THE FOUNDRY

@SIGGRAPH 2012

Accelerating high-end compositing with CUDA in NUKE

Jon Wadelton

NUKE Product Manager

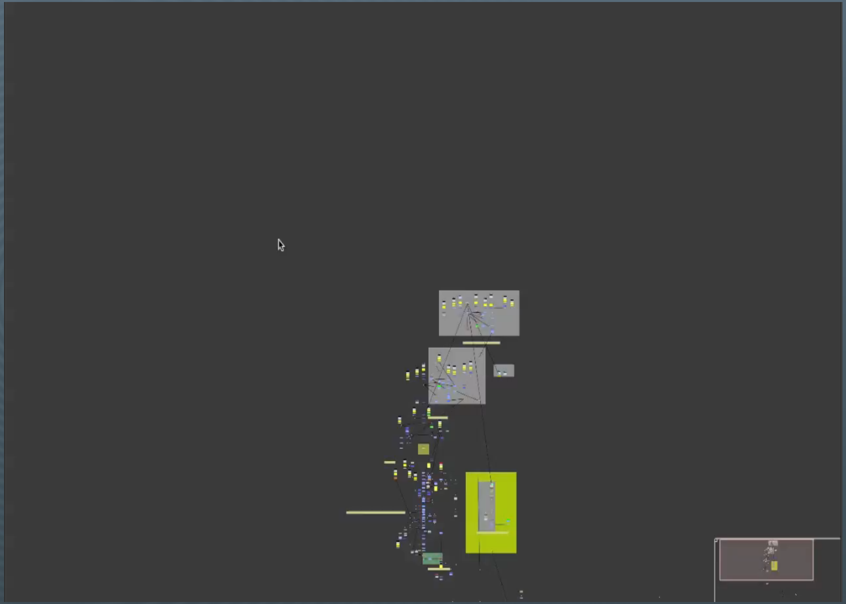
Overview

– What is NUKE?
– Image processing - exploiting the GPU
– The Foundry Approach
– Simple examples in NUKEX
– Real world examples in NUKEX
– Future research
– What we learnt

THE FOUNDRY @SIGGRAPH

What is NUKE?

- NUKE is a node based compositing system.
- Designed to work at high quality, large image sizes, large processing nodes
- All image processing traditionally done on the CPU
- Users extensively use 'render farms' to parallelise processing of frames



Xmen First Class © 2011 Fox / Marvel. All rights reserved. Images courtesy of Digital Domain

Modern Compute Devices

- CPUs
  - 'easy' to program not highly parallel
  - very flexible, can program anything and get OK performance
- GPUs
  - harder to program highly parallel
  - only really suitable for highly parallel workloads
- Both are parallel, huge difference is in degree
- Image processing likes parallelism. GPU beats the CPU.

GPUs Come of Age For Complex Processing

- CUDA/OpenCL > OpenGL
  - Allows for complex image processing! Eg. Motion estimation.

- New opportunity to improve our software
  - reduce latency for the artist
  - increase throughput on renders
  - use existing algorithms in new situations

Challenges

- We still need a CPU compute path
  - for CPU based render farms
  - CPUs are resources, we should use everything we can improve FLOPs.
  - for machines old or slow GPUs

- CPU and GPU results <u>must</u> agree
  - not truly possible due to nature of the hardware
  - visually indistinguishable is the metric we need

More Challenges

- ‘Compute Landscape’ is changing rapidly
  - New hardware continually appearing, SSE, AVX, CUDA, OpenCL…
  - CPUs and GPUs are car crashing
  - Programming APIs are evolving and new ones appearing

- Which winner do we back?
- Will there even be a single ‘winner’?

Even More Challenges

- Getting peak performance is a specialist task
  - You need to do it differently per device
  - Hand optimisation gets in the way of writing algorithms

How do we solve this?

