A Sound and Complete Abstraction for Reasoning About Parallel Prefix Sums

Nathan Chong  Alastair F. Donaldson  Jeroen Ketema

Imperial College London

York Concurrency Workshop, 2014

Supported by the EU FP7 STREP project CARP: Correct and Efficient Accelerator Programming
Based on our paper:

- Nathan Chong, Alastair F. Donaldson, Jeroen Ketema: A sound and complete abstraction for reasoning about parallel prefix sums. POPL 2014: 397-410
Prefix sums

Prefix sum of

\([x_1, x_2, \ldots, x_n]\)

is

\([x_1, x_1 + x_2, \ldots, x_1 + x_2 + \ldots + x_n]\)
Prefix sums

E.g., prefix sum of:

\[0, 0, 0, 1, 1, 1, 2, 2, 2, 2\]

is

\[0, 0, 0, 1, 2, 3, 5, 7, 9\]

Let us implement a sequential prefix sum
void sequential_prefix_sum(int * in, int * out) {
    out[0] = in[0];
    for(unsigned i = 1; i < NELEMENTS; ++i) {
        out[i] = out[i-1] + in[i];
    }
}
More generally

Let $A$ be a set and $\bullet$ an associative binary operator on $A$:

$$a \bullet (b \bullet c) = (a \bullet b) \bullet c$$

for any $a, b, c$ in $A$.

The prefix sum of:

$$[x_1, x_2, \ldots, x_n]$$

is

$$[x_1, x_1 \bullet x_2, \ldots, x_1 \bullet x_2 \bullet \ldots \bullet x_n]$$
Prefix sums are widely used

<table>
<thead>
<tr>
<th>Application</th>
<th>Datatype</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream compaction</td>
<td>Int</td>
<td>+</td>
</tr>
<tr>
<td>Sorting algorithms</td>
<td>Int</td>
<td>+</td>
</tr>
<tr>
<td>Polynomial interpolation</td>
<td>Float</td>
<td>*</td>
</tr>
<tr>
<td>Line-of-sight calculation</td>
<td>Int</td>
<td>max</td>
</tr>
<tr>
<td>Binary addition</td>
<td>Pairs of bits</td>
<td>carry operator</td>
</tr>
<tr>
<td>Finite state machine simulation</td>
<td>Transition functions</td>
<td>Function composition</td>
</tr>
</tbody>
</table>
GPU implementations of prefix sums

Key building blocks in parallel applications

We would like to prove correctness of GPU implementations of generic prefix sums

Our method:

1. Prove that implementation is race-free
2. Run single test case

Knowing (1), (2) precisely determines correctness for all data types

Restriction: only works for fixed-length arrays
GPU implementations of prefix sums

Let’s implement a prefix sum as a GPU kernel
__kernel
void prefixsum(__global TYPE * __restrict input, __global TYPE * __restrict output) {

    unsigned tid = get_local_id(0);
    unsigned gid = get_global_id(0);

    __local TYPE result[NELEMENTS];

    result[tid] = input[gid];

    barrier(CLK_LOCAL_MEM_FENCE);
    for(unsigned i = 1; i < NELEMENTS; i *= 2) {
        TYPE temp;
        if(tid >= i) {
            temp = result[tid - i];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
        if(tid >= i) {
            result[tid] = OPERATOR(temp, result[tid]);
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    output[gid] = result[tid];
}
Idea: Exploit the fact that a generic prefix sum can depend on nothing more than associativity

Imagine a generic prefix sum from in to out

out[k] should be

\[\text{in}[0] \bullet \text{in}[1] \bullet \ldots \bullet \text{in}[k]\]

How could \(\text{in}[0] \bullet \text{in}[1] \bullet \ldots \bullet \text{in}[k]\) have been computed?
How could \( \text{in}[0] \bullet \text{in}[1] \bullet \ldots \bullet \text{in}[k] \) have been computed?

Only one way: by plugging together contiguous summations that “kiss”

\[
( \text{in}[0] \bullet \text{in}[1] \bullet \ldots \bullet \text{in}[d] )
\]

\[
( \text{in}[d+1] \bullet \text{in}[d+2] \bullet \ldots \bullet \text{in}[k] )
\]

Anything else would depend on more than associativity
How could \((\text{in}[d+1] \bullet \text{in}[d+2] \bullet \ldots \bullet \text{in}[k])\) have been computed?

