UNIVERSITY OF TWENTE.

VERIFICATION OF GPU KERNELS WITH VERCORS

MARIEKE HUISMAN

UNIVERSITY OF TWENTE, FORMAL METHODS AND TOOLS GROUP

OVERVIEW

- What is VerCors?
- GPU programs: main characteristics
- A small example: leftRotation
- A logic for GPU kernels
- VerCors and Viper encoding
- Kernels with atomics
- Case studies
- Future work

VERCORS (VERIFICATION OF CONCURRENT PROGRAMS)

- Basis for reasoning: Permission-based Separation Logic
- Java-like programs (thread creation, thread joining, reentrant locks)
- GPU-like programs
- Permissions:
  - Write permission: exclusive access
  - Read permission: shared access
  - Permission specifications combined with functional properties

HTTP://UTWENTE.NL/VERCORS/

REASONING WITH PERMISSIONS

- Permissions: fractional value between 0 and 1
  - Write permission: exclusive access (encoded by 1)
  - Read permission: shared access (encoded by fractional value between 0 and 1)
- Global invariant: for each heap location, the sum of all the permissions in the system is never more than 1
- Read and write permissions can be exchanged whenever threads synchronise
- Permissions can be split and combined

  $\mathrm{Perm}(x, 1) \ast\!-\!\ast\ \mathrm{Perm}(x, ½) \ast \mathrm{Perm}(x, ½)$
- Permission specifications frame functional properties

PERMISSION-BASED SEPARATION LOGIC

Assertions: extension of predicate logic:

$\varphi ::= \mathrm{Perm}(x, \pi) \mid \varphi \ast \varphi \mid \ldots$

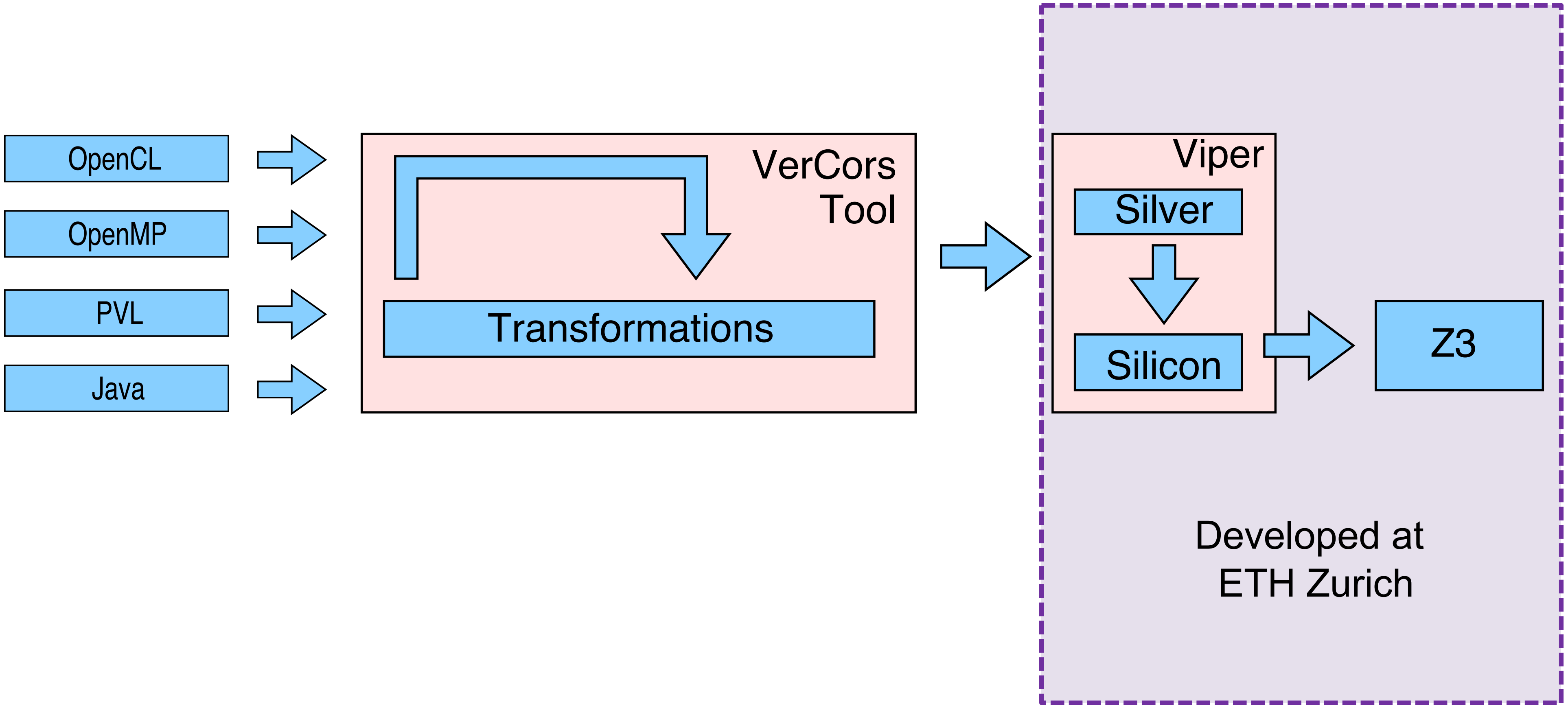
- $\mathrm{Perm}(x, \pi)$ – thread has permission π to access field x on heap

All formulas should be properly framed, i.e. you can only reason about heap locations that you have access to

- $\varphi 1 \ast \varphi 2$ – heap can be split in disjoint parts, satisfying $\varphi 1$ and $\varphi 2$

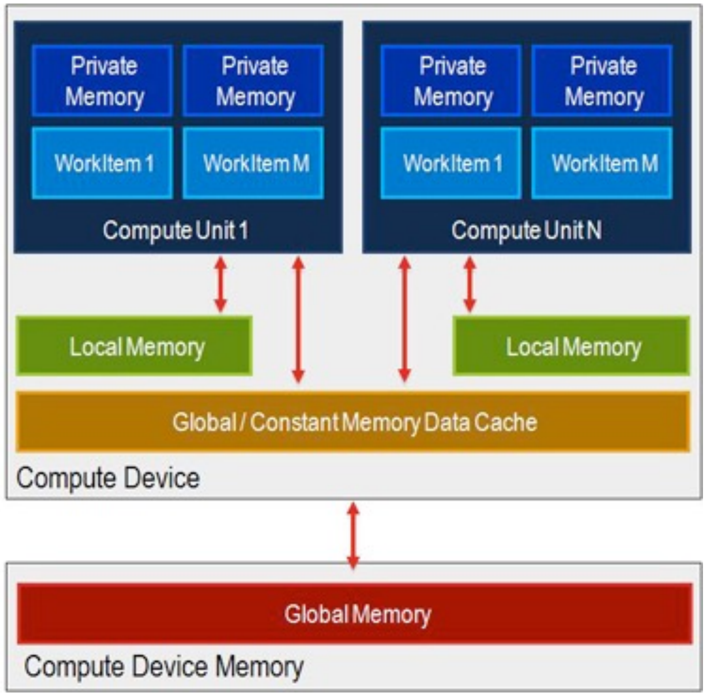
Supports local reasoning

VERCORS TOOL ARCHITECTURE



See iFM 2017

GENERAL-PURPOSE GPU PROGRAMMING



- Execution model: Single Instruction Multiple Data
  All threads in a kernel execute the same instruction, but work on their own share of memory
- 3 memory layers: global, local, private
- Threads organized within workgroups
  - Share local memory
  - Can synchronize via barrier

GPU EXECUTION MODEL

- Host program on CPU runs the main program
- Host program copies all relevant data to the GPU
- Host invokes kernel on a GPU
- All threads on the GPU execute kernel
- Once the kernel terminates, the host retrieves the result of the computation and continues the execution

EXAMPLE: LEFT ROTATION

```
__kernel void leftRotation(int array[], int size) {
    int temp;
    int tid = get_global_id(0);  // get the thread id
    if (tid != size – 1) {
        temp = array[tid + 1];
    } else {
        temp = array[0];
    }
    barrier(CLK_GLOBAL_MEM_FENCE);
    array[tid] = temp;
}
```

A LOGIC FOR GPU KERNELS

Plus: functional specifications (pre- and postconditions)

- Kernel specification
  - All permissions that a kernel needs for its execution
  - Separated in permissions for
    - Global Memory – given up by host code
    - Shared Memory – local to the GPU
- Group specification
  - All permissions that a group needs for its execution
  - Should be a subset of kernel permissions
- Thread specification
  - Permissions needed by single thread
  - Should be a subset of group permissions

GLOBAL PROOF OBLIGATIONS

- All workgroups together use no more resources than available in the kernel
- Kernel precondition implies universal quantification overall group identifiers of group preconditions
- Universal quantification over all group identifiers of group postconditions implies kernel postcondition

- All threads together use no more resources than available in the kernel
- Workgroup precondition implies universal quantification overall thread identifiers of thread preconditions
- Universal quantification over all thread identifiers of thread postconditions implies workgroup postconditions

KERNEL SPECIFICATION: LEFT ROTATION

```
/*@  context_everywhere array != NULL && array.length == size;
     requires tid != size – 1 ? Perm(array[tid + 1], write) : Perm(array[0], write);
     ensures Perm(array[tid], write);
     ensures tid != size – 1 ==> array[tid] == \old(array[tid + 1]);
     ensures tid == size – 1 ==> array[tid] == \old(array[0]); */
__kernel void leftRotation(int array[], int size) {
    int temp;
    int tid = get_global_id(0);   // get the thread id
    if (tid != size – 1) { temp = array[tid + 1];} else { temp = array[0];}
    barrier(CLK_GLOBAL_MEM_FENCE);
    array[tid] = temp;
}
```

BARRIER SYNCHRONIZATION

- Barrier specification
  - Each barrier allows redistribution of permissions
- Global proof obligations
  - All redistributed permissions available in workgroup
  - Universal quantification over all thread identifiers of barrier precondition implies universal quantification over all thread identifiers of barrier postcondition

    Transfer of knowledge about the state

BARRIER SPECIFICATION: LEFT ROTATION

```
/*@  context_everywhere array != NULL && array.length == size;
     requires tid != size – 1 ? Perm(array[tid + 1], write) : Perm(array[0], write);
     ensures Perm(array[tid], write);
     ensures tid != size – 1 ==> array[tid] == \old(array[tid + 1]);
     ensures tid == size – 1 ==> array[tid] == \old(array[0]); */
__kernel void leftRotation(int array[], int size) {
   int temp;
   int tid = get_global_id(0);   // get the thread id
   if (tid != size – 1) { temp = array[tid + 1];} else { temp = array[0];}
   /*@  requires tid != size – 1 ? Perm(array[tid + 1], write) : Perm(array[0], write);
        ensures Perm(array[tid], write); */
   barrier(CLK_GLOBAL_MEM_FENCE);
   array[tid] = temp;
}
```

VERCORS ENCODING

Host

```
context_everywhere array != null && array.length == size;
requires (\forall* int i; i >= 0 &&  i < size; Perm(array[i], 1));
ensures (\forall* int i; i >= 0 &&  i < size; Perm(array[i], 1));
ensures (\forall int i; i >= 0 &&  i < size; (i != size-1 ==> array[i] == \old(array[i+1])) && (i == size-1 ==> array[i] == \old(array[0])));
void leftRotation(int[] array, int size) {
    par thread (int tid = 0 .. size)
    requires tid != size-1 ==> Perm(array[tid+1], 1);
    requires tid == size-1 ==> Perm(array[0], 1);
    ensures Perm(array[tid], 1);
    ensures tid != size-1 ==> array[tid] == \old(array[tid+1]);
    ensures tid == size-1 ==> array[tid] == \old(array[0]);
    {
        int temp;
        if (tid != size-1) { temp = array[tid+1]; } else { temp = array[0]; }
        barrier(thread)
            requires tid != size-1 ==> Perm(array[tid+1], 1);
            requires tid == size-1 ==> Perm(array[0], 1);
            ensures Perm(array[tid], 1); {}
        array[tid] = temp;
    }
 }
```

Kernel

VIPER ENCODING

- Parallel Block
  - Abstract method with the kernel contract: called from the host code
  - Single thread method:
    - Thread contract
    - Thread body
- Barrier
  - Method with single thread specification, to verify thread body
  - Empty method to verify global proof obligations on barrier

VIPER: HOST CODE

```
method method_Example2_leftRotation__Option<Array<Cell<Integer>>>__Integer(diz: Ref, current_thread_id: Int,
globals: Ref, array: VCTOption[VCTArray[Ref]], size: Int)
  requires diz != null
  requires array != VCTNone() && alen(getVCTOption1(array)) == size
  requires 0 <= current_thread_id
  requires (forall i: Int :: 0 <= i && i < size ==> acc(loc(getVCTOption1(array), i).Integer__item, write))
  ensures array != VCTNone() && alen(getVCTOption1(array)) == size
  ensures (forall i: Int :: 0 <= i && i < size ==> acc(loc(getVCTOption1(array), i).Integer__item, write))
  ensures (forall i: Int :: 0 <= i && i < size ==> (i != size - 1 ==> loc(getVCTOption1(array), i).Integer__item ==
old(loc(getVCTOption1(array), i + 1).Integer__item)) && (i == size - 1 ==> loc(getVCTOption1(array),
i).Integer__item == old(loc(getVCTOption1(array), 0).Integer__item)))
{
  parallel_region_main_1(diz, current_thread_id, size, array)
}
```

VIPER: PARALLEL BLOCK SPEC

```
method parallel_region_main_1(diz: Ref, current_thread_id: Int, size: Int, array: VCTOption[VCTArray[Ref]])
  requires diz != null
  requires 0 <= current_thread_id
  requires array != VCTNone() && alen(getVCTOption1(array)) == size
  requires (forall k_fresh_rw_0: Int :: 0 <= k_fresh_rw_0 - 1 && k_fresh_rw_0 - 1 < size && k_fresh_rw_0 - 1 != size
- 1 ==> acc(loc(getVCTOption1(array), k_fresh_rw_0).Integer__item, write))
  requires 0 <= size - 1 && size - 1 < size ==> acc(loc(getVCTOption1(array), 0).Integer__item, write)
  ensures array != VCTNone() && alen(getVCTOption1(array)) == size
  ensures (forall tid: Int :: 0 <= tid && tid < size ==> acc(loc(getVCTOption1(array), tid).Integer__item, write))
  ensures (forall tid: Int :: 0 <= tid && tid < size && tid != size - 1 ==> loc(getVCTOption1(array), tid).Integer__item
== old(loc(getVCTOption1(array), tid + 1).Integer__item))
  ensures (forall tid: Int :: 0 <= tid && tid < size && tid == size - 1 ==> loc(getVCTOption1(array), tid).Integer__item
== old(loc(getVCTOption1(array), 0).Integer__item))
{
  inhale false
}
```

VIPER: PARALLEL BLOCK IMPLEMENTATION

```
method parallel_body_2(diz: Ref, current_thread_id: Int, size: Int, tid: Int, array: VCTOption[VCTArray[Ref]])
  requires diz != null
  requires 0 <= current_thread_id
  requires array != VCTNone() && alen(getVCTOption1(array)) == size
  requires 0 <= tid && tid < size
  requires tid != size - 1 ==> acc(loc(getVCTOption1(array), tid + 1).Integer__item, write)
  requires tid == size - 1 ==> acc(loc(getVCTOption1(array), 0).Integer__item, write)
  ensures array != VCTNone() && alen(getVCTOption1(array)) == size
  ensures 0 <= tid && tid < size
  ensures acc(loc(getVCTOption1(array), tid).Integer__item, write)
  ensures tid != size - 1 ==> loc(getVCTOption1(array), tid).Integer__item == old(loc(getVCTOption1(array), tid +
1).Integer__item)
  ensures tid == size - 1 ==> loc(getVCTOption1(array), tid).Integer__item == old(loc(getVCTOption1(array),
0).Integer__item)
{
  // body (next slide)
}
```

