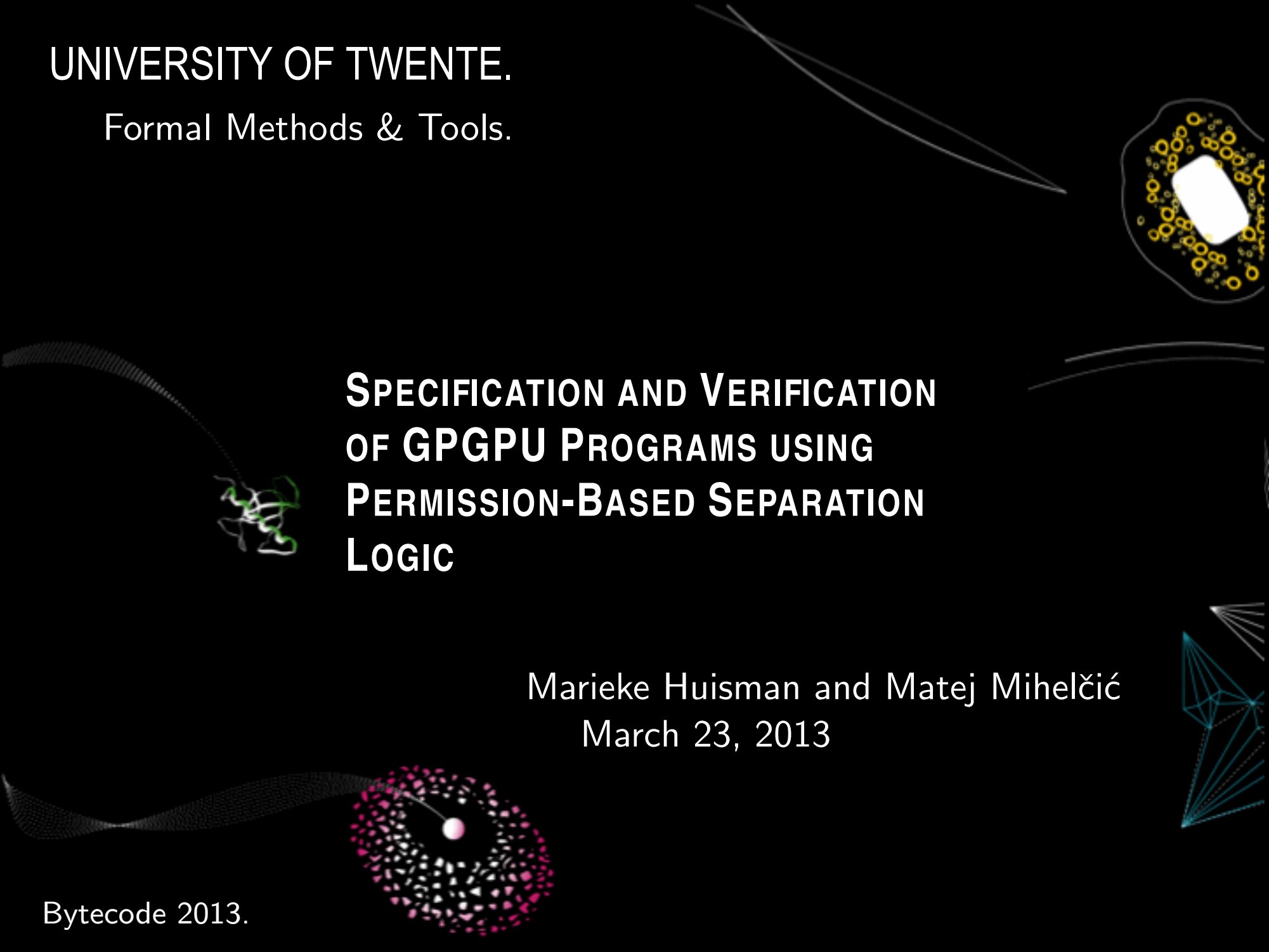
UNIVERSITY OF TWENTE.

Formal Methods & Tools.



Specification and Verification of GPGPU Programs using Permission-Based Separation Logic

Marieke Huisman and Matej Mihelčić

March 23, 2013

Bytecode 2013.

Graphics Processing Units (GPUs):

- specialized electronic circuits
- rapidly manipulate and alter memory
- accelerate the building of images intended for output to a display

- Graphics Processing Units (GPUs) are increasingly used for general-purpose applications
- Used in media processing, medical imaging, eye-tracking etc.
- Urgent need for verification techniques of accelerator software
- Safety is critical in applications like medical imaging: incorrect imaging results could lead indirectly to loss of life.
- Software bugs in media processing domains can have drastic financial implications.

Two main programming frameworks:

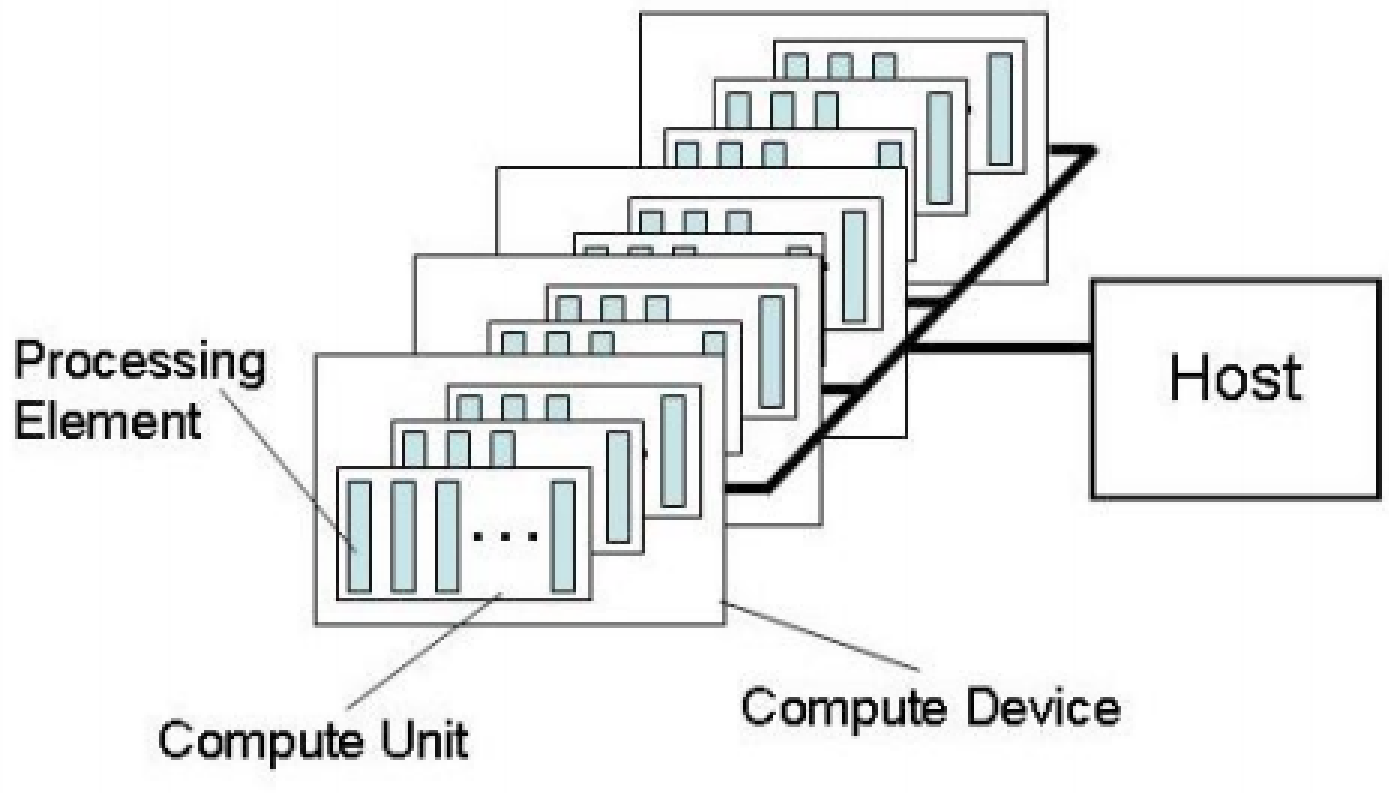
CUDA:

- Parallel computing platform by NVIDIA
- CUDA-enabled NVIDIA gpu's

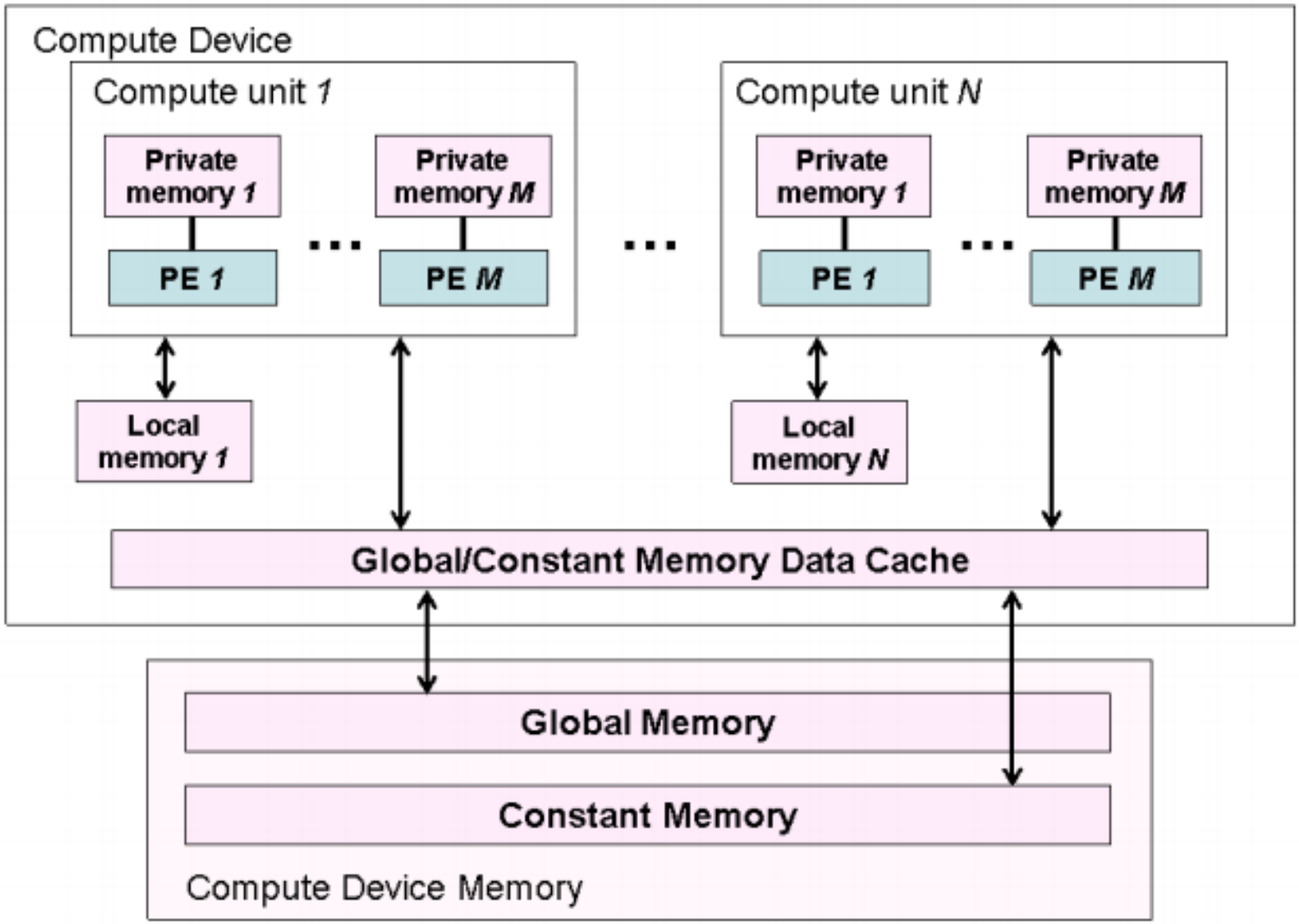
OpenCl:

- Framework for writing programs for heterogeneous platforms by the Khronos group
- Support for Intel, AMD cpu's and NVIDIA, ATI gpu's, ARM processors

OpenCL model:



Memory and computation model:



Verification approach and challenges

- Logic based verification approach
- Challenges:
  - Reasoning about hundreds, even thousands of parallel threads
  - Complex memory and execution model
  - Reasoning about barriers (the main synchronization mechanism)

Permission-based Separation logic

- Main mechanism used in our verification approach
- Separation logic developed as an extension of Hoare logic
- Convenient to reason modularly about concurrent programs
- To reason about shared resources, numerical fractions (permissions) denoting access rights to shared locations are added to the logic
- A full permission 1 denotes a write permission, whereas any fraction in the interval $< 0, 1]$ denotes a read permission

Motivating example:

```
__kernel void example(__global int *a) {
    int tid = get_global_id(0);
    a[tid]=tid;
                }
```

- Simple OpenCL kernel function example
- Represents one thread execution
- Parametrized by global *tid* or local *ltid*
- Number of threads and groups running the kernel defined in the host program
- Currently we have no information about the number of threads or the input data

Motivating example:

Solution:

Add the kernel specification

Kernel spec:

(resources: $*_{i \in [0 \ldots size-1]}$ Perm($a[i]$, 1),
precondition: $size = n \wedge numthreads = n$, postcondition: true)

```
__kernel void example(__global int *a) {
    int tid = get_global_id(0);
    a[tid]=tid;
                }
```

Gain information about the number of threads and the size of the input array

Gain information about kernel access permissions to this array

Motivating example:

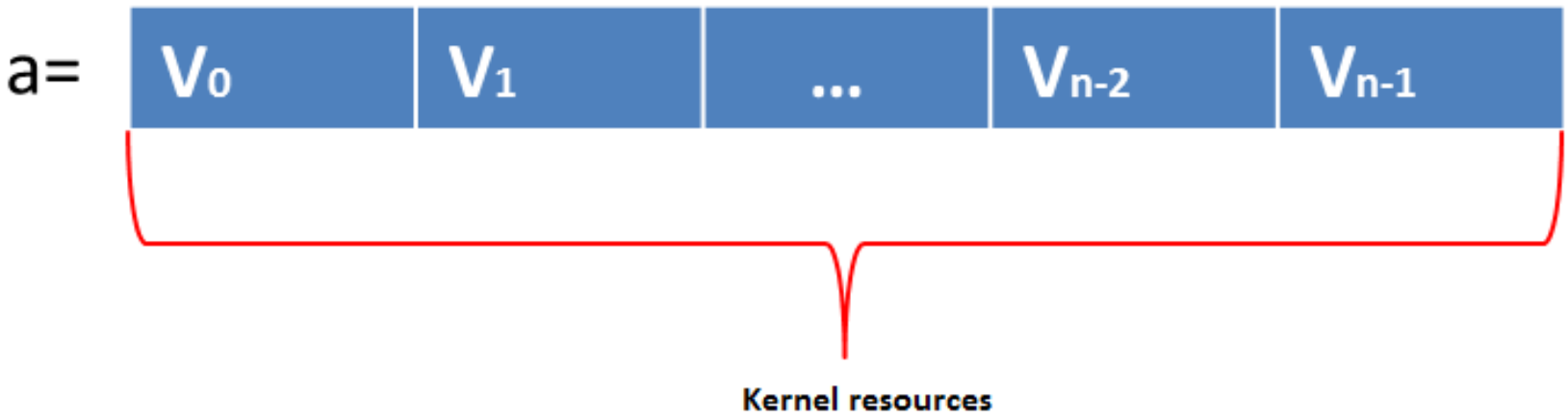


Figure : Kernel has access permission 1 for each field in the input array *a*

Motivating example:

- We need to distribute kernel permissions to individual threads
- We do this with the thread specification.

Kernel spec:

(resources: $*_{i \in [0 \ldots size-1]}$ Perm$(a[i], 1)$,
precondition: $size = n \wedge numthreads = n$, postcondition: true)

Thread spec:
(resources: Perm$(a[tid], 1)$, precondition: true,
postcondition: true)

```
__kernel void example(__global int *a) {
    int tid = get_global_id(0);
    a[tid]=tid;
                }
```

Motivating example:

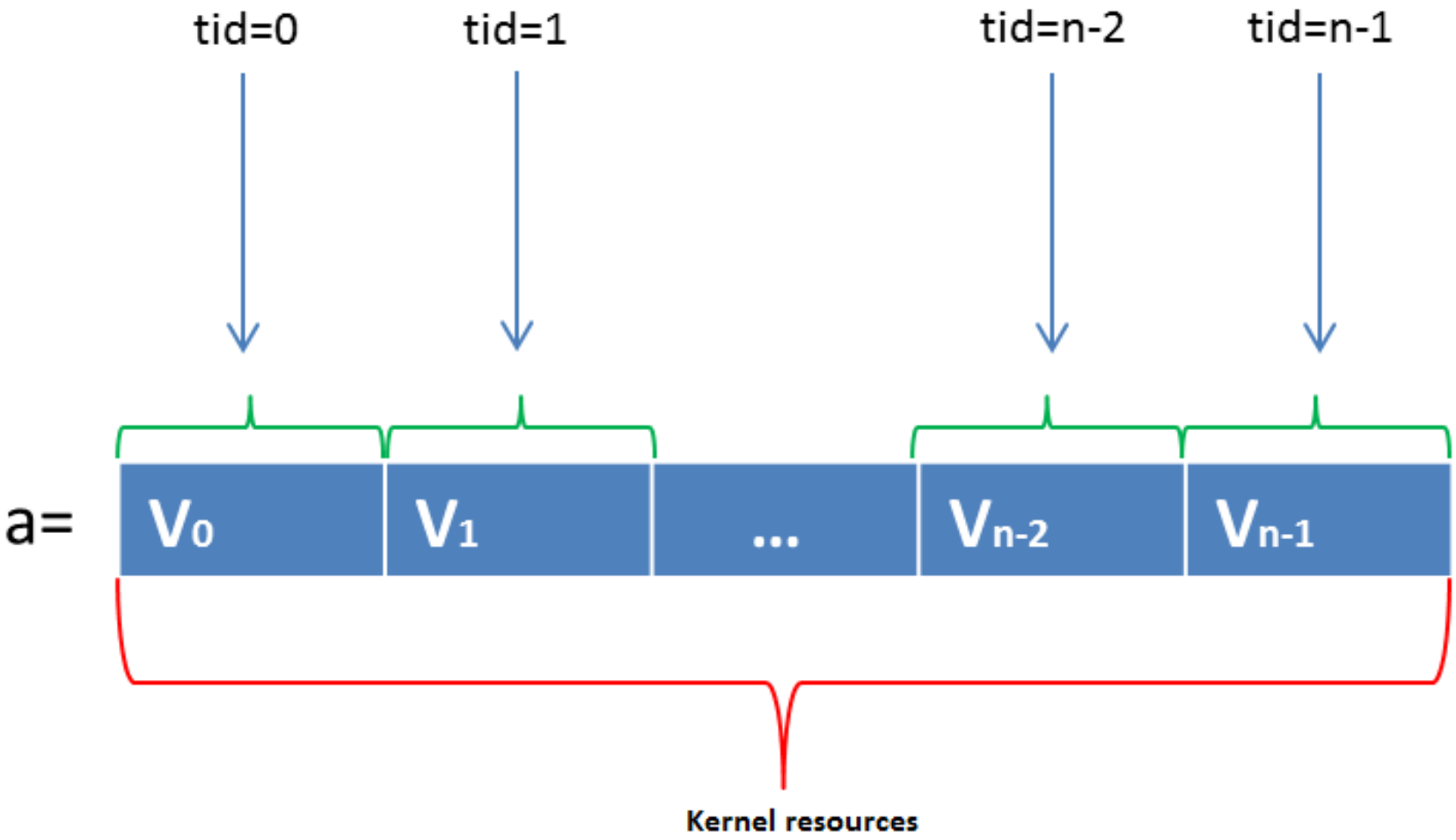


Figure : Thread with id *tid* has access permission 1 for the element $a[tid]$

Motivating example:

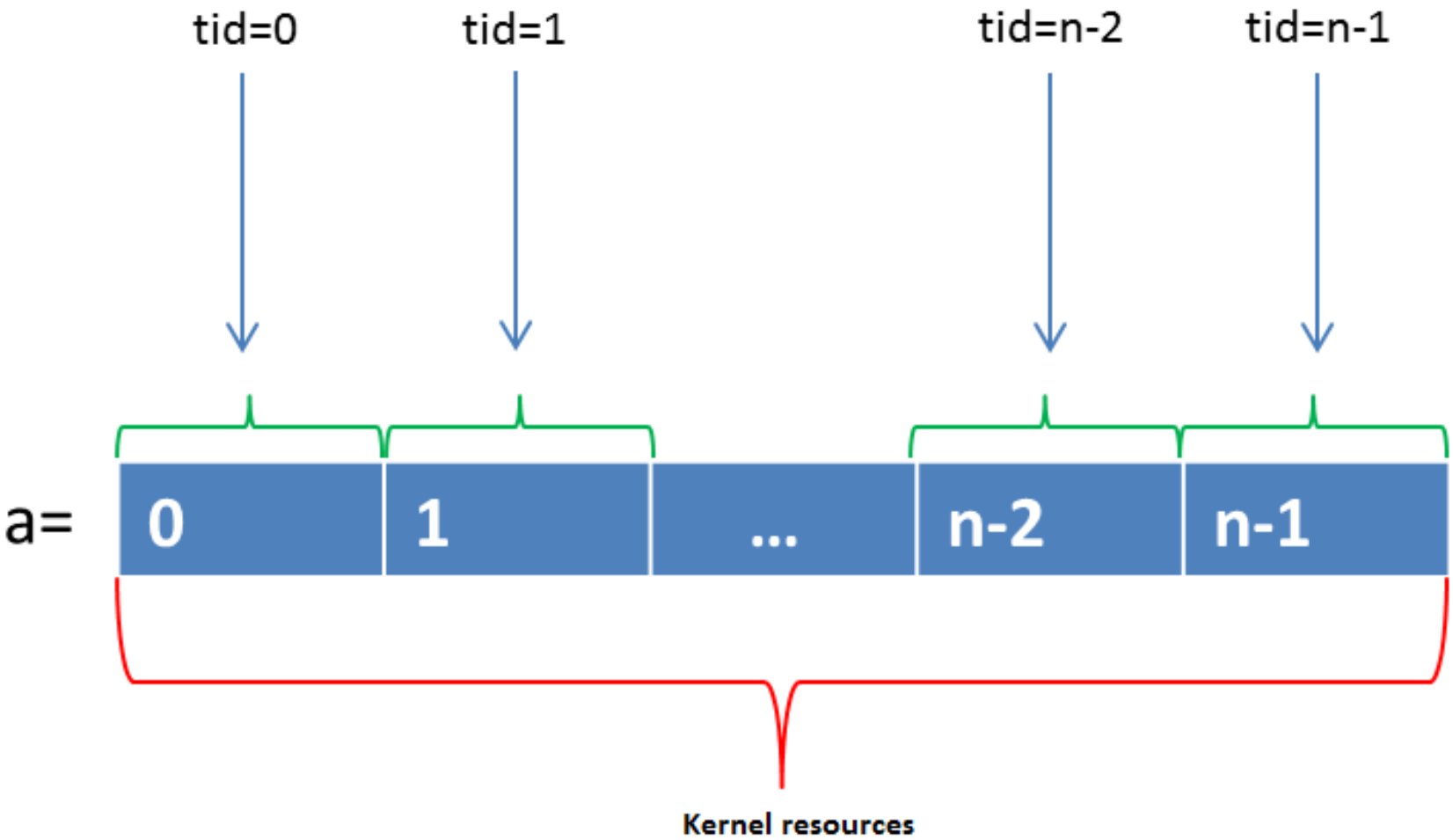


Figure : Array after the kernel execution

Verification of GPU kernels:

The verification is performed in several steps:

1. The kernel resources are shown to be sufficient for the thread specification
   - $$K_{res} \& K_{pre} \mathrel{-\!\!*} \ast_{tid \in Tid} (T_{res|glob} \& T_{pre})$$
   - $$\ast_{v \in Local} \texttt{Perm}(v, 1) \mathrel{-\!\!*} \ast_{ltid \in LTid} T_{res|loc}$$
2. Single thread execution is verified using standard logic rules

3. Each barrier with a memory fence on global memory, redistributes only the permissions that are available in the kernel

$$K_{res} \mathrel{-\!\!*} \ast_{tid \in Tid} B_{res|glob}$$

4. For each barrier with a global memory fence, its postcondition follows from the precondition (over all threads).

$$G_{res} \& _{tid \in Tid} B_{pre} \mathrel{-\!\!*} \& _{tid \in Tid} B_{post} \big|_{RGPerm(tid)}$$

Kernel specification examples:

Kernel spec:

(resources: $*_{i \in [0 \ldots size-1]}$ Perm($a[i]$, 1),
precondition: $size = n \wedge numthreads = n$, postcondition: true)

Thread spec:

(resources: Perm($a[tid]$, 1), precondition: true,
postcondition: true)

```
__kernel void example(__global int *a, __global int *b) {
    int tid = get_global_id(0);
    a[tid]=tid;
    a[(tid+1)%size]=a[(tid+1)%size]+1; }
```

Barrier usage:

Kernel spec:

$(\underline{\texttt{resources:}} \ * \ _{i\in[0\ldots size-1]} \ \texttt{Perm}(a[i],1),$
$\underline{\texttt{precondition:}} \ size = n \wedge numthreads = n, \underline{\texttt{postcondition:}} \ \text{true})$

Thread spec:

$(\underline{\texttt{resources:}} \ \texttt{Perm}(a[tid],1), \underline{\texttt{precondition:}} \ \text{true},$
$\underline{\texttt{postcondition:}} \ \text{true})$

```
__kernel void example(__global int *a) {
    int tid = get_global_id(0);
    a[tid]=tid;
    barrier(CLK_GLOBAL_MEM_FENCE); //B
    a[(tid+1)%size]=a[(tid+1)%size]+1;
    }
```

Barrier spec(B) : $(\texttt{Perm}(a[(tid+1)\%size],1), \text{true}, \text{true})$

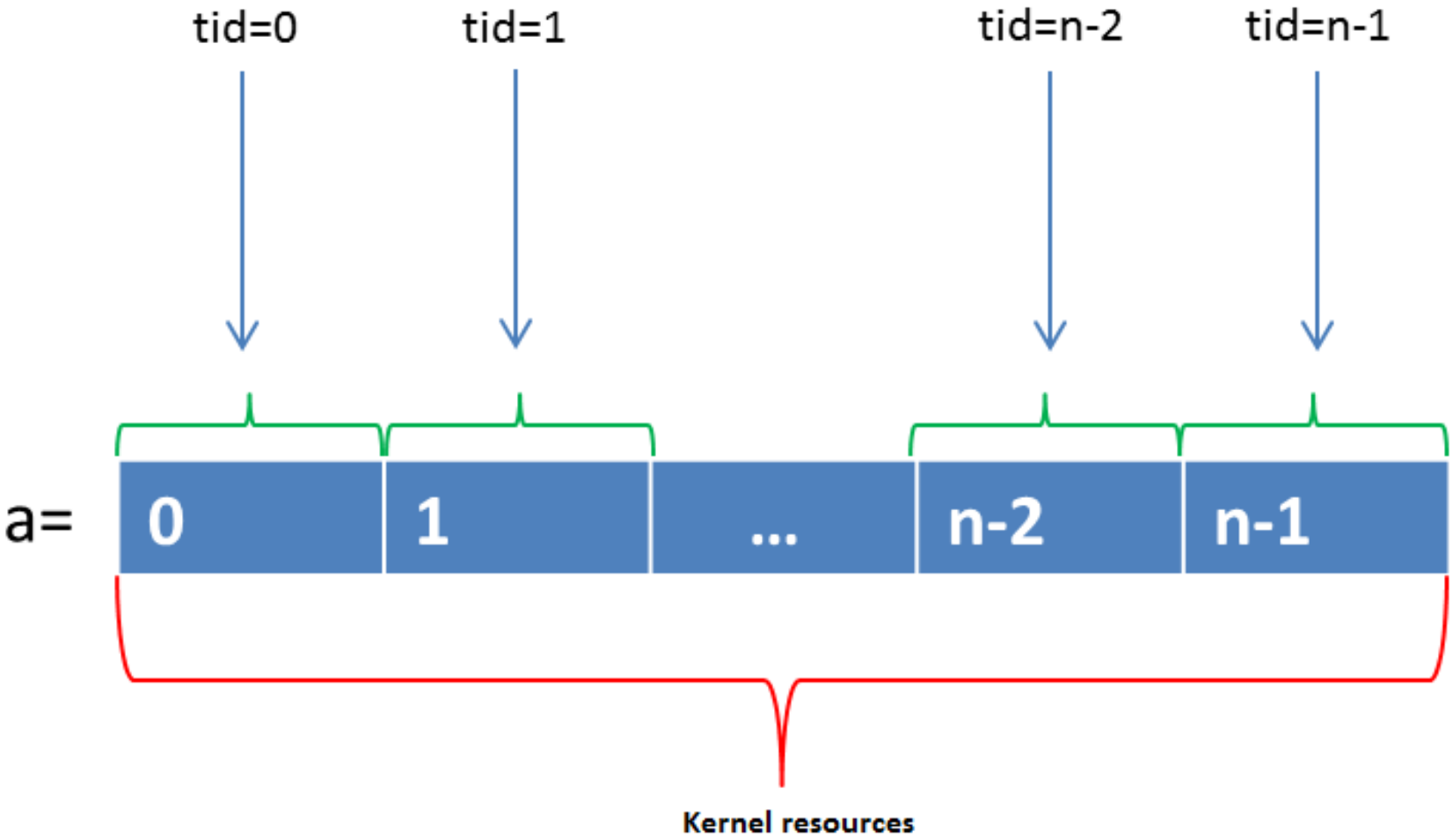


Figure : Array at the moment threads entered the barrier

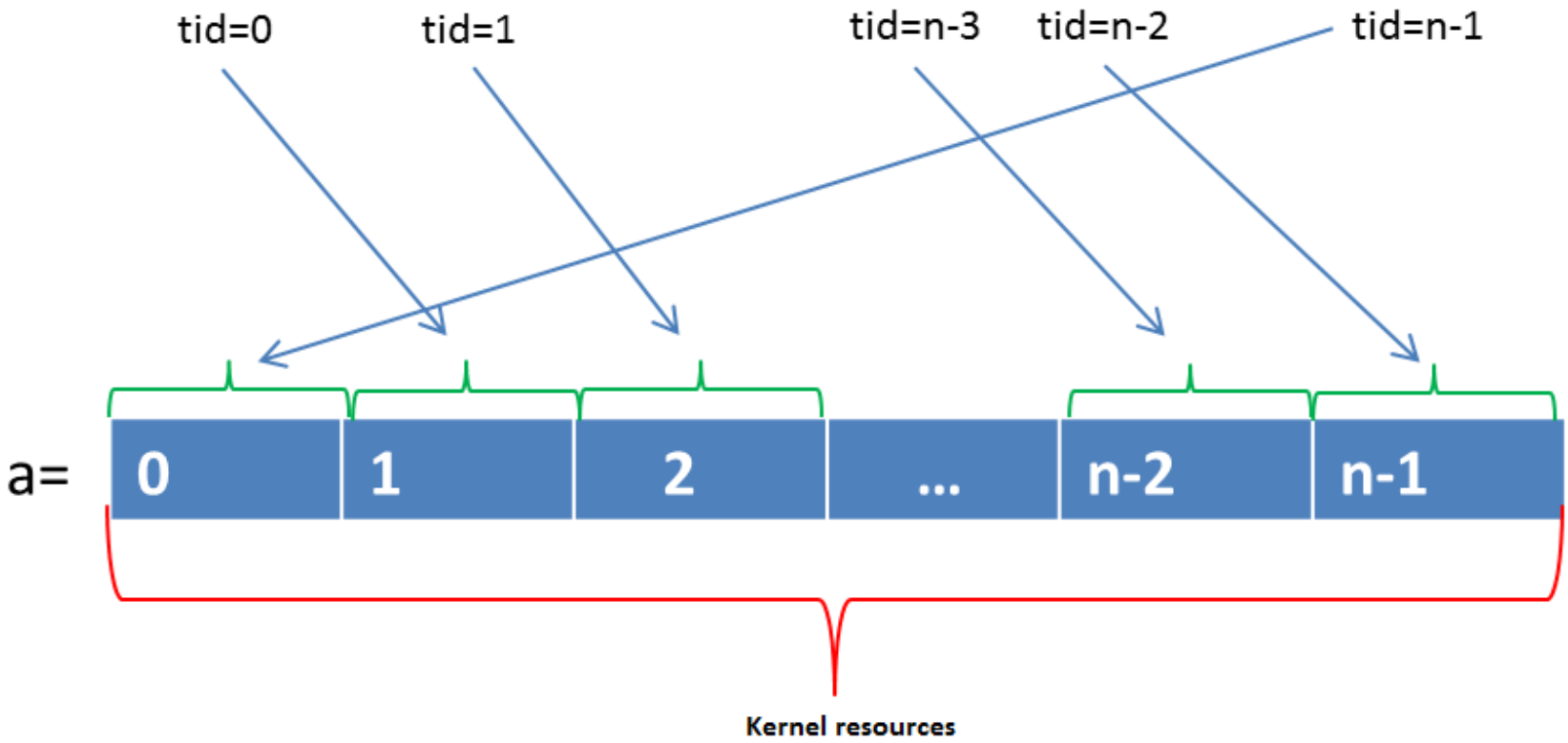


Figure : Permission redistribution at the barrier

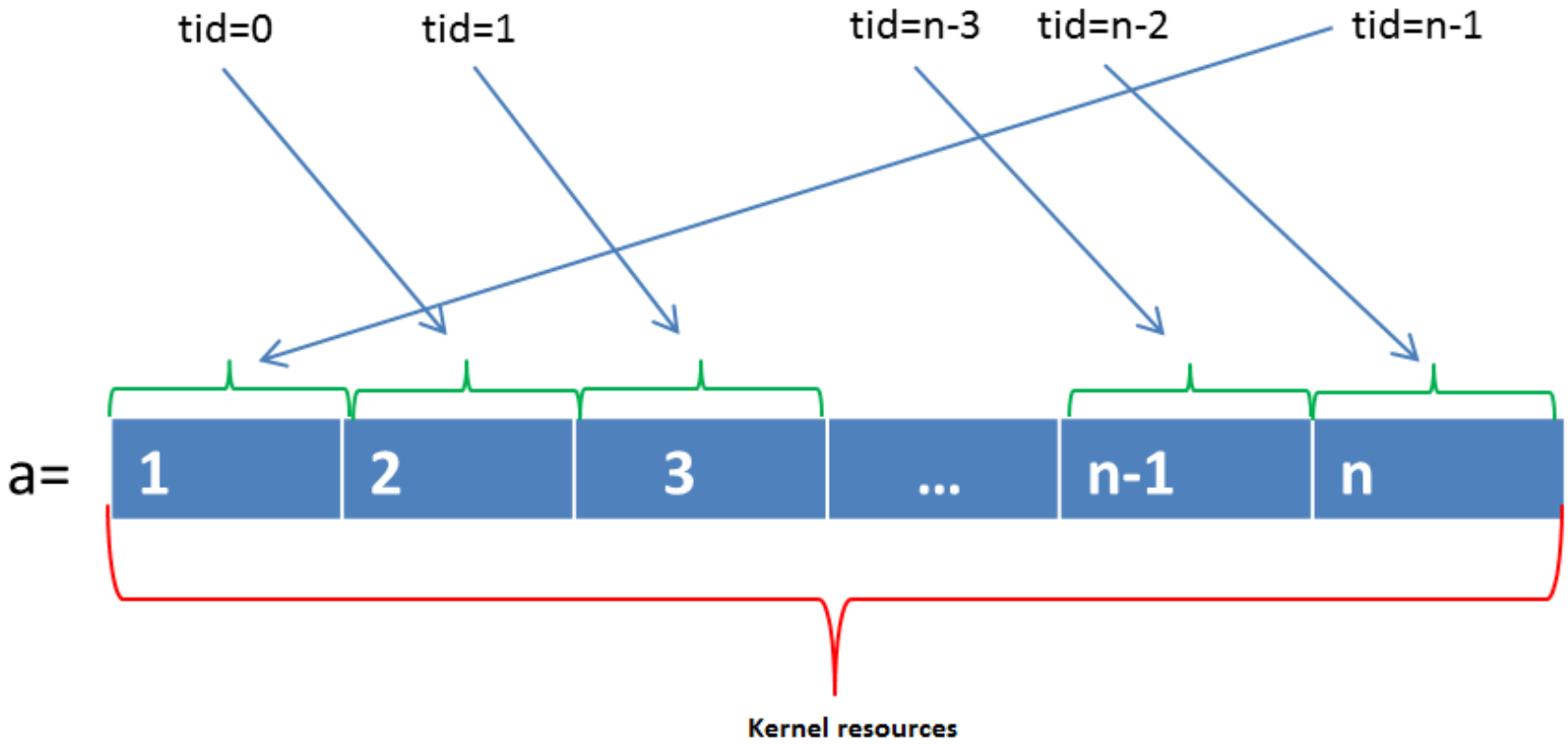


Figure : Array after the kernel execution

With the following barrier specification, verification of the example above would fail:

Barrier spec(B**)** : $(\texttt{Perm}(a[tid], 1) * \texttt{Perm}(a[(tid+1)\%size], \frac{1}{2}),$ $\text{true}, \text{true})$

We can show that the following properties are respected for our example kernel.

Kernel spec:

$(\underline{\texttt{resources:}} \ \ast_{i \in [0 \ldots size-1]} \texttt{Perm}(a[i], 1),$
$\underline{\texttt{precondition:}} \ size = n \wedge numthreads = n,$
$\underline{\texttt{postcondition:}} \ \forall_{i \in [0 \ldots size-1]} a[i] = (i + 1))$

Thread spec:
$(\underline{\texttt{resources:}} \ \texttt{Perm}(a[tid], 1),$
$\underline{\texttt{precondition:}} \ \text{true},$
$\underline{\texttt{postcondition:}} \ a[tid] = (tid + 1))$

Barrier spec(B) :
$(\texttt{Perm}(a[(tid + 1)\%size], 1), a[tid] = tid, \ \ true)$

The VerCors tool architecture:

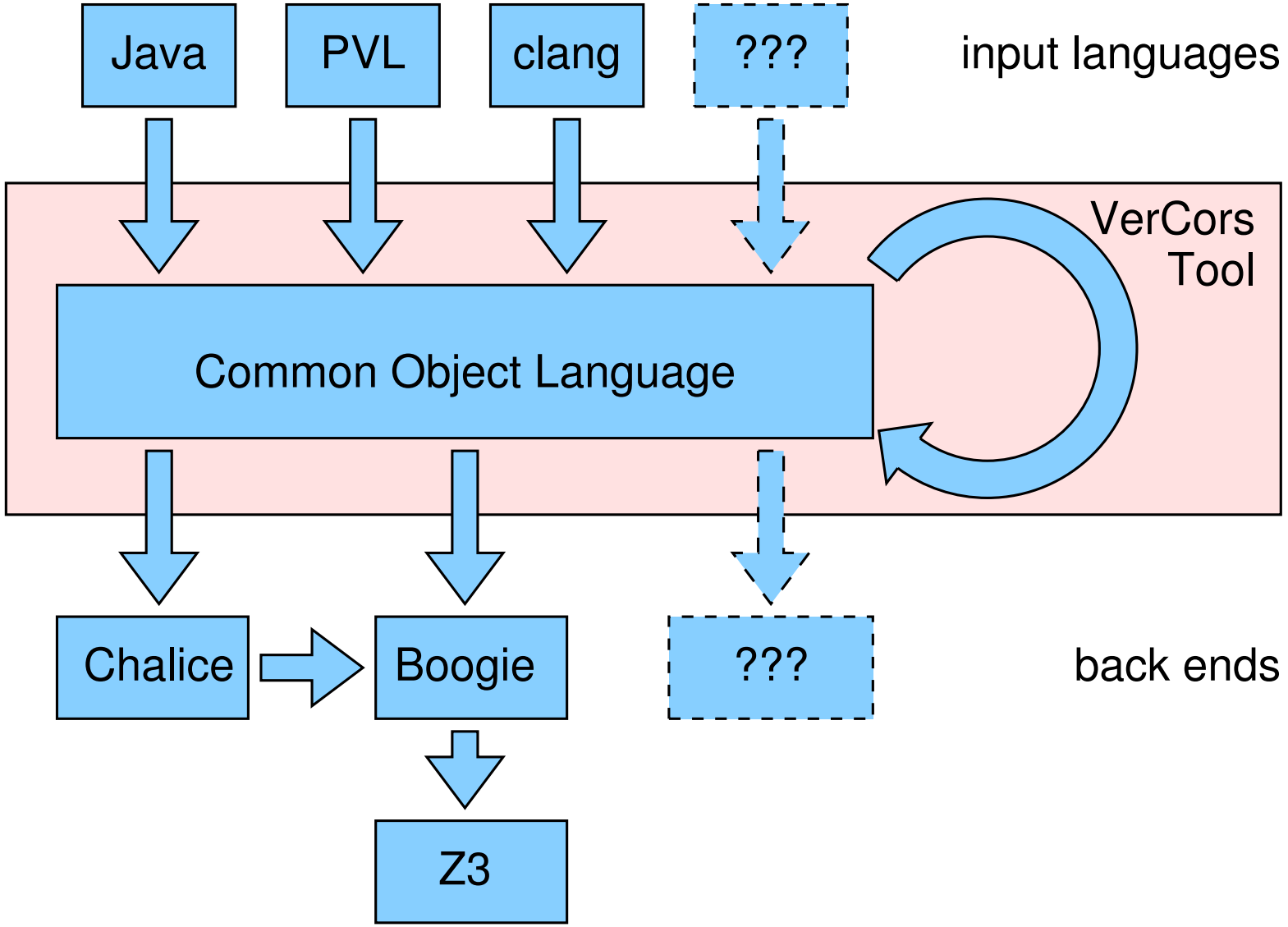


Figure : The VerCors tool architecture

Conclusion:

- We present a verification technique for GPGPU kernels, based on permission-based separation logic.
- For each kernel we specify all permissions that are necessary to execute the kernel
- The permissions in the kernel are distributed over the threads
- At each barrier the permissions are redistributed over the threads.
- Verification of individual threads uses standard program verification techniques
- Additional verification conditions check consistency of the specifications

Future work:

- Create a detailed formalisation of the logic and its soundness proof
- Develop the tool support as an extension of the VerCors tool
- Study automatic generation of permission specifications
- Study more kernel examples
- Explore the ways to verify absence of barrier divergence in our approach
- Reason about the host program to allow verification of multi-kernel applications running in a heterogeneous setting.

Questions?