- Write multiple separate implementations for each device?
    - Costly
    - Buggy
    - Needs redoing for each new hardware innovation

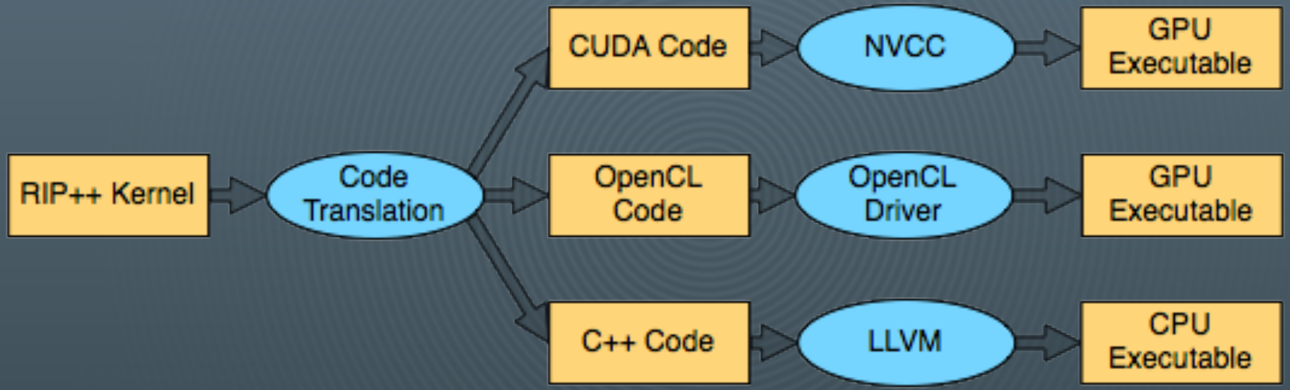
Introducing `Blink'

- Or AKA "Righteous Image Processing". **RIP.**
- A multi-device image processing framework
- Based on research done with Imperial College London

RIP Overview

- RIP is a domain specific C++ like languages for image processing
- We express operations as ‘kernels’
- These are device independent and clear expressions of an algorithm

RIP Workflow



Doesn’t OpenCL Do That?

- No
- RIP is image processing domain specific
- A device independent way to describe the algorithm

Data Dependence Is Key To Parallelism

- Parallelism is where is where FLOPs are
- Algorithm's data dependence is what constrains its parallelism
- Explicit dependence = analysis free knowledge of parallelism

RIP Basic Design

- Purely for image processing
- Access to all data is abstracted and made explicit
  - Access patterns
  - Kernel types:
    - Regular
    - Reductions
    - Carry dependence

Access Pattern Specifications

- Pattern of access at each point in iteration space is main abstraction
  - ‘tap’ i.e. the current point
  - 1D or 2D range around the current iteration position
  - random access
- Read or Write
- Integer transforms
  - scale, rotate, translate, transpose
- Edge conditions

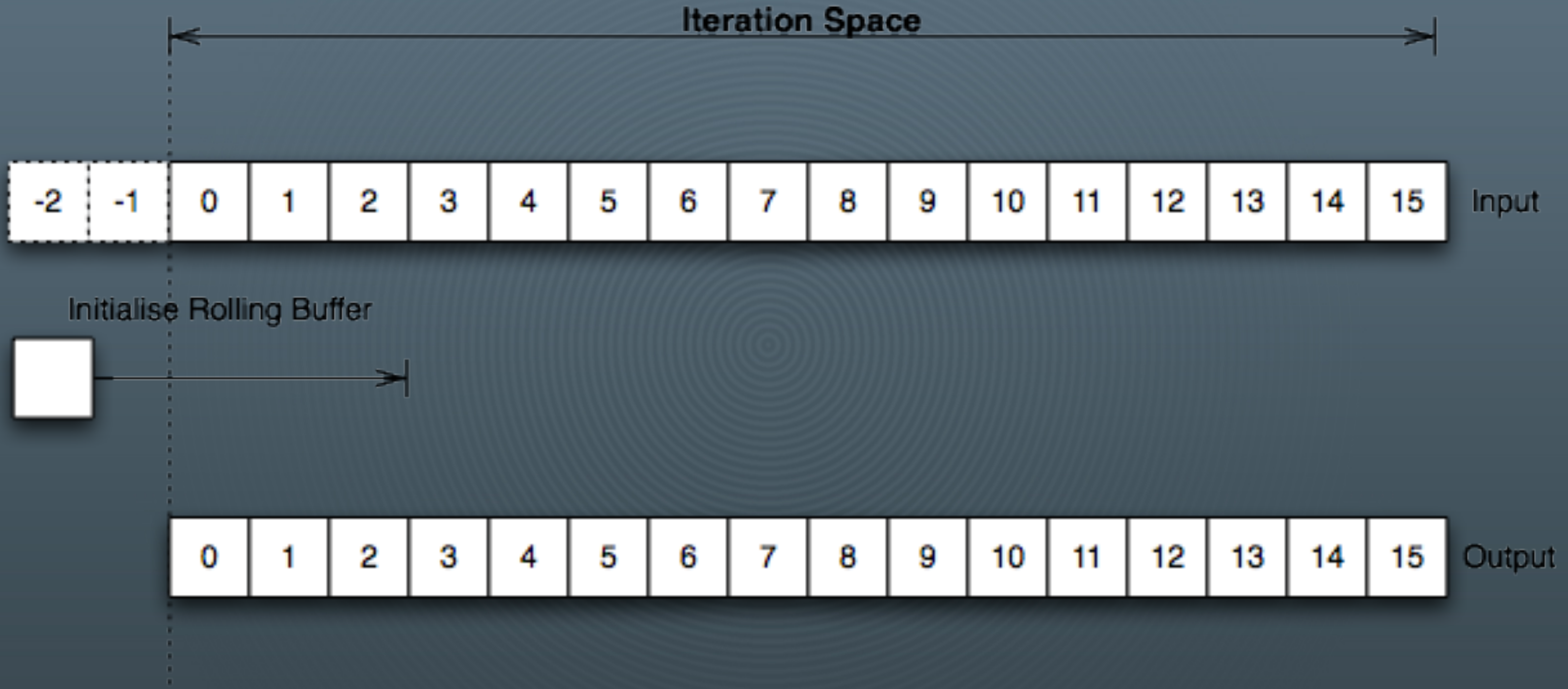
Regular Kernels

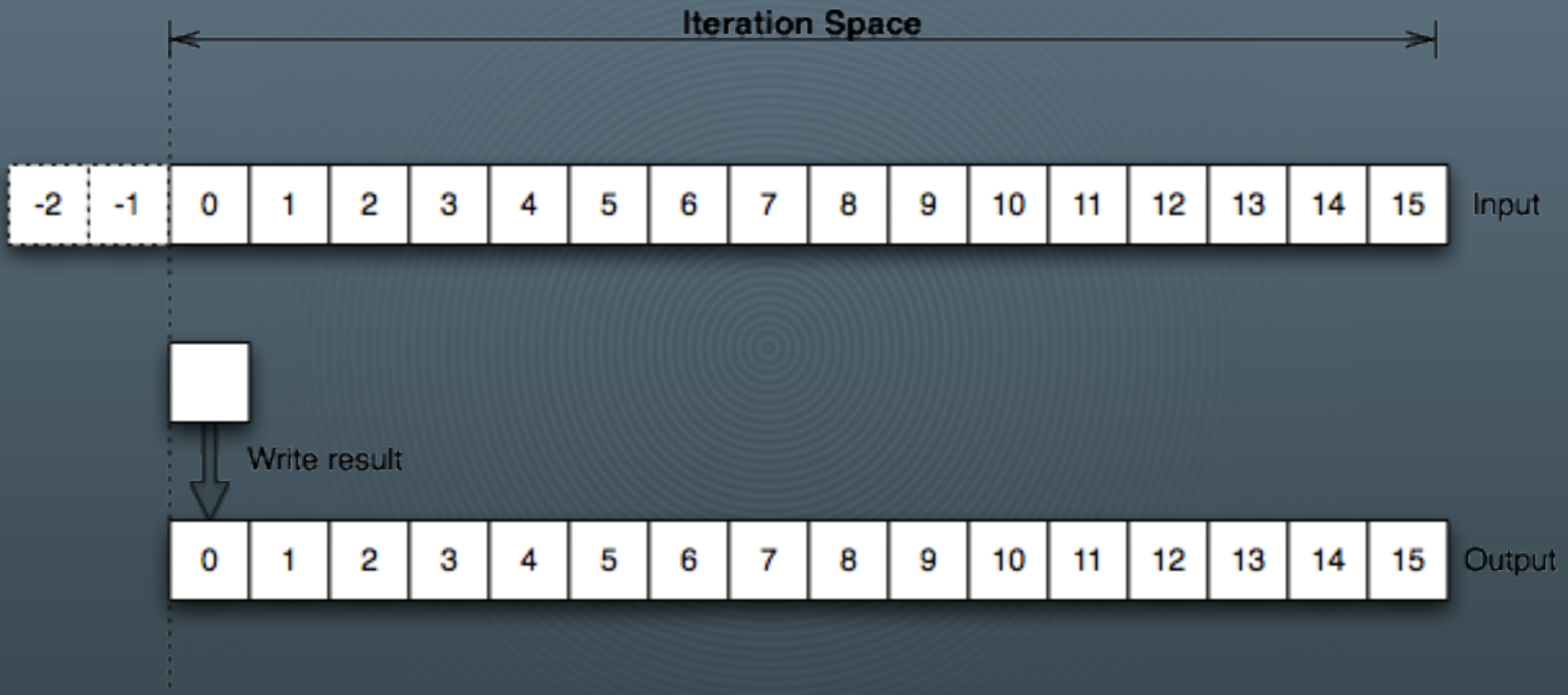
- Process zero or more input images to one or more output images,
  - any number of inputs or outputs
  - arbitrary access specifications on images
  - no dependencies between points in the iteration space

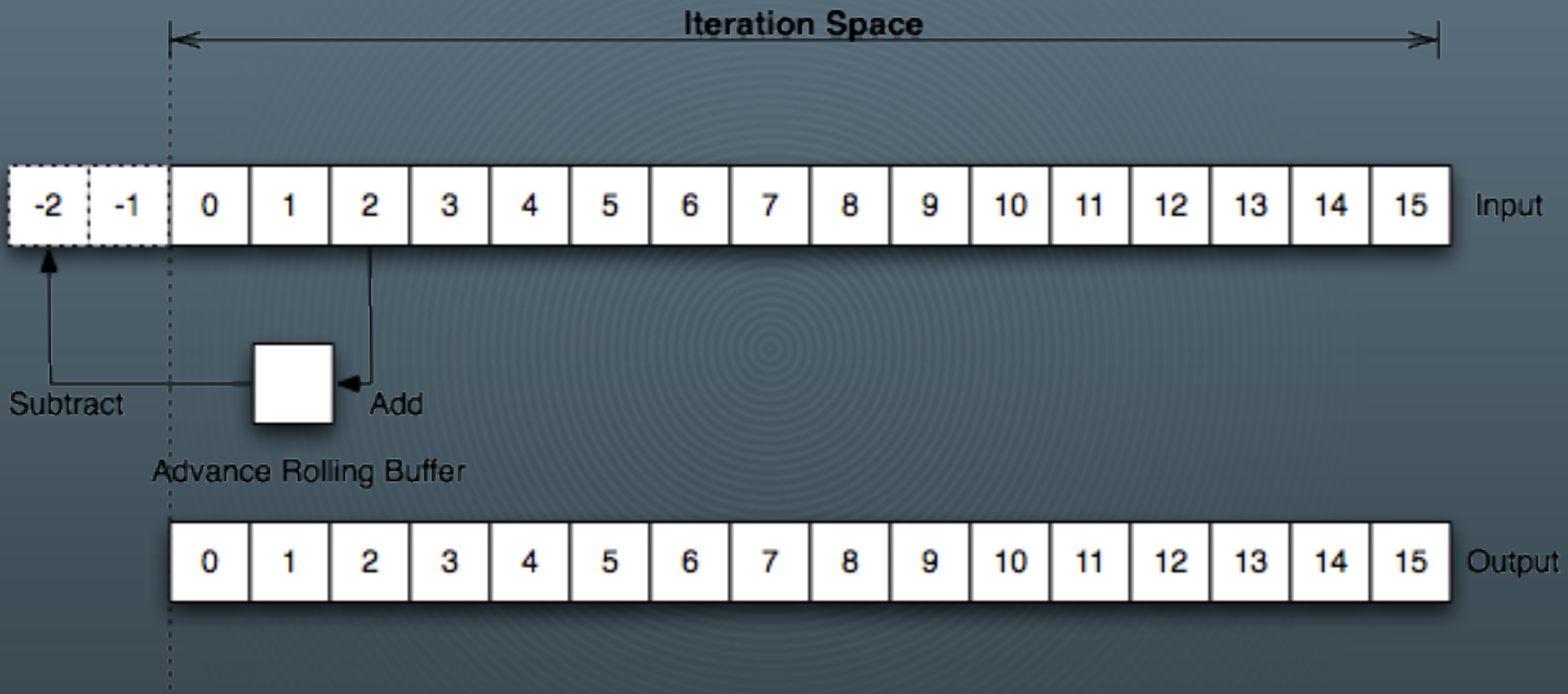
hp NVIDIA. fxguide THE FOUNDRY THE FOUNDRY @SIGGRAPH

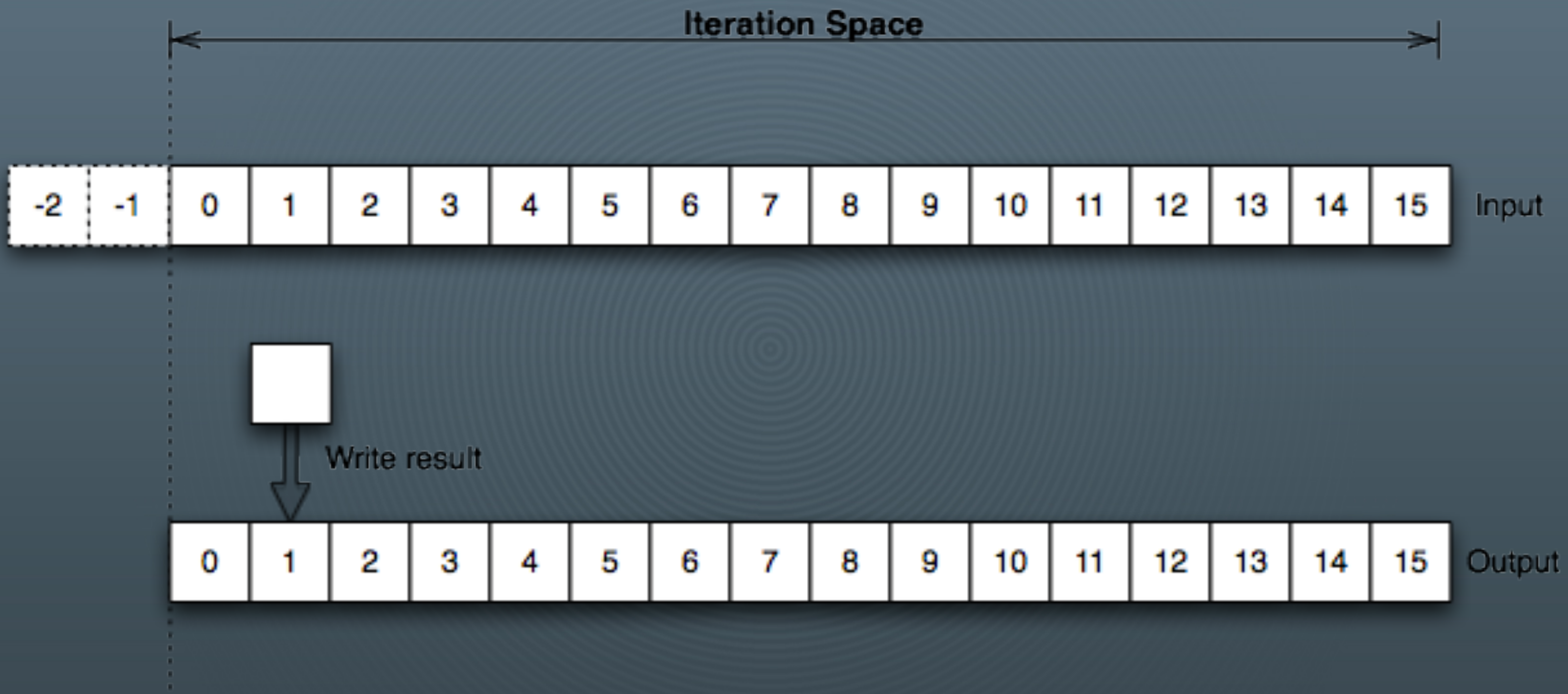
Carry Dependencies

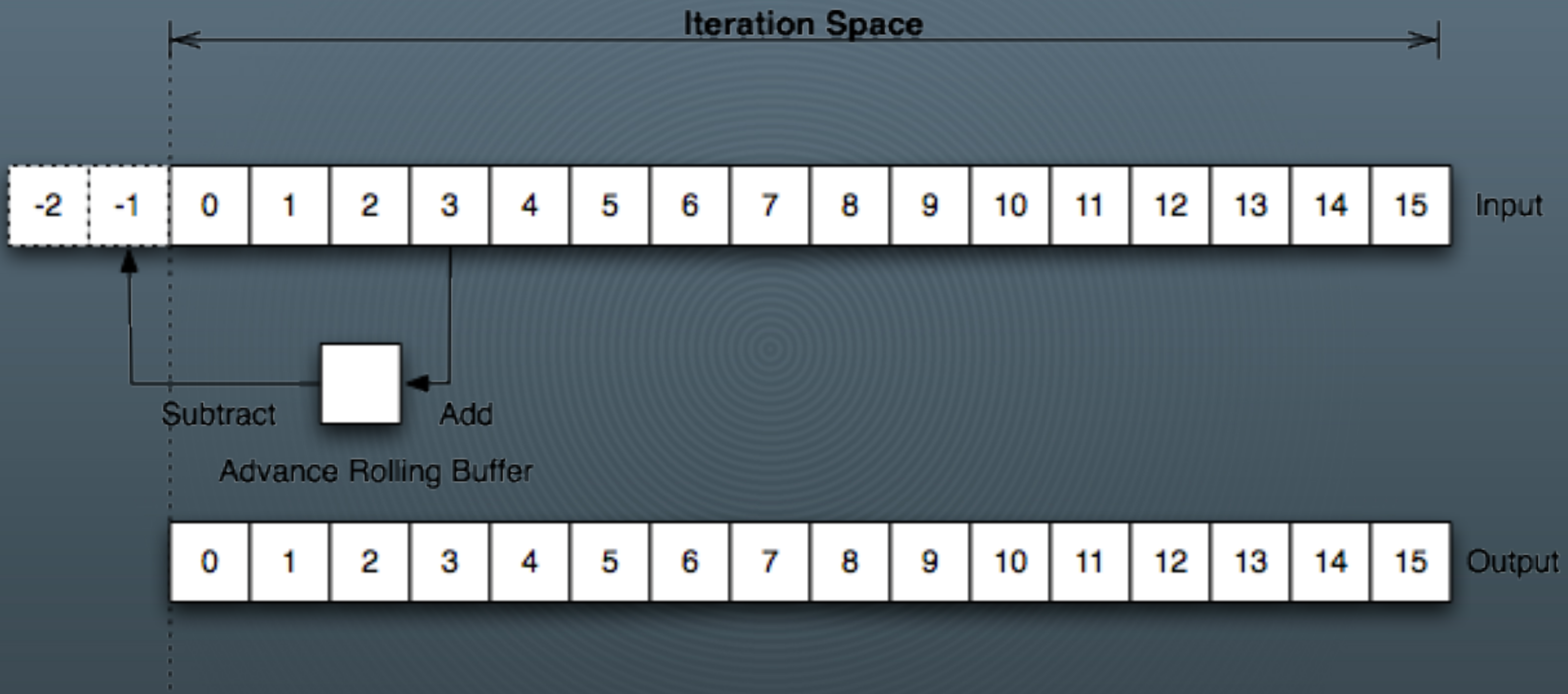
- RIP allows for data carry between points in the iteration space
  - classic use case is the rolling buffer box blur
  - We make a distinction between
    - local carries, eg: box blur
    - full carries, eg: analysis algorithms