VIPER: PARALLEL BLOCK BODY

```
var temp__1: Int
  var __flatten_1__2: Ref
  var __flatten_2__3: Ref
  var __flatten_4__4: Ref
  var __flatten_6__5: VCTArray[Ref]
  var __flatten_7__6: Ref
  var __flatten_8__7: VCTArray[Ref]
  var __flatten_9__8: Ref
  var __flatten_11__9: VCTArray[Ref]
  var __flatten_12__10: Ref
  if (tid != size - 1) {
    __flatten_6__5 := getVCTOption1(array)
    __flatten_7__6 := loc(__flatten_6__5, tid + 1)
    __flatten_1__2 := __flatten_7__6
    temp__1 := __flatten_1__2.Integer__item
  } else {
    __flatten_8__7 := getVCTOption1(array)
    __flatten_9__8 := loc(__flatten_8__7, 0)
    __flatten_2__3 := __flatten_9__8
    temp__1 := __flatten_2__3.Integer__item
  }
  barrier_main_2(diz, current_thread_id, size, tid, array)
  __flatten_11__9 := getVCTOption1(array)
  __flatten_12__10 := loc(__flatten_11__9, tid)
  __flatten_4__4 := __flatten_12__10
  __flatten_4__4.Integer__item := temp__1
}
```

VIPER: BARRIER CALL

```
method barrier_main_2(diz: Ref, current_thread_id: Int, size: Int, tid: Int, array: VCTOption[VCTArray[Ref]])
  requires diz != null
  requires array != VCTNone() && alen(getVCTOption1(array)) == size
  requires 0 <= current_thread_id
  requires tid != size - 1 ==> acc(loc(getVCTOption1(array), tid + 1).Integer__item, write)
  requires tid == size - 1 ==> acc(loc(getVCTOption1(array), 0).Integer__item, write)
  ensures array != VCTNone() && alen(getVCTOption1(array)) == size
  ensures acc(loc(getVCTOption1(array), tid).Integer__item, write){
  inhale false
}
```

VIPER: BARRIER PROOF OBLIGATION

```
method barrier_check_2(diz: Ref, current_thread_id: Int, size: Int, array: VCTOption[VCTArray[Ref]])
  requires diz != null
  requires array != VCTNone() && alen(getVCTOption1(array)) == size
  requires 0 <= current_thread_id
  requires 0 < |[0..size)|
  requires (forall k_fresh_rw_0: Int :: 0 <= k_fresh_rw_0 - 1 && k_fresh_rw_0 - 1 < size && k_fresh_rw_0 - 1 != size
- 1 ==> acc(loc(getVCTOption1(array), k_fresh_rw_0).Integer__item, write))
  requires 0 <= size - 1 && size - 1 < size ==> acc(loc(getVCTOption1(array), 0).Integer__item, write)
  ensures array != VCTNone() && alen(getVCTOption1(array)) == size
  ensures 0 < |[0..size)|
  ensures (forall tid: Int :: 0 <= tid && tid < size ==> acc(loc(getVCTOption1(array), tid).Integer__item, write))
{
}
```

REASONING ABOUT KERNELS WITH ATOMICS

```
requires Perm(values[ltid],1/2);
ensures Perm(values[ltid],1/2);
kernel void gpadd(int x, int values){
   atomic_add(x,values[ltid]);
}
```

How to verify that x is the sum of all elements in value?

$$\frac{\{o.invariant() * P\}\ S\ \{o.invariant() * Q\}}{\{P\}\ \textbf{atomic}(o)\{S\}\ \{Q\}}$$

Group resource invariant: Permission obtained by the thread when executing the atomic operation

ATOMICS AND FUNCTIONAL PROPERTIES

```
given int cont[gsize];
group invariant Perm(x,1) ✳ Perm(cont[*],1/2)} ✳ x== \sum(cont[*]);
requires Perm(values[ltid],1/2) ✳ Perm(cont[ltid],1/2) ✳ cont[ltid]==0;
ensures Perm(values[ltid],1/2) ✳ Perm(cont[ltid],1/2) ✳ cont[ltid]==values[ltid];
kernel void gpadd(int x, int values){
   atomic_add(x,values[ltid]) /*@ then { cont[ltid]=values[ltid]; } @*/;
}
```

cont: ghost variable array to keep track of which values have been summed up

Similar techniques when atomic operation used by multiple workgroups

CASE STUDIES OF VARIOUS PARALLEL ALGORITHMS

- In-place prefix sum algorithms (exclusive and inclusive prefix sum)
- Algorithms that use prefix sum
  - Stream compaction
  - Summed-area Table
- Single-Source Shortest Path algorithm

FUTURE PLANS

- Improve support for CUDA/OpenCL
- More case studies
- Annotation generation
- Correctness preserving optimisations