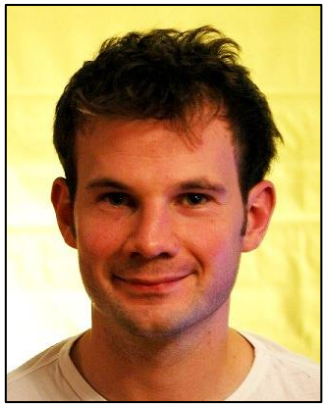
A Sound and Complete Abstraction for Reasoning About Parallel Prefix Sums


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*

SEVENTH FRAMEWORK PROGRAMME

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

A list of integers

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

is

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

Integer addition

Prefix sums

E.g., prefix sum of:

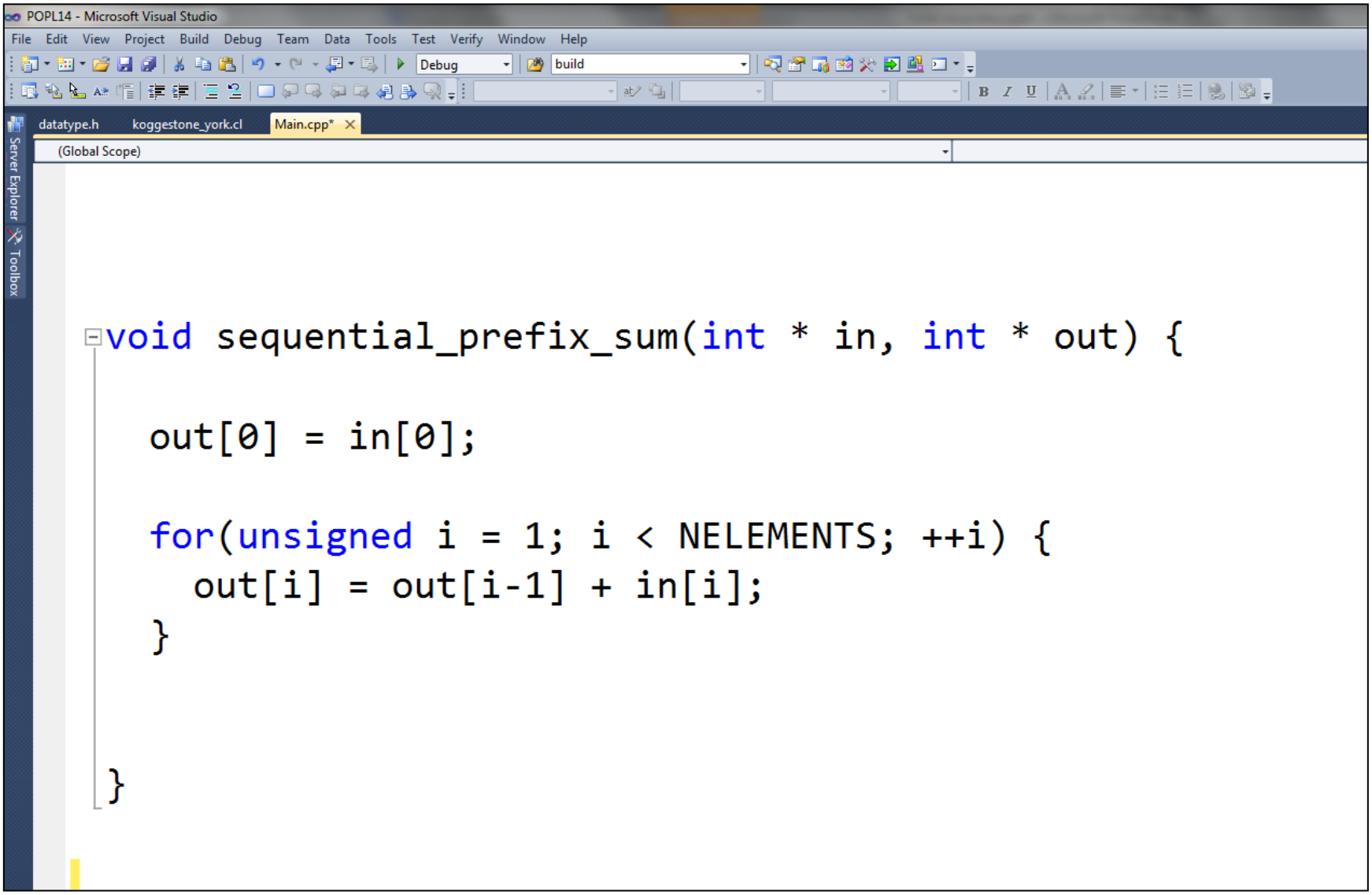
[0, 0, 0, 1, 1, 1, 2, 2, 2]

