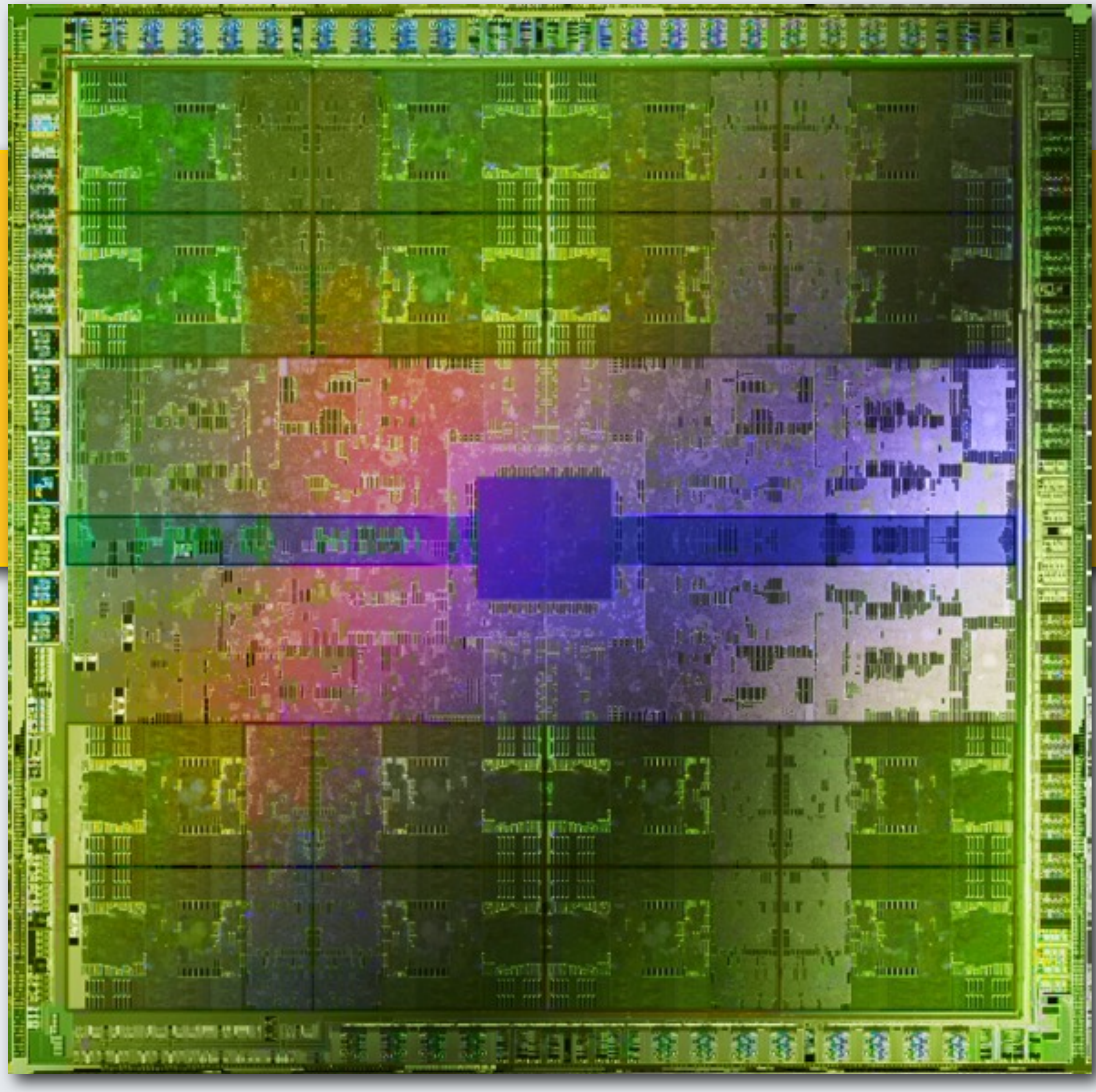


Developing GPU Enabled Visual Effects For Film And Video

Bruno Nicoletti

Founder and CTO

About Me

- I am the founder and CTO of The Foundry
- Worked in CG and VFX since completing my CS degree in 1987
  – when you rolled your own
- Worked in production on effects and animations
  – Zap, Rushes, Computer Film Company, Animal Logic
- Worked at software houses making commercial VFX software
  – Discreet Logic, Softimage, Animal Logic
- Since starting The Foundry, concentrated on image processing for VFX

About The Foundry

- The Foundry is a developer of VFX software for film and TV
- Academy Award winning software used on 8 of the 10 highest grossing movies of all time
- Main emphasis has been on compositing tools
  - Nuke, Ocula, Furnace, Keylight
- Recently branching out
  - Mari - CGI texture paint tool
  - Katana - CGI lighting tool (still in development)
  - Storm - realtime digital cinematography tool (still in development)

Visual Effects

- 'Stuff' done to images after live action shooting
- Pretty much all digital now
- Traditionally part of post-production, alongside editing etc...
- Not just giant killer robots and big explosions
  - replacing practicals and live action elements
  - used for fixup/repair/replacement
  - used for mood

Simplified Production Workflow

Script

Simplified Production Workflow

Script → Preproduction

- Storyboarding
- Design
- Blocking
- Pre-vis
- Set Build

Simplified Production Workflow

Script → Preproduction → Production

Preproduction

- Storyboarding
- Design
- Blocking
- Pre-vis
- Set Build

Production

- Shooting stuff
- Review
- On-set vis

Simplified Production Workflow

Script → Preproduction → Production → Postproduction

Preproduction
- Storyboarding
- Design
- Blocking
- Pre-vis
- Set Build

Production
- Shooting stuff
- Review
- On-set vis

Postproduction
- Editing
- CGI
- Compositing
- Colour Grading

Simplified VFX Post Workflow

Simplified VFX Post Workflow

CGI

- Modelling
- Animation
- Simulation
- Rendering

Simplified VFX Post Workflow

CGI → **Composite**

CGI
- Modelling
- Animation
- Simulation
- Rendering

Composite
- Painting
- Fix-ups
- Layering
- Effects

Simplified VFX Post Workflow

CGI → Composite → Grade

CGI

- Modelling
- Animation
- Simulation
- Rendering

Composite

- Painting
- Fix-ups
- Layering
- Effects

Grade

- Colour correct
- Simple effects

Movie of VFX Breakdown Goes Here

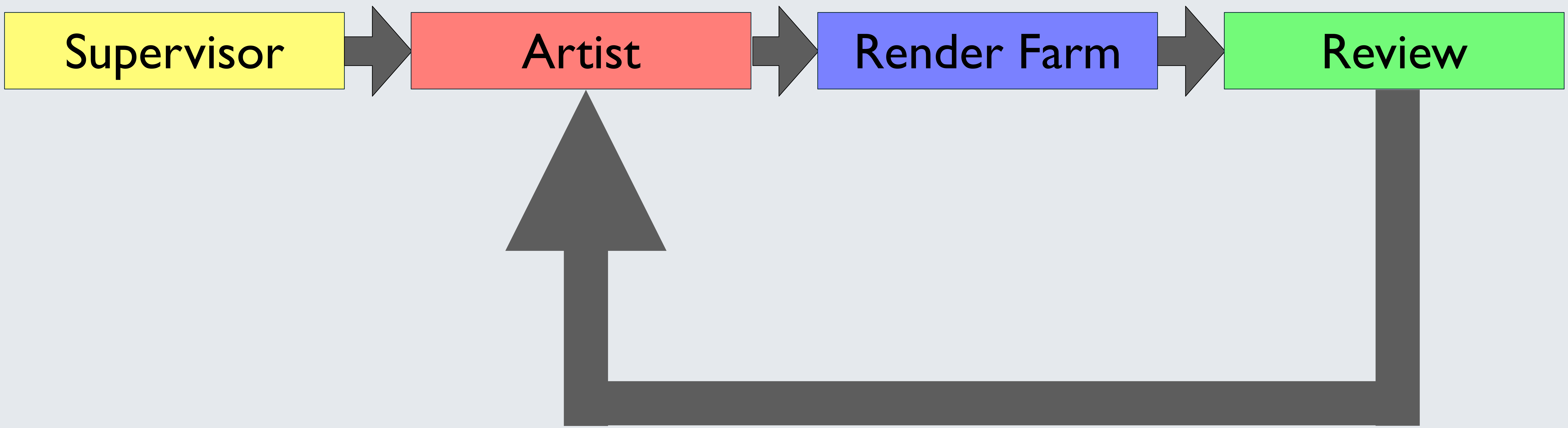
Avatar Breakdown, Courtesy of Twentieth Century Fox and Weta Digtal

Avatar Breakdown, Courtesy of Twentieth Century Fox and Weta Digtal

VFX Workflows I

- Highly collaborative
  - many people working on many stages of the production
- Highly pipelined
  - modelling feeds to animation feeds to rendering feeds to comp
- Highly iterative
  - many passes required to achieve the final results
  - iterative both within and between stages

VFX Workflows II



Artists

- In larger houses, artists tend to be specialised
- In smaller shops, tend to be more generalists
- Typically equipped with a high spec workstation
  – big CPU, big GPU and big disks
- Sits on a fast network with SANs and access to a render farm
- Puts project together, and previews several frames
  – **low latency is key**
- Batches rest of sequence off to render farm to finish

Rendering

- Terabytes of data and days of compute can go into a single frame
  - **throughput is key**
- Currently achieved by servers, and lots of them
  - Weta Digital used 40,000 cores to render Avatar
- Simulation, CGI and compositing computed on render farms
- CPUs are almost exclusively used for rendering
  - early days for GPU rendering software
  - will be hard to GPU everything, CPUs here for a while yet

VFX Compute Ecosystems

- We have little control over the hardware our users buy
  - unlike a dedicated HPC centre
- They have a varied set of computers including...
  - workstations with big GPUs and big CPUs
  - render farms with no GPUs and big CPUs
  - laptops with incy CPUs and smaller GPUs
  - everything between
- They expect our software to make the same pictures on all of them

What GPUs are doing to VFX

- Increased performance from GPUs is starting to...
  - reducing time and cost of render/review iterations
  - give ‘realtime’ VFX in some cases, removing the need for renders
  - allow for more complex effects
    - render times seem to stay constant despite the available FLOPs
  - allow VFX to be used more pervasively throughout production
  - blurs stages of production
    - post increasingly being brought into production

The Foundry's Compositing Software

- Currently specialise in image processing for compositing
  - Nuke - feature rich compositing application
- Specialist plug-in created by dedicated research team
  - Furnace - motion estimation based tools for compositing
  - Ocula - tools for stereo compositing
  - CameraTracker - computes camera position in live action shot
- Mostly CPU based, but we are starting to exploit the GPU
- Also being used in pre-production and actual production

GPUs Come of Age For Image Processing

- Advent of CUDA/OpenCL has allowed for complex image processing
  - many algorithms not possible with GPGPU approach
  - e.g. motion estimation, a key piece of Foundry IP
- We have a fantastic opportunity to improve our software
  - to reduce latency for the artist
  - to increase throughput on renders
  - use it in new situations
  - do cool new stuff

Developing GPU Enabled Effects I

- Why not 'dive-in' and develop GPU enabled effects?
- We have to have a CPU compute path
  - for CPU based render farms
  - for old or slow GPUs
- CPUs have FLOPs we should use even if there is a decent GPU
- CPU and GPU results must agree
  - not truly possible due to nature of the hardware
  - visually indistinguishable is the metric we want

Developing GPU Enabled Effects II

- Writing separate CPU and GPU implementations is
  - twice the effort to implement
  - easy enough for simple algos to agree, e.g. brightness effect
  - practically impossible to make sure complicated algorithms agree
    - where much of our bread and butter is
  - horribly difficult to debug and maintain agreement

Developing GPU Enabled Effects III

- Getting peak performance is a specialist task
  - You need to do it differently per device
  - Hand optimisation gets in the way of writing algorithms
  - My researchers aren't performance engineers
- How do you deal with new hardware or new optimisation techniques?
  - Hand crafting code locks you in
  - Need to individually recode everything = expensive

Don’t Go There

- We have hundreds of effects and millions of lines of code
- Will need to rewrite all of them to exploit GPUs
- An ad-hoc approach to exploiting GPUs will not scale
  – it be slow to deliver anything
  – it would increase development costs
  – it would be a nightmare to maintain
- So we chose not to go that route

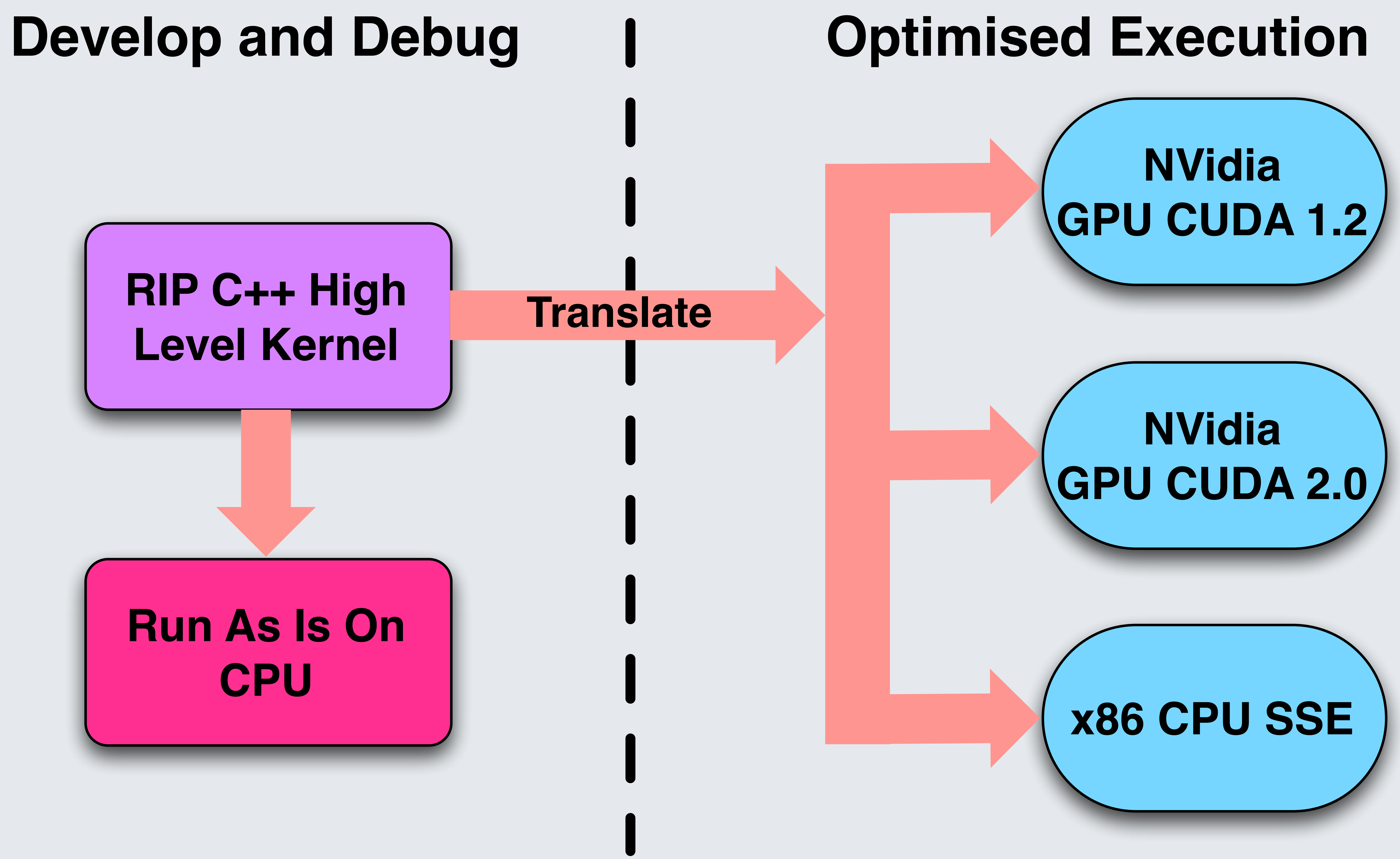
Introducing 'Blink'

- Or "Righteous Image Processing", RIP, as we call it internally
- Project to deliver a multi-device image processing framework
- Allows us to exploit GPUs and CPUs and avoid those problems
- Based on work done with Imperial College London
- And it works
  - we have shipping software based on it
  - the gnomes at home are writing more as I speak

RIP Overview

- RIP wraps image processing up into a high level **C++** API
- Programmer writes special C++ classes to do their work
- These are device independent and clear expressions of an algorithm
- At compile time, we translate those classes into specific implementations for each device we support
- Programmer can also run untranslated kernel as-is on the CPU,
  - for easy debugging and development

RIP Workflow



Doesn’t OpenCL Do That?

- OpenCL gives you a multi-device programming framework
- But memory and compute behave different between devices
  - you can’t forget that with OpenCL
- To get any performance, you still need to code differently per device
- OpenCL makes a good back end for RIP however
  - but still a young technology with immature drivers

Data Dependence Is Key To Parallelism

- Parallelism is where all the FLOPs now are
- Algorithm's data dependence is what constrains its parallelism
- Traditional implementations obscure that data dependence
- Making data dependence explicit = analysis free knowledge of parallelism
- Knowing that you can
  – map algorithm to devices in appropriate manner
  – allows for inter algorithm optimisations

RIP Basic Design

- Purely for image processing
- Application of map/reduce for that domain, with some extras
- Access to all data is abstracted and made explicit
  - images
  - reductions
  - carry dependence
- Programmer never given direct access to or ownership of the data
  - always controlled by the framework

RIP Kernel

- Abstraction of a single pass image processing operation
- Implicit 3D iteration space, (X and Y ranges + N components)
- Explicit declaration of how data is accessed at each point in space
  – rich set of access specifications
- A function is executed once at each point in the iteration space
  – in which you only have restricted access to data
  – and read only access to class members
- A bit like a high level version of a GPU kernel for image processing

```
class InvertKernel : public Kernel2<eComponentWise,
                                    AccessSpec<TapAccess,  eRead,   eComponentWise>,
                                    AccessSpec<TapAccess,  eWrite,  eComponentWise> >
{
public:
  template <class SRC, class DST>
  void kernel(SRC &src, DST &dst, const IterationPosition &)
  {
    *dst = DST::clamp(DST::kWhitePoint - *src);
  }
};

void InvertImage(Compute::Image &source, Compute::Image &destination)
{
  InvertKernel inverter;
  destination.device().iterate(inverter, source, destination);
}
```

Trivial Example

Access Spec

```
class InvertKernel : public Kernel2<eComponentWise,
                                AccessSpec<TapAccess,  eRead,   eComponentWise>,
                                AccessSpec<TapAccess,  eWrite,  eComponentWise> >

{
public:
  template <class SRC, class DST>
  void kernel(SRC &src, DST &dst, const IterationPosition &)
  {
    *dst = DST::clamp(DST::kWhitePoint - *src);
  }
};

void InvertImage(Compute::Image &source, Compute::Image &destination)
{
  InvertKernel inverter;
  destination.device().iterate(inverter, source, destination);
}
```

Trivial Example

```
class InvertKernel : public Kernel2<eComponentWise,
                                    AccessSpec<TapAccess,  eRead,   eComponentWise>,
                                    AccessSpec<TapAccess,  eWrite,  eComponentWise> >
{
public:
  template <class SRC, class DST>
  void kernel(SRC &src, DST &dst, const IterationPosition &)
  {
    *dst = DST::clamp(DST::kWhitePoint - *src);
  }
};

void InvertImage(Compute::Image &source, Compute::Image &destination)
{
  InvertKernel inverter;
  destination.device().iterate(inverter, source, destination);
}
```

Trivial Example

```
class InvertKernel : public Kernel2<eComponentWise,
                                    AccessSpec<TapAccess,  eRead,   eComponentWise>,
                                    AccessSpec<TapAccess,  eWrite,  eComponentWise> >
{
public:
  template <class SRC, class DST>
  void kernel(SRC &src, DST &dst, const IterationPosition &)
  {
    *dst = DST::clamp(DST::kWhitePoint - *src);
  }
};

void InvertImage(Compute::Image &source, Compute::Image &destination)
{
  InvertKernel inverter;
  destination.device().iterate(inverter, source, destination);
}
```

Kernel Body

Trivial Example

```
class InvertKernel : public Kernel2<eComponentWise,
                                    AccessSpec<TapAccess,  eRead,   eComponentWise>,
                                    AccessSpec<TapAccess,  eWrite,  eComponentWise> >
{
public:
  template <class SRC, class DST>
  void kernel(SRC &src, DST &dst, const IterationPosition &)
  {
    *dst = DST::clamp(DST::kWhitePoint - *src);
  }
};

void InvertImage(Compute::Image &source, Compute::Image &destination)
{
  InvertKernel inverter;
  destination.device().iterate(inverter, source, destination);
}
```

Trivial Example

```
class InvertKernel : public Kernel2<eComponentWise,
                                    AccessSpec<TapAccess,  eRead,   eComponentWise>,
                                    AccessSpec<TapAccess,  eWrite,  eComponentWise> >
{
public:
  template <class SRC, class DST>
  void kernel(SRC &src, DST &dst, const IterationPosition &)
  {
    *dst = DST::clamp(DST::kWhitePoint - *src);
  }
};

void InvertImage(Compute::Image &source, Compute::Image &destination)
{
  InvertKernel inverter;
  destination.device().iterate(inverter, source, destination);
}
```

Accessors

Trivial Example

```
class InvertKernel : public Kernel2<eComponentWise,
                                    AccessSpec<TapAccess,  eRead,   eComponentWise>,
                                    AccessSpec<TapAccess,  eWrite,  eComponentWise> >
{
public:
  template <class SRC, class DST>
  void kernel(SRC &src, DST &dst, const IterationPosition &)
  {
    *dst = DST::clamp(DST::kWhitePoint - *src);
  }
};

void InvertImage(Compute::Image &source, Compute::Image &destination)
{
  InvertKernel inverter;
  destination.device().iterate(inverter, source, destination);
}
```

Trivial Example

```
extern "C" void __global__
CUDA_GPU_InvertKernel_kernel_unsigned_char_1_unsigned_char_1(
             int4 const _fc_dod, int const _fc_c,
             unsigned char const *const src, int4 const src_bounds, int3 const src_steps,
             unsigned char *const dst, int4 const dst_bounds, int3 const dst_steps)
{
  int2 const _fc_pos = _fc_compute_pos();
  if (_fc_pos.x < _fc_dod.z && _fc_pos.y < _fc_dod.w) {
    _fc_component(dst, uchar, _fc_pos.x, _fc_pos.y, _fc_c)
       = clamp((unsigned char)((255U -
                                 _fc_component(src, uchar, _fc_pos.x, _fc_pos.y, _fc_c))),
               (unsigned char)((0)),
               (unsigned char)((255)));
  }
}
```

Equivalent generated CUDA kernel, one of 32 variants.

One Of Many Generated

```
extern "C" void __global__
CUDA_GPU_InvertKernel_kernel_unsigned_char_1_unsigned_char_1(
             int4 const _fc_dod, int const _fc_c,
             unsigned char const *const src, int4 const src_bounds, int3 const src_steps,
             unsigned char *const dst, int4 const dst_bounds, int3 const dst_steps)
{
  int2 const _fc_pos = _fc_compute_pos();
  if (_fc_pos.x < _fc_dod.z && _fc_pos.y < _fc_dod.w) {
    _fc_component(dst, uchar, _fc_pos.x, _fc_pos.y, _fc_c)
      = clamp((unsigned char)((255U -
                               _fc_component(src, uchar, _fc_pos.x, _fc_pos.y, _fc_c))),
               (unsigned char)((0)),
               (unsigned char)((255)));
  }
}
```

Equivalent generated CUDA kernel, one of 32 variants.

```
extern "C" void __global__
CUDA_GPU_InvertKernel_kernel_unsigned_char_1_unsigned_char_1(
             int4 const _fc_dod, int const _fc_c,
             unsigned char const *const src, int4 const src_bounds, int3 const src_steps,
             unsigned char *const dst, int4 const dst_bounds, int3 const dst_steps)
{
  int2 const _fc_pos = _fc_compute_pos();
  if (_fc_pos.x < _fc_dod.z && _fc_pos.y < _fc_dod.w) {
    _fc_component(dst, uchar, _fc_pos.x, _fc_pos.y, _fc_c)
       = clamp((unsigned char)((255U -
                            _fc_component(src, uchar, _fc_pos.x, _fc_pos.y, _fc_c))),
               (unsigned char)((0)),
               (unsigned char)((255)));
  }
}
```

Equivalent generated CUDA kernel, one of 32 variants.

```
extern "C" void __global__
CUDA_GPU_InvertKernel_kernel_unsigned_char_1_unsigned_char_1(
             int4 const _fc_dod, int const _fc_c,
             unsigned char const *const src, int4 const src_bounds, int3 const src_steps,
             unsigned char *const dst, int4 const dst_bounds, int3 const dst_steps)
{
  int2 const _fc_pos = _fc_compute_pos();
  if (_fc_pos.x < _fc_dod.z && _fc_pos.y < _fc_dod.w) {
    _fc_component(dst, uchar, _fc_pos.x, _fc_pos.y, _fc_c)
       = clamp((unsigned char)((255U -
                                 _fc_component(src, uchar, _fc_pos.x, _fc_pos.y, _fc_c))),
               (unsigned char)((0)),
               (unsigned char)((255)));
  }
}
```

Translate Function Body

Equivalent generated CUDA kernel, one of 32 variants.

```
extern "C" void __global__
CUDA_GPU_InvertKernel_kernel_unsigned_char_1_unsigned_char_1(
             int4 const _fc_dod, int const _fc_c,
             unsigned char const *const src, int4 const src_bounds, int3 const src_steps,
             unsigned char *const dst, int4 const dst_bounds, int3 const dst_steps)
{
  int2 const _fc_pos = _fc_compute_pos();
  if (_fc_pos.x < _fc_dod.z && _fc_pos.y < _fc_dod.w) {
    _fc_component(dst, uchar, _fc_pos.x, _fc_pos.y, _fc_c)
       = clamp((unsigned char)((255U -
                                _fc_component(src, uchar, _fc_pos.x, _fc_pos.y, _fc_c))),
               (unsigned char)((0)),
               (unsigned char)((255)));
  }
}
```

Equivalent generated CUDA kernel, one of 32 variants.

Not Quite C++

- C++ is a very rich and flexible language
  - the reason we chose it to express our kernels
- However to code translate we only use restricted subset in a kernel
  - native C types, e.g. int, float, char etc...
  - 'blessed' types and functions, e.g. RIP::Vec2f, cos, fabs etc....
  - any purely inlined function, POD type or simple class
  - no recursion
- Aggregate types (ie: std::vector like) are a work in progress

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, reverse,
- Edge conditions.

"Ordinary" Kernels

- The 'easy' case,
- Process zero or more input images to one or more output images,
  - any number of inputs or outputs
  - arbitrary access specifications on images
    - can get very complex with the variety of access pattern we have
  - no dependencies between points in the iteration space

Reductions

- Reductions combine all elements in a data structure in some way
  - e.g. find the sum of all the pixels in an image
- RIP can perform associative reductions
  - done via explicit RIP::Kernel::Reduction abstraction class
- Object being reduced into is given to the kernel
  - making data independent to the kernel
- Allows for appropriate parallelisation on each device
  - including shared memory usage on the GPU

```
class SumKernel : public Kernel1<eComponentWise,
                                 AccessSpec<TapAccess, eRead, eComponentWise> >
                , public Reduction<PerComponentReductionData<float> >
{
  public:
    template <class SRC>
    void reduce(SRC &src,
                PerComponentReductionData<float> &reductionData,
                const IterationPosition &pos) const
    {
      reductionData.addSample(pos.component(), float(*src));
    }
}
```

Summation Reduction Code Example

```
class SumKernel : public Kernel1<eComponentWise,
                                 AccessSpec<TapAccess, eRead, eComponentWise> >
                  public Reduction<PerComponentReductionData<float> >
{
  public:
    template <class SRC>
    void reduce(SRC &src,
                PerComponentReductionData<float> &reductionData,
                const IterationPosition &pos) const
    {
      reductionData.addSample(pos.component(), float(*src));
    }
}
```

Class Decorator Specifing Reduction Type

Summation Reduction Code Example

```
class SumKernel : public Kernel1<eComponentWise,
                                  AccessSpec<TapAccess, eRead, eComponentWise> >
                , public Reduction<PerComponentReductionData<float> >
{
  public:
    template <class SRC>
    void reduce(SRC &src,
                PerComponentReductionData<float> &reductionData,
                const IterationPosition &pos) const
    {
      reductionData.addSample(pos.component(), float(*src));
    }
}
```

Summation Reduction Code Example

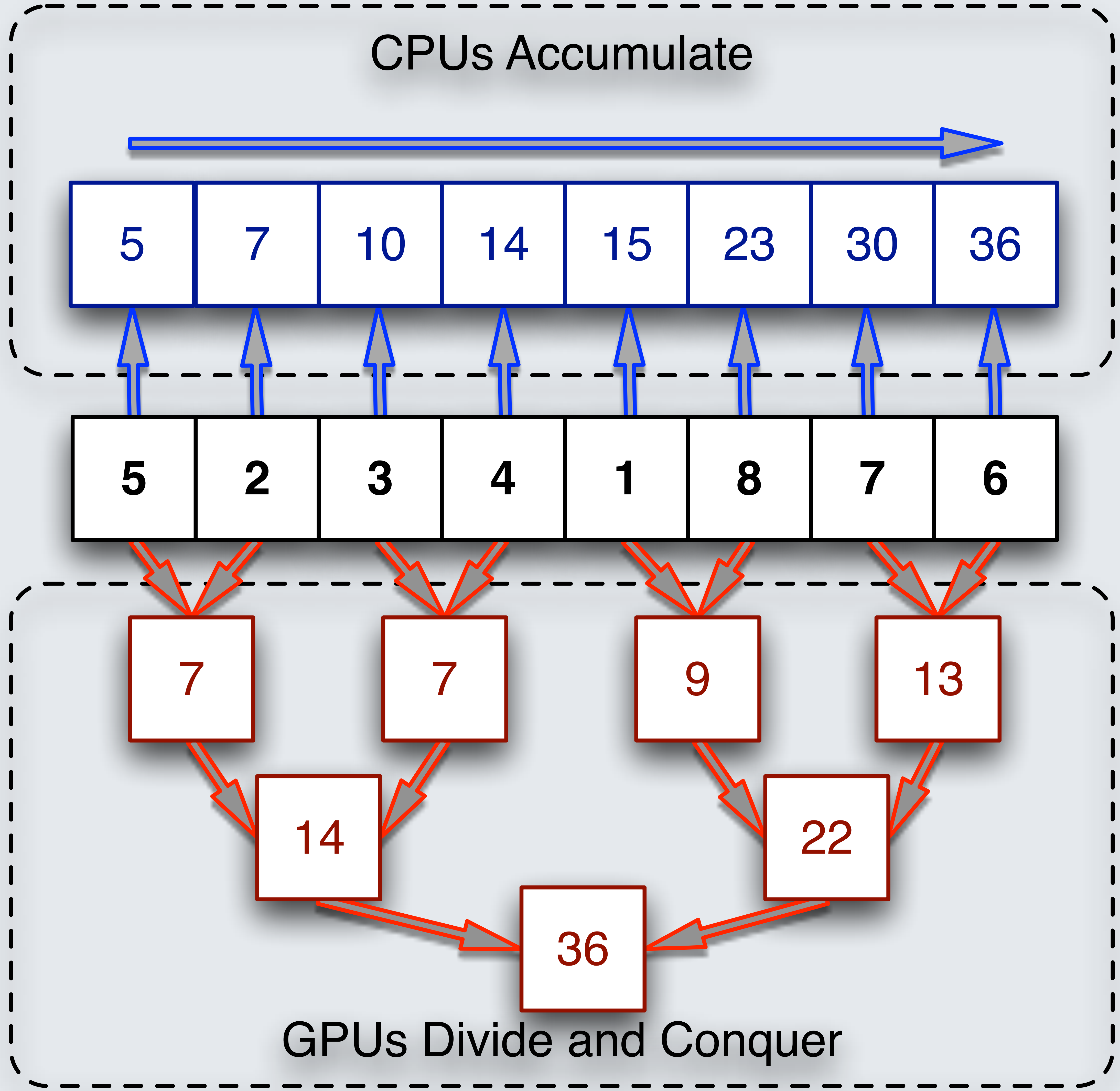
```
class SumKernel : public Kernel1<eComponentWise,
                                 AccessSpec<TapAccess, eRead, eComponentWise> >
                , public Reduction<PerComponentReductionData<float> >
{
  public:
    template <class SRC>
    void reduce(SRC &src,
                PerComponentReductionData<float> &reductionData,
                const IterationPosition &pos) const
    {
      reductionData.addSample(pos.component(), float(*src));
    }
}
```

Reduction Object Is A Parameter

Summation Reduction Code Example

```
class SumKernel : public Kernel1<eComponentWise,
                                AccessSpec<TapAccess, eRead, eComponentWise> >
                , public Reduction<PerComponentReductionData<float> >
{
  public:
    template <class SRC>
    void reduce(SRC &src,
                PerComponentReductionData<float> &reductionData,
                const IterationPosition &pos) const
    {
      reductionData.addSample(pos.component(), float(*src));
    }
}
```

Summation Reduction Code Example

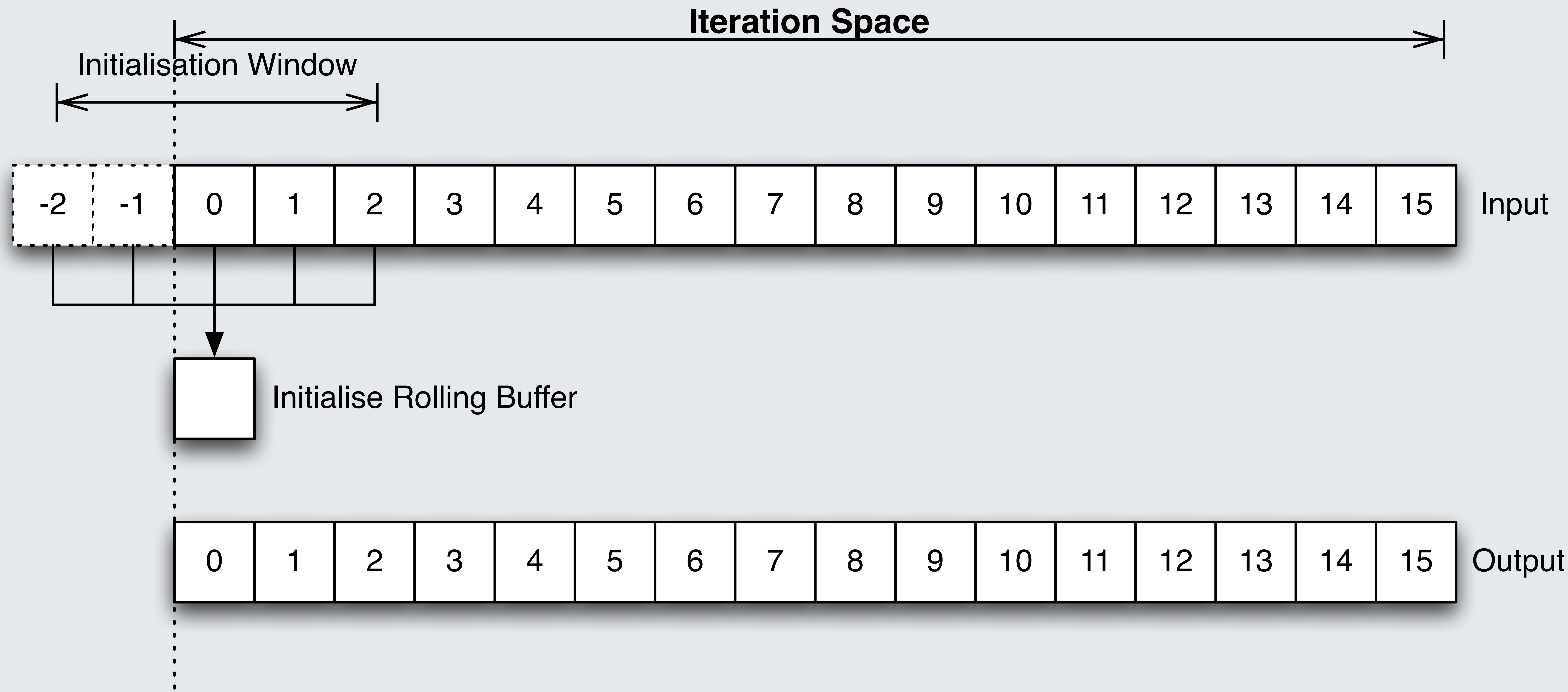

CPUs Accumulate
5
7
10
14
15
23
30
36
5
2
3
4
1
8
7
6
7
7
9
13
14
22
36
GPUs Divide and Conquer

Problems with Reductions

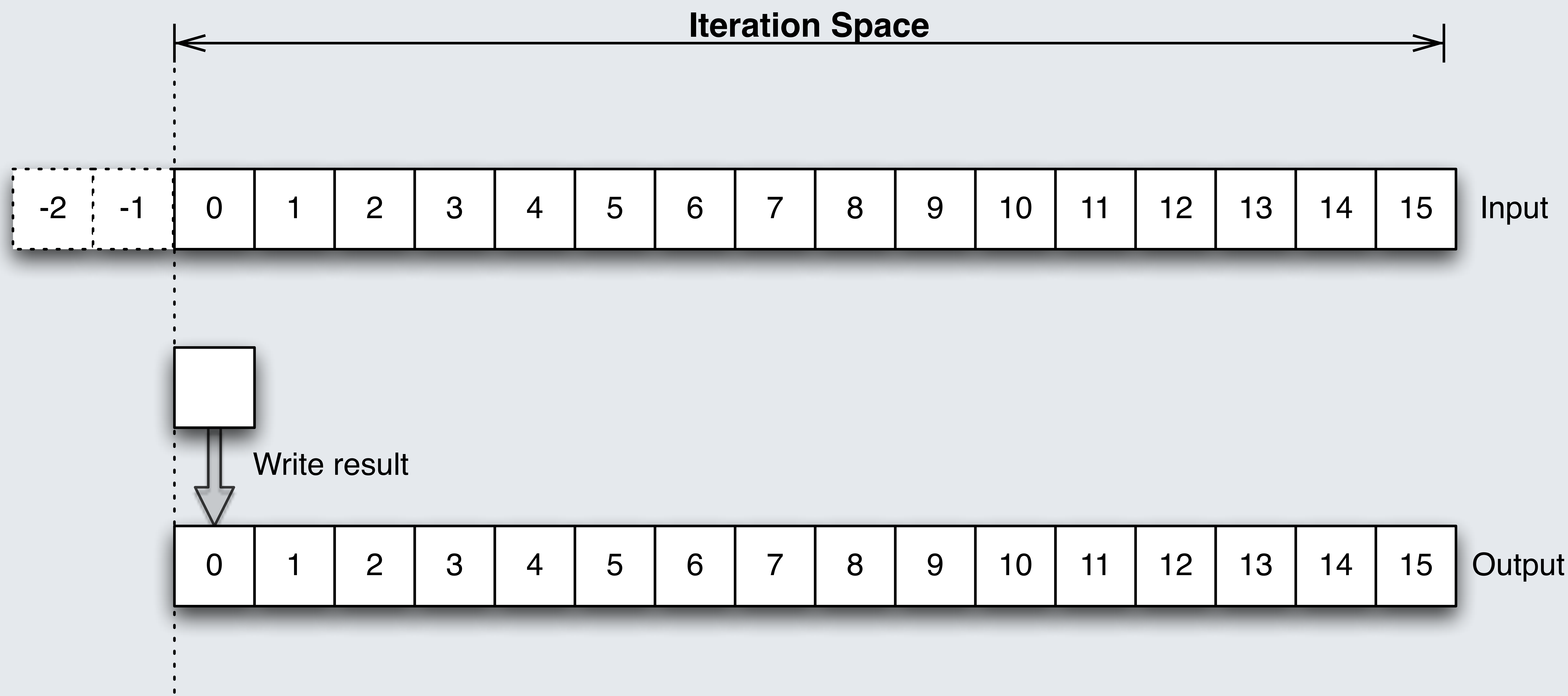
- Floating point precision is finite,
  - which means `(a+b)+(c+d)  !=  ((a+b)+c)+d`
- GPUs and CPUs join their data in different orders
- So CPUs and GPUs reductions will produce different results
  - same problem for parallel reduce on multicore CPUs
- Main source of uncontrollable divergence between devices
- In practice, not that big an issue however, but must be aware of it

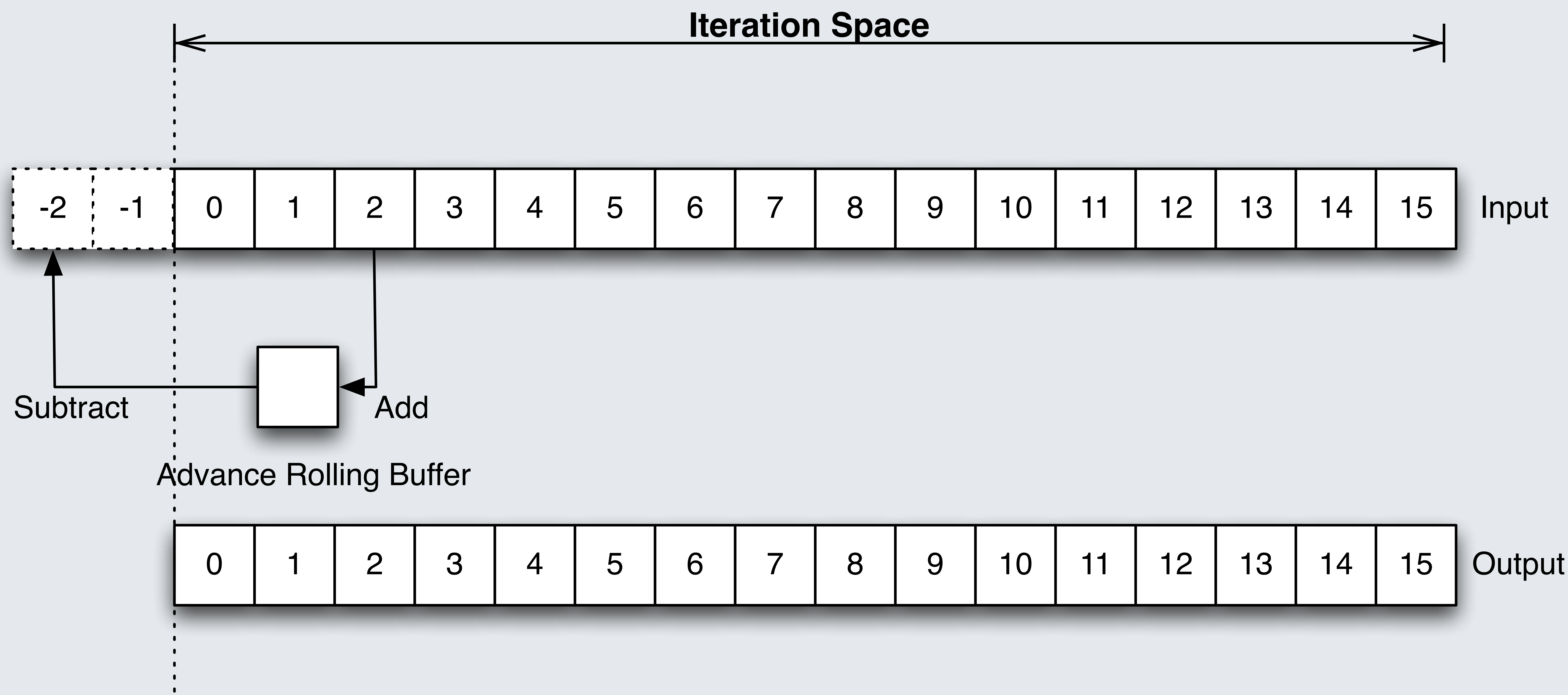
Carry Dependencies I

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

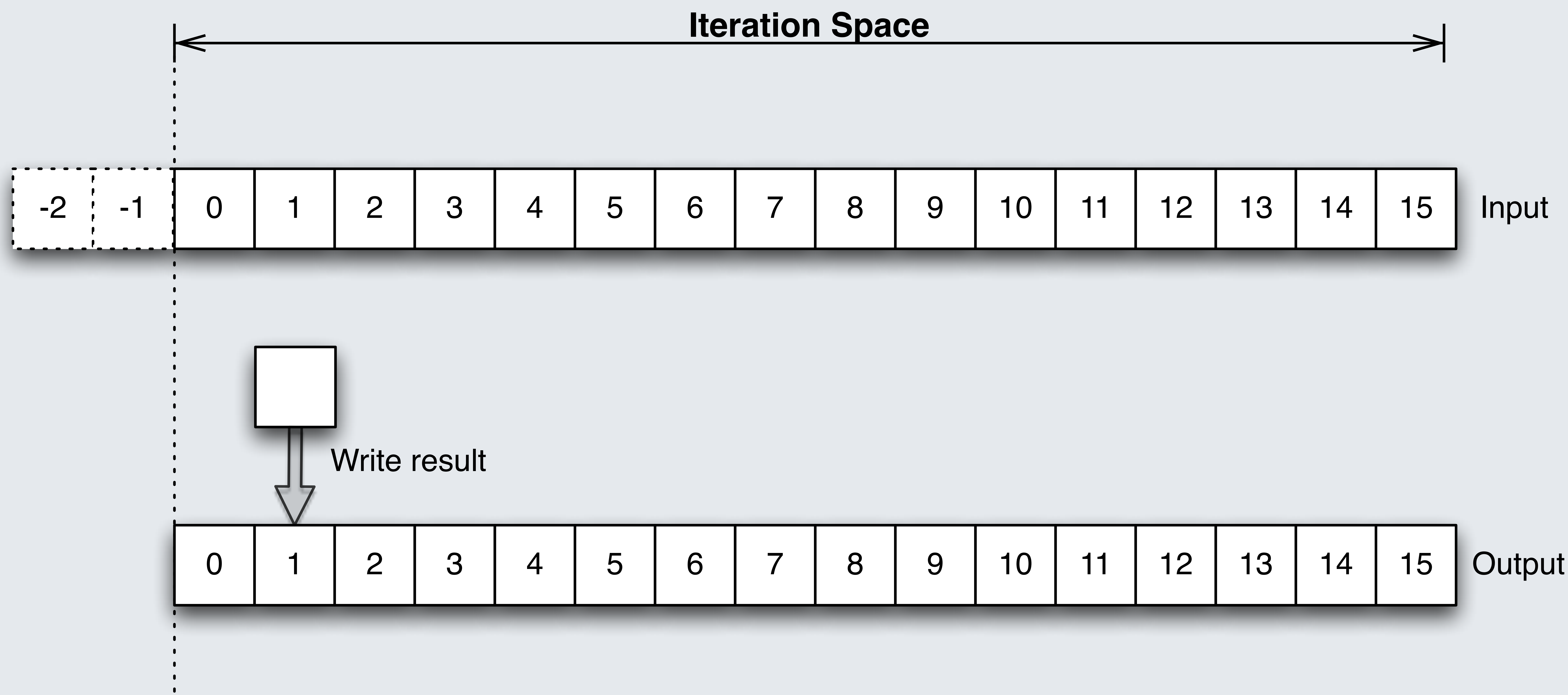


Iteration Space
Initialisation Window
-2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Input
Initialise Rolling Buffer
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Output

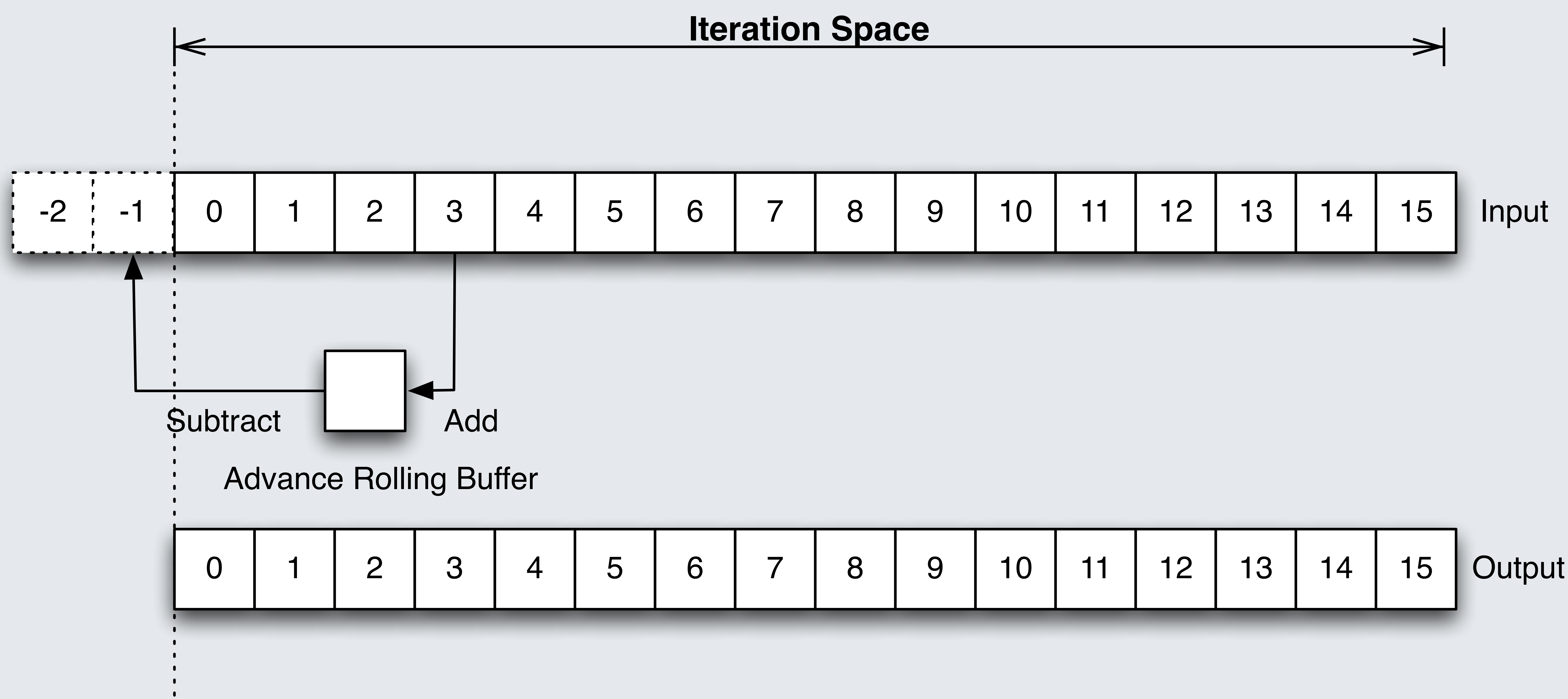

Iteration Space
-2
-1
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Input
Write result
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Output

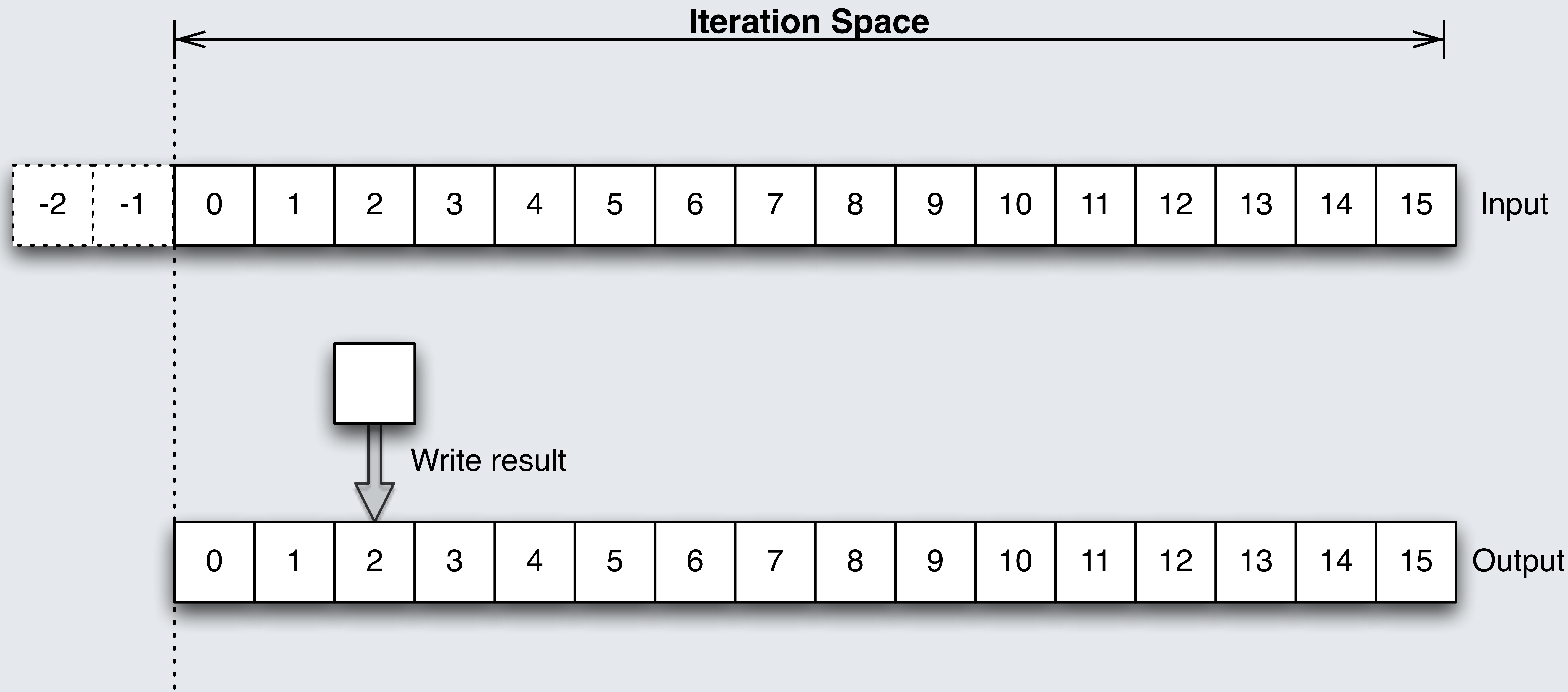


Iteration Space
-2
-1
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Input
Subtract
Add
Advance Rolling Buffer
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Output



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

Carry Dependencies II

- Again, explicitly declare type of data being carried
- Kernels can access images and carried data at each point in iteration
- Carried data has an initialisation window and carry range
- Allows automatic partitioning of the parallelisation
  – for small windows, can parallelise per pixel on GPU
    - by ‘running up’ at each, and writing data out
  – always drag data along row/column on CPU
- Whole row data carries have poorer partitioning

```
class BoxBlurKernel : public Kernel2<eComponentWise,
                                     AccessSpec<Ranged1DAccess, eRead, eComponentWise, ClampedEdge>,
                                     AccessSpec<TapAccess, eWrite, eComponentWise> >
                    , public Rolling<float>
{
  void initialiseRollingData(float &rollingData,
                             const IterationPosition &pos)
  { rollingData = 0; }

  template<class SRC, class DST>
  void rollingRunup(SRC &src, DSTACCESS &dst, float &rollingData,
                    int runupPosition, const IterationPosition &pos)
  { rollingData += src(runupPosition, pos.component()); }

  template <class SRC, class DST>
  void kernel(SRC &src, DST &dst, float &rollingData,
              const IterationPosition &pos)
  {
    float value = rollingData * _filterWidthInv;
    *dst = DSTACCESS::clamp(value);
    rollingData += float(src(_radius+1, pos.component()) - float(src(-_radius, pos.component()));
  }

protected :
  const int _radius;
  const float _filterWidthInv;
};
```

```
class BoxBlurKernel : public Kernel2<eComponentWise,
                                     AccessSpec<Ranged1DAccess, eRead, eComponentWise, ClampedEdge>,
                                     AccessSpec<TapAccess, eWrite, eComponentWise> >
                    , public Rolling<float>
{
  void initialiseRollingData(float &rollingData,
                             const IterationPosition &pos)
  { rollingData = 0; }

  template<class SRC, class DST>
  void rollingRunup(SRC &src, DSTACCESS &dst, float &rollingData,
                    int runupPosition, const IterationPosition &pos)
  { rollingData += src(runupPosition, pos.component()); }

  template <class SRC, class DST>
  void kernel(SRC &src, DST &dst, float &rollingData,
              const IterationPosition &pos)
  {
    float value = rollingData * _filterWidthInv;
    *dst = DSTACCESS::clamp(value);
    rollingData += float(src(_radius+1, pos.component()) - float(src(-_radius, pos.component()));
  }

protected :
  const int _radius;
  const float _filterWidthInv;
};
```

```
class BoxBlurKernel : public Kernel2<eComponentWise,
                                     AccessSpec<Ranged1DAccess, eRead, eComponentWise, ClampedEdge>,
                                     AccessSpec<TapAccess, eWrite, eComponentWise> >
                    , public Rolling<float>
{
  void initialiseRollingData(float &rollingData,
                             const IterationPosition &pos)
  { rollingData = 0; }

  template<class SRC, class DST>
  void rollingRunup(SRC &src, DSTACCESS &dst, float &rollingData,
                    int runupPosition, const IterationPosition &pos)
  { rollingData += src(runupPosition, pos.component()); }

  template <class SRC, class DST>
  void kernel(SRC &src, DST &dst, float &rollingData,
              const IterationPosition &pos)
  {
    float value = rollingData * _filterWidthInv;
    *dst = DSTACCESS::clamp(value);
    rollingData += float(src(_radius+1, pos.component()) - float(src(-_radius, pos.component()));
  }

protected :
  const int _radius;
  const float _filterWidthInv;
};
```

```
class BoxBlurKernel : public Kernel2<eComponentWise,
                                     AccessSpec<Ranged1DAccess, eRead, eComponentWise, ClampedEdge>,
                                     AccessSpec<TapAccess, eWrite, eComponentWise> >
                      , public Rolling<float>
{
  void initialiseRollingData(float &rollingData,
                             const IterationPosition &pos)
  { rollingData = 0; }

  template<class SRC, class DST>
  void rollingRunup(SRC &src, DSTACCESS &dst, float &rollingData,
                    int runupPosition, const IterationPosition &pos)
  { rollingData += src(runupPosition, pos.component()); }

  template <class SRC, class DST>
  void kernel(SRC &src, DST &dst, float &rollingData,
              const IterationPosition &pos)
  {
    float value = rollingData * _filterWidthInv;
    *dst = DSTACCESS::clamp(value);
    rollingData += float(src(_radius+1, pos.component()) - float(src(-_radius, pos.component()));
  }

protected :
  const int _radius;
  const float _filterWidthInv;
};
```

```
class BoxBlurKernel : public Kernel2<eComponentWise,
                                     AccessSpec<Ranged1DAccess, eRead, eComponentWise, ClampedEdge>,
                                     AccessSpec<TapAccess, eWrite, eComponentWise> >
                    , public Rolling<float>
{
  void initialiseRollingData(float &rollingData,
                             const IterationPosition &pos)
  { rollingData = 0; }

  template<class SRC, class DST>
  void rollingRunup(SRC &src, DSTACCESS &dst, float &rollingData,
                    int runupPosition, const IterationPosition &pos)
  { rollingData += src(runupPosition, pos.component()); }

  template <class SRC, class DST>
  void kernel(SRC &src, DST &dst, float &rollingData,
              const IterationPosition &pos)
  {
    float value = rollingData * _filterWidthInv;
    *dst = DSTACCESS::clamp(value);
    rollingData += float(src(_radius+1, pos.component()) - float(src(-_radius, pos.component()));
  }

protected :
  const int _radius;
  const float _filterWidthInv;
};
```

```
class BoxBlurKernel : public Kernel2<eComponentWise,
                                     AccessSpec<Ranged1DAccess, eRead, eComponentWise, ClampedEdge>,
                                     AccessSpec<TapAccess, eWrite, eComponentWise> >
                    , public Rolling<float>
{
  void initialiseRollingData(float &rollingData,
                             const IterationPosition &pos)
  { rollingData = 0; }

  template<class SRC, class DST>
  void rollingRunup(SRC &src, DSTACCESS &dst, float &rollingData,
                    int runupPosition, const IterationPosition &pos)
  { rollingData += src(runupPosition, pos.component()); }

  template <class SRC, class DST>
  void kernel(SRC &src, DST &dst, float &rollingData,
              const IterationPosition &pos)
  {
    float value = rollingData * _filterWidthInv;
    *dst = DSTACCESS::clamp(value);
    rollingData += float(src(_radius+1, pos.component()) - float(src(-_radius, pos.component()));
  }

protected :
  const int _radius;
  const float _filterWidthInv;
};
```

```
class BoxBlurKernel : public Kernel2<eComponentWise,
                                     AccessSpec<Ranged1DAccess, eRead, eComponentWise, ClampedEdge>,
                                     AccessSpec<TapAccess, eWrite, eComponentWise> >
                    , public Rolling<float>
{
  void initialiseRollingData(float &rollingData,
                             const IterationPosition &pos)
  { rollingData = 0; }

  template<class SRC, class DST>
  void rollingRunup(SRC &src, DSTACCESS &dst, float &rollingData,
                    int runupPosition, const IterationPosition &pos)
  { rollingData += src(runupPosition, pos.component()); }

  template <class SRC, class DST>
  void kernel(SRC &src, DST &dst, float &rollingData,
              const IterationPosition &pos)
  {
    float value = rollingData * _filterWidthInv;
    *dst = DSTACCESS::clamp(value);
    rollingData += float(src(_radius+1, pos.component()) - float(src(-_radius, pos.component()));
  }

protected :
  const int _radius;
  const float _filterWidthInv;
};
```

```
class BoxBlurKernel : public Kernel2<eComponentWise,
                                     AccessSpec<Ranged1DAccess, eRead, eComponentWise, ClampedEdge>,
                                     AccessSpec<TapAccess, eWrite, eComponentWise> >
                    , public Rolling<float>
{
  void initialiseRollingData(float &rollingData,
                             const IterationPosition &pos)
  { rollingData = 0; }

  template<class SRC, class DST>
  void rollingRunup(SRC &src, DSTACCESS &dst, float &rollingData,
                    int runupPosition, const IterationPosition &pos)
  { rollingData += src(runupPosition, pos.component()); }

  template <class SRC, class DST>
  void kernel(SRC &src, DST &dst, float &rollingData,
              const IterationPosition &pos)
  {
    float value = rollingData * _filterWidthInv;
    *dst = DSTACCESS::clamp(value);
    rollingData += float(src(_radius+1, pos.component()) - float(src(-_radius, pos.component()));
  }

protected :
  const int _radius;
  const float _filterWidthInv;
};
```

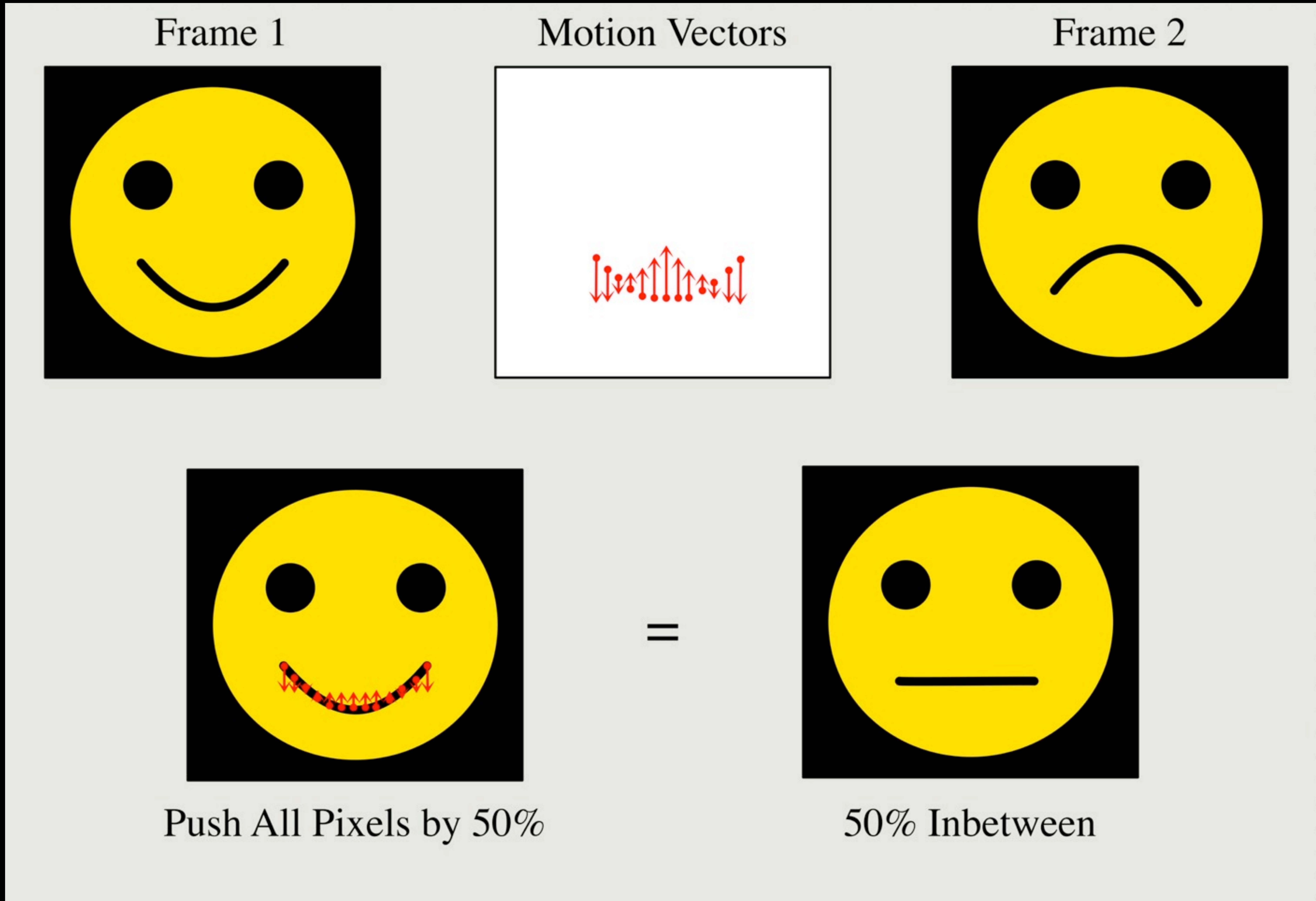
```
class BoxBlurKernel : public Kernel2<eComponentWise,
                                      AccessSpec<Ranged1DAccess, eRead, eComponentWise, ClampedEdge>,
                                      AccessSpec<TapAccess, eWrite, eComponentWise> >
                    , public Rolling<float>
{
  void initialiseRollingData(float &rollingData,
                             const IterationPosition &pos)
  { rollingData = 0; }

  template<class SRC, class DST>
  void rollingRunup(SRC &src, DSTACCESS &dst, float &rollingData,
                    int runupPosition, const IterationPosition &pos)
  { rollingData += src(runupPosition, pos.component()); }

  template <class SRC, class DST>
  void kernel(SRC &src, DST &dst, float &rollingData,
              const IterationPosition &pos)
  {
    float value = rollingData * _filterWidthInv;
    *dst = DSTACCESS::clamp(value);
    rollingData += float(src(_radius+1, pos.component()) - float(src(-_radius, pos.component()));
  }

protected :
  const int _radius;
  const float _filterWidthInv;
};
```

```
class BoxBlurKernel : public Kernel2<eComponentWise,
                                     AccessSpec<Ranged1DAccess, eRead, eComponentWise, ClampedEdge>,
                                     AccessSpec<TapAccess, eWrite, eComponentWise> >
                    , public Rolling<float>
{
  void initialiseRollingData(float &rollingData,
                             const IterationPosition &pos)
  { rollingData = 0; }

  template<class SRC, class DST>
  void rollingRunup(SRC &src, DSTACCESