants (optional)

Houdini


set of candidate invariants
refutations
maximal fixpoint
empty set

```
__kernel void
stride(__local int *A, uint n) {
  uint tid = get_local_id(0);
  uint size = get_group_size(0);
  for (uint i=tid; i<n; i+=size) {
    A[i] = val;
  }
}
```

0	1	2	3	4	5	6	7
T0	T1	T2	T3	T0	T1	T2	T3

```
__kernel void
stride(__local int *A, uint n) {

  uint tid = get_local_id(0);
  uint size = get_group_size(0);

  for (uint i=tid; i<n; i+=size) {
    A[i] = val;
  }
}
```

0	1	2	3	4	5	6	7
T0	T1	T2	T3	T0	T1	T2	T3

```
__write(A) ==>
  __offset(A) % size == tid
```

```
__kernel void
slice(__local int *A, uint m) {
  uint tid = get_local_id(0);
  for (uint i=0; i<m; i++) {
    A[tid*m+i] = val;
  }
}
```

0	1	2	3	4	5	6	7
T0	T0	T0	T0	T1	T1	T1	T1

```
__kernel void
slice(__local int *A, uint m) {

  uint tid = get_local_id(0);

  for (uint i=0; i<m; i++) {
    A[tid*m+i] = val;
  }
}
```

0	1	2	3	4	5	6	7
T0	T0	T0	T0	T1	T1	T1	T1

```
__write(A) ==>
  tid*m <= __offset(A) < (tid+1)*m
```

```
__kernel void
slice(__local int *A, uint m) {
  uint tid = get_local_id(0);
  for (uint i=0; i<m; i++) {
    A[tid*m+i] = val;
  }
}
```

0	1	2	3	4	5	6	7
T0	T0	T0	T0	T1	T1	T1	T1

```
__write(A) ==>
  __offset(A) / m == tid
```

Summary

- Using the Houdini algorithm avoids having to design abstract domains or search for invariants
- Instead we depend on heuristics for generating candidate invariants

Close

Show how the *theory, algorithms and tools* of the verification community can be brought to bear on an interesting and important domain

What we covered

- GPUs and their programming models
- The need for semantics and verification tools
- Building a static analyser with off-the-shelf parts
- Examining the GPUVerify verification method

What we didn’t cover

- Multidimensional groups
- Inter-group race checking with global arrays
- Benign data races
- Atomics and warp synchronisation
- Data-dependent kernels

GPUVerify on YouTube

GPUVerify contributors

Alastair F. Donaldson

Shaz Qadeer

- Ethel Bardsley
- Adam Betts
- Nathan Chong
- Peter Collingbourne
- Pantazis Deligiannis
- Jeroen Ketema
- Egor Kyshtymov
- Daniel Liew
- Paul Thomson
- John Wickerson

EU FP7 STREP project CARP (project number 287767)
EPSRC PSL project (EP/I006761/1)
EPSRC First Grant (EP/K011499/1)