is

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

Let us implement a sequential prefix sum

Live coding


POPL14 - Microsoft Visual Studio
File Edit View Project Build Debug Team Data Tools Test Verify Window Help
Debug
build
datatype.h
koggestone_york.cl
Main.cpp*
(Global Scope)
Server Explorer
Toolbox
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 ● 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:

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

List of As

is

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

Prefix sums are widely used

Application	Datatype	Operator
Stream compaction	Int	+
Sorting algorithms	Int	+
Polynomial interpolation	Float	*
Line-of-sight calculation	Int	max
Binary addition	Pairs of bits	carry operator
Finite state machine simulation	Transition functions	Function composition

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:

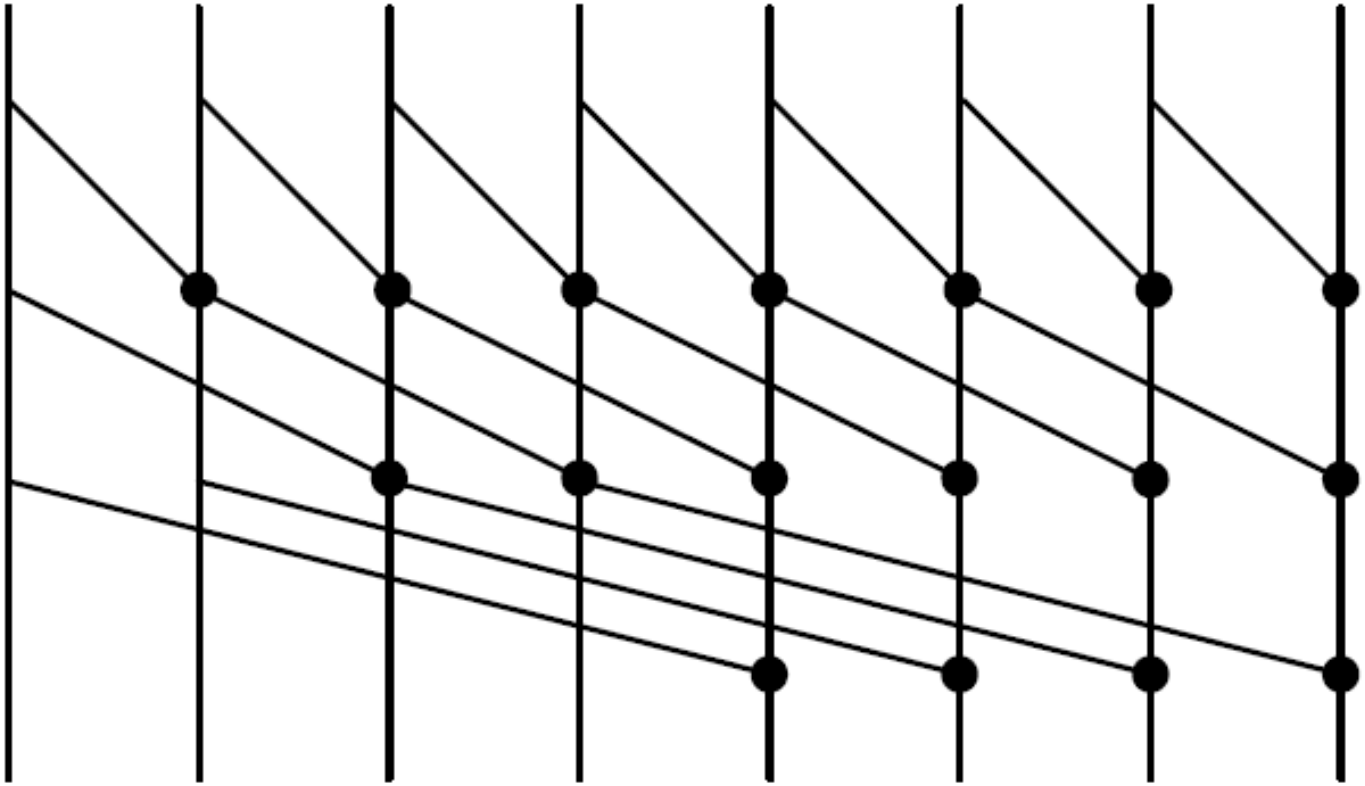
(1) Prove that implementation is race-free
(2) Run **single** test case