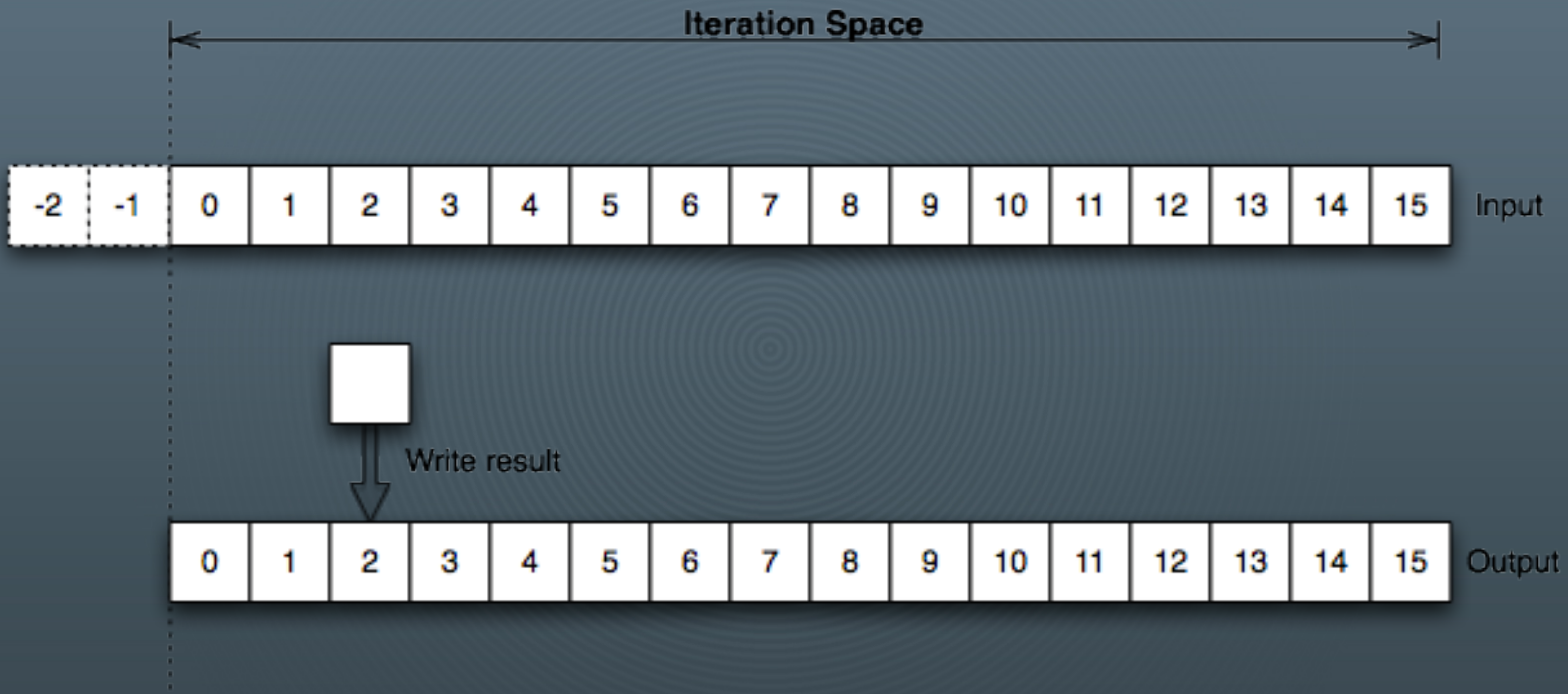

Iteration Space
-2
-1
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Input
Initialise Rolling Buffer
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Output


Iteration Space
-2
-1
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Input
Write result
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Output


Iteration Space
-2
-1
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Input
Subtract
Add
Advance Rolling Buffer
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Output


Iteration Space
-2
-1
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Input
Write result
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Output

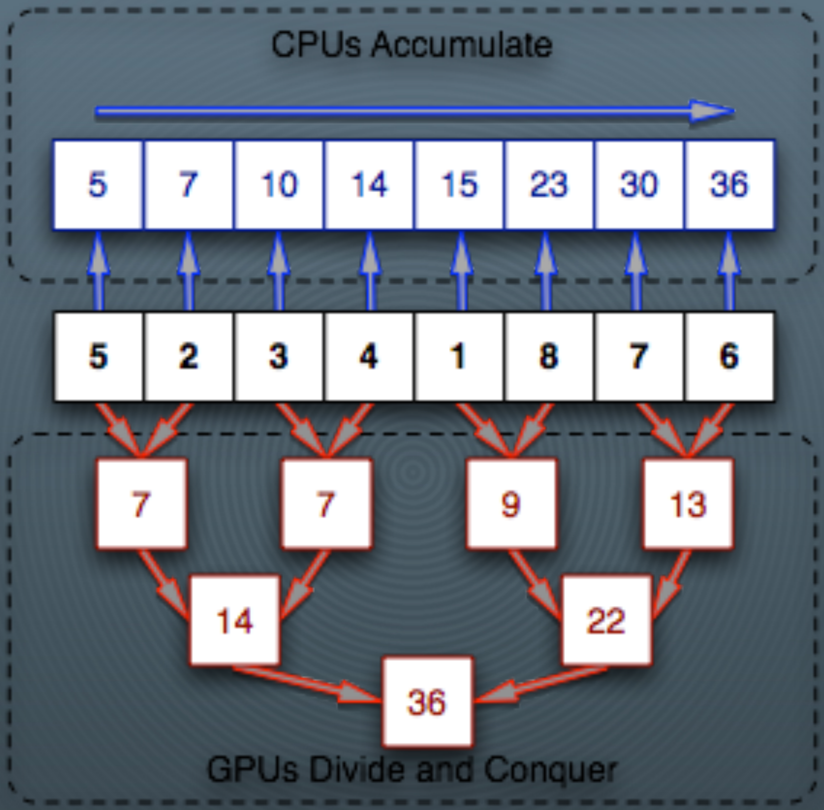

Iteration Space
-2
-1
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Input
Subtract
Add
Advance Rolling Buffer
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Output


Iteration Space
-2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Input
Write result
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Output

Reductions

- Reductions combine all elements in a data structure in some way
  - e.g. find the sum of all the pixels in an image

THE FOUNDRY @SIGGRAPH


CPUs Accumulate
5
7
10
14
15
23
30
36
5
2
3
4
1
8
7
6
7
7
9
13
14
22
36
GPUs Divide and Conquer

Problems with Reductions

- Floating point precision is finite

```
(a+b)+(c+d)  !=  ((a+b)+c)+d
```

- Different ordering produces different results!

DEMO – NUKE proto-typing plugin

- Simple kernels
- Introspection
- Run-time code generation

DEMO –Real World Algorithms in NUKEX

- NUKE nodes ported to RIP
- CUDA for GPU, x86 for CPU
- Non-trivial image processing:
  - Depth of field, motion estimation based retiming, motion blur, denoising, convolution.

Porting RIP research to NUKE

- Dealing with large image sizes and finite GPU memory
  - NUKE CPU unit of work is one scanline
  - GPU favours bigger unit of work because of more cores

Post Process Depth of Field

- Old CPU approach was brute force convolution
- GPU port of CPU approach would hang GPU on large convolution kernels
- Moved to FFT approach, both on CPU (MKL) and GPU (CUDA FFT)
- RIP kernels used to resize convolution kernels, process layers, do some special sauce processing to reduce artifacts

Future Work I

- Beef up our RIP processing graph
  - schedule CPU/GPU computation
  - stream inputs and outputs with CUDA

Future Work II

- Kernel Fusion
  - munging multiple RIP kernels together to reduce memory access
  - not just point wise kernels, but complex ones as well
    - by exploiting explicit data dependencies

- More caching
  - Deal with the IO bottle neck with fast IO. Eg FusionFX

What We Learnt

- Clang/LLVM rocks basis of our parsing and runtime x86 support
- Breaking CPU/GPU agreement is occasionally necessary
- Transfer times can be the killer, Kepler will help with this