Again, just one way:

\[(\text{in}[d+1] \bullet \text{in}[d+2] \bullet \ldots \bullet \text{in}[e])\]
Interval of summations abstraction

For $i \leq j$, use:

$$(i, j)$$

to abstractly represent the summation interval:

$$\text{in}[i] \bullet \text{in}[i+1] \bullet \ldots \bullet \text{in}[j]$$

Use a special “value of death”:

$\pmb{\text{∞}}$

to abstractly represent something that might not be a summation interval
Abstract addition: kiss or die

Define binary operator $\oplus$ on the interval of summations domain:

$$(i, j) \oplus (k, l) = \begin{cases} (i, l) & \text{if } j+1 = k \\ \text{otherwise} & \end{cases}$$

$x \oplus \text{skull} = \text{skull} \oplus x = \text{skull}$ is an absorbing element
⊕ is an associative operator on the interval of summations domain

We can do a prefix sum w.r.t. ⊕

Main result:
a sequential generic prefix sum of length $n$
is correct for all data types

$\iff$
it produces the correct result for the interval domain on the input:

$$[(0,0),(1,1),\ldots,(n-1,n-1)]$$
Some intuition for why this holds
(formal proof: POPL’14)

=> is trivial: a correct generic prefix sum should work just fine on the interval domain

<= (waffle waffle waffle)
Extension to data-parallel programs

Observation
If barriers provide the only means of synchronization, then:

race-freedom $\Rightarrow$ determinism

Consequence:
Our main result also holds for race-free GPU kernel implementations of generic prefix sums
Let's do it

```
// 2D vector used to represent an interval
#define TYPE uint2

// (1, 0) chosen arbitrarily to represent the value of death
#define DEATH ((uint2) { 1, 0 })

#define OPERATOR(A, B) (  
    A.x <= A.y && /* A is well-formed */  
    A.y + 1 == B.x && /* A and B kiss */  
    B.x <= B.y /* B is well-formed */  
)?  
    ((uint2) { A.x, B.y }) 
:  
    DEATH )

// Some printing code
#define PRINT(A) 
    if(!(A.s[0] <= A.s[1])) { 
        std::cout << "DEATH";
    } else { 
        std::cout << "(" << A.s[0] << ", " << A.s[1] << ");";
    }
```
Let’s do it

Running the test case

```
C:\Users\afd\Documents\Visual Studio 2010\Projects\POPL14\x64\Debug\YorkConcurrenc
ncy.exe
input:
(0, 0) (1, 1) (2, 2) (3, 3) (4, 4) (5, 5) (6, 6) (7, 7) (8, 8) (9, 9) (10, 10) (11, 11)
(12, 12) (13, 13) (14, 14) (15, 15) (16, 16) (17, 17) (18, 18) (19, 19)
(29, 29) (30, 30) (31, 31)
output:
(0, 0) (0, 1) (0, 2) (0, 3) (0, 4) (0, 5) (0, 6) (0, 7) (0, 8) (0, 9) (0, 10)
(0, 11) (0, 12) (0, 13) (0, 14) (0, 15) (0, 16) (0, 17) (0, 18) (0, 19) (0, 20)
(0, 21) (0, 22) (0, 23) (0, 24) (0, 25) (0, 26) (0, 27) (0, 28) (0, 29) (0, 30)
(0, 31)
Abstract prefix sum computed the correct result
```
Let’s do it

Verifying race-freedom

This generic prefix sum is thus correct for length 32, for all datatypes and operators.
Experimental summary

Four prefix sum algorithms: Kogge-Stone, Blelloch, Sklansky, Brent-Kung

Race-freedom proven efficiently for all power-of-two array lengths up to $2^{32}$

Interval abstraction test cases run efficiently on two NVIDIA GPUs and two Intel CPUs

We’ve verified all the prefix sums people care about, for all interesting array sizes
Related work

Janis Voigtländer: Much ado about two (pearl): a pearl on parallel prefix computation. POPL 2008: 29-35


These works show that functional correctness of sequential prefix sums can be determined by running a single test case. Our work brings: a much more efficient representation (with lower space complexity) and an extension to data-parallel programs. See our POPL paper for a detailed discussion.
Summary

A highly automatic method for proving functional correctness of GPU implementations of generic prefix sums

Basis: an abstraction which is just right for this problem

Questions:
- Can this abstraction be applied elsewhere? (I don’t think so)
- Can the idea of finding a “just right” abstraction for a problem be applied elsewhere? (I hope so!)