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

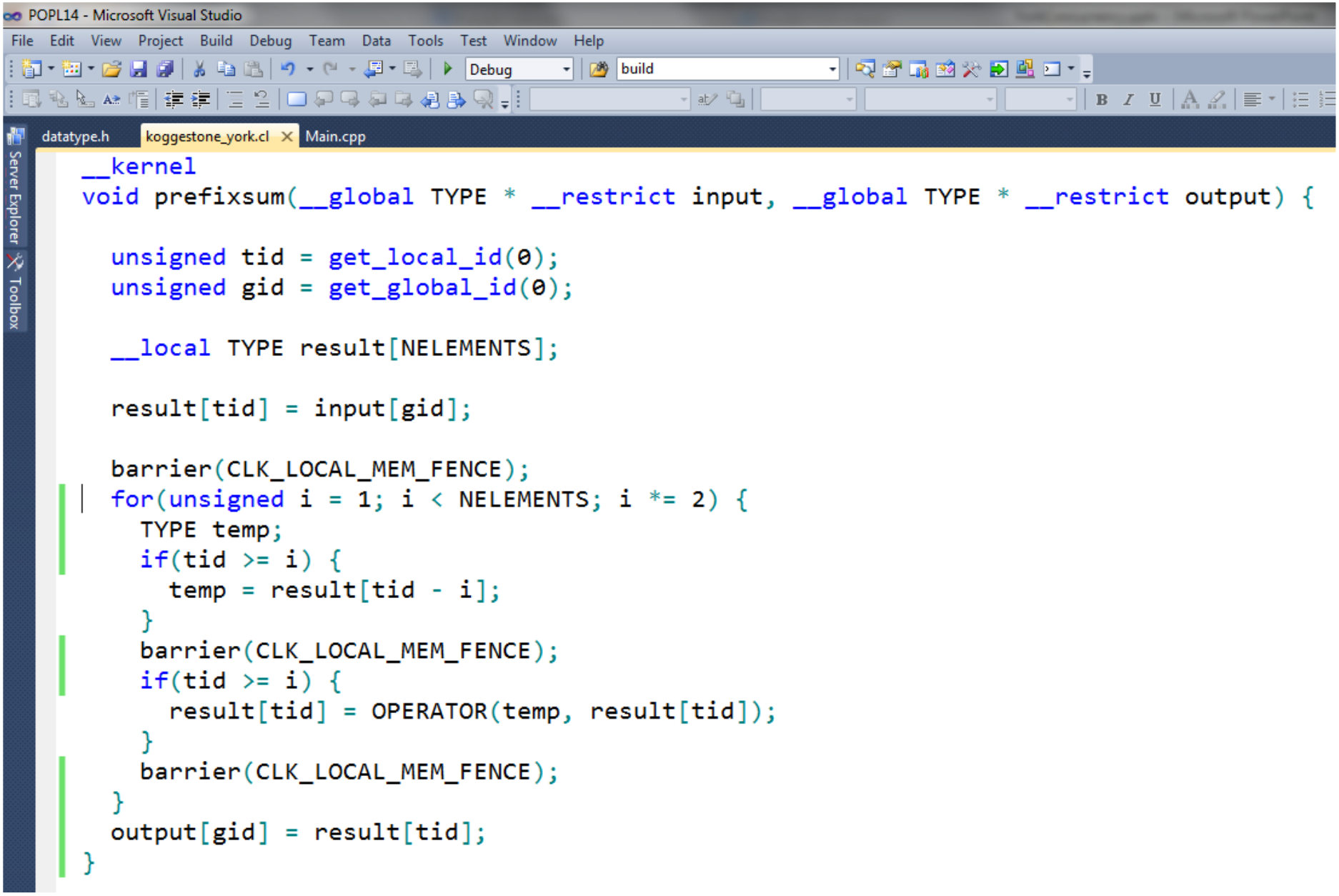
Restriction: only works for fixed-length arrays

GPU implementations of prefix sums

Let's implement a prefix sum as a GPU kernel



Live coding


POPL14 - Microsoft Visual Studio
File Edit View Project Build Debug Team Data Tools Test Window Help
Debug
build
datatype.h
koggestone_york.cl
Main.cpp
Server Explorer
Toolbox
__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

in[0] ● **in**[1] ● … ● **in**[k]

How could **in**[0] ● **in**[1] ● … ● **in**[k]

have been computed?

How could **in**[0] ● **in**[1] ● … ● **in**[k]
have been computed?

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

(**in**[0] ● **in**[1] ● … ● **in**[d])

●

(**in**[d+1] ● **in**[d+2] ● … ● **in**[k])

Anything else would depend on more than
associativity

(**in**[0] ● **in**[1] ● … ● **in**[d])

●

(**in**[d+1] ● **in**[d+2] ● … ● **in**[k])

How could (**in**[d+1] ● **in**[d+2] ● … ● **in**[k]) have been computed?

Again, just one way:

(**in**[d+1] ● **in**[d+2] ● … ● **in**[e])

●

(**in**[e+1] ● **in**[d+2] ● … ● **in**[e])

et cetera

Interval of summations abstraction

For $i \leq j$, use:

$$(i,\ j)$$

to abstractly represent the summation interval:

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

Use a special "value of death": 

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:

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

$$x \oplus \text{☠} = \text{☠} \oplus x = \text{☠}$$

☠ is an **absorbing** element

$\oplus$ is an associative operator on the interval of summations domain

We can do a prefix sum w.r.t. $\oplus$

Main result:

a sequential generic prefix sum of length *n*
is correct for **all data types**

<=>

it produces the correct result for the
interval domain on the input:

$$[(0,0),(1,1),\ldots,(n\text{-}1,n\text{-}1)]$$

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

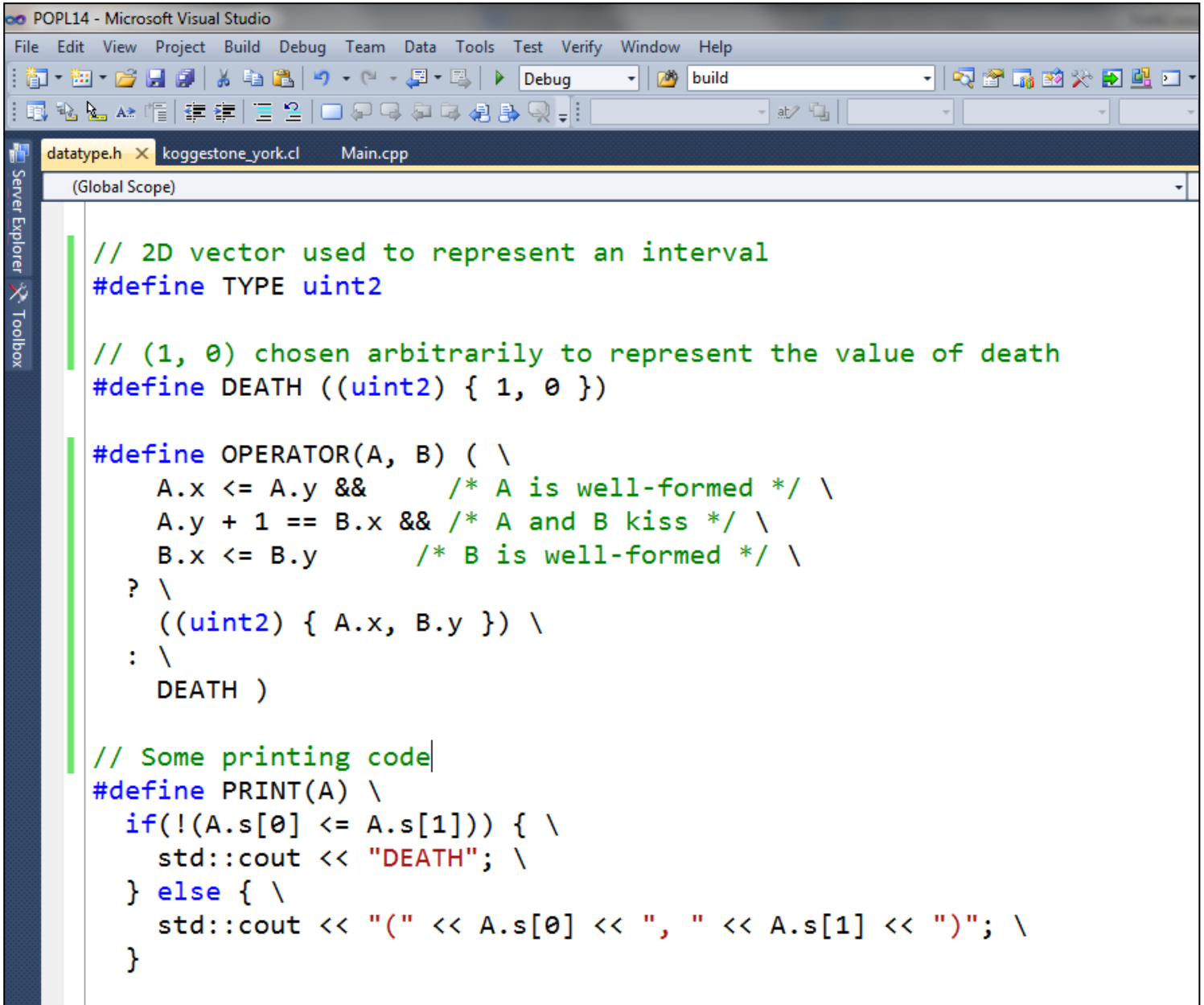
If barriers provide the only means of synchronization, then:

race-freedom => determinism

Consequence:

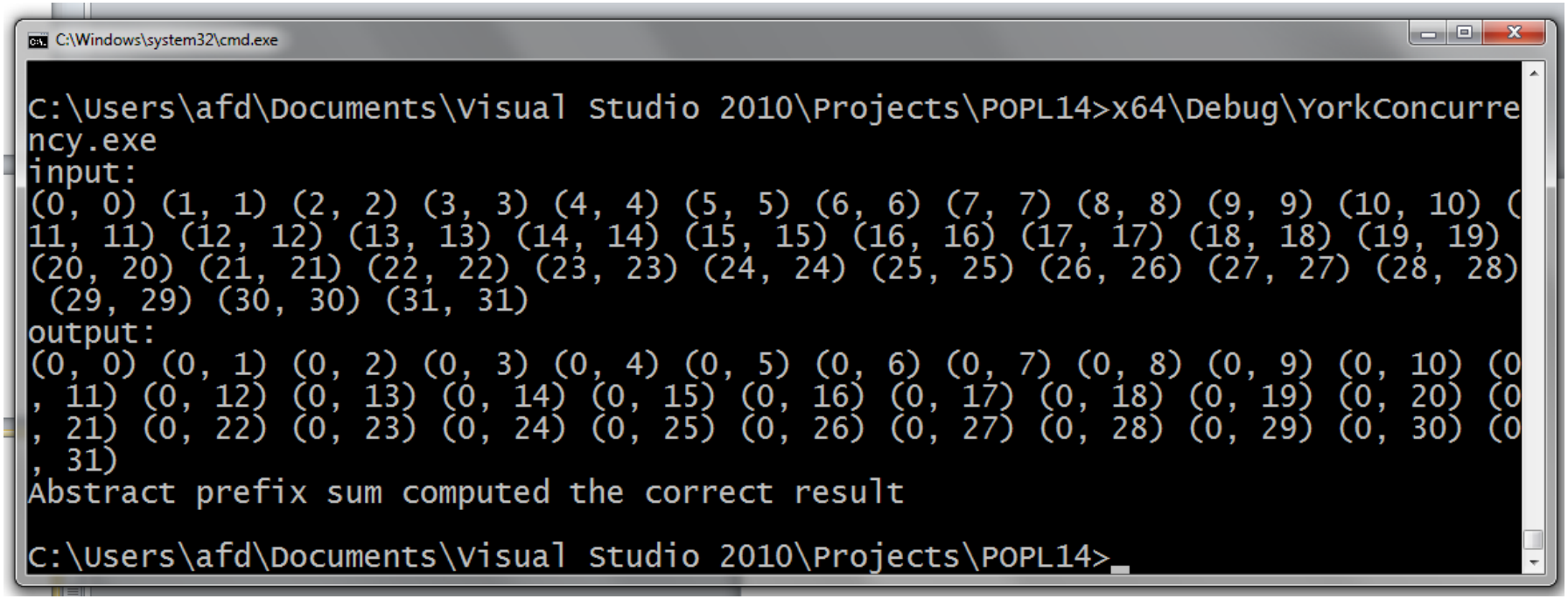
Our main result also holds for race-free GPU kernel implementations of generic prefix sums

Let's do it


POPL14 - Microsoft Visual Studio
File Edit View Project Build Debug Team Data Tools Test Verify Window Help
Debug
build
datatype.h
koggestone_york.cl
Main.cpp
(Global Scope)
Server Explorer
Toolbox
// 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] << ")"; \
}

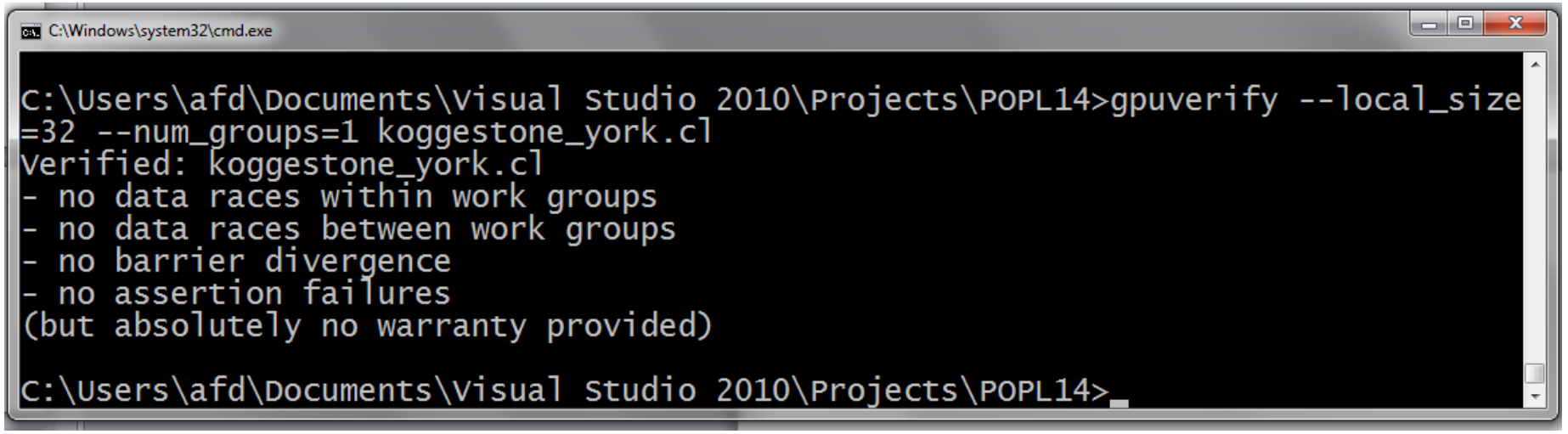
Let’s do it

Running the test case



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

Mary Sheeran: Functional and dynamic programming in the design of parallel prefix networks. J. Funct. Program. 21(1): 59-114 (2011)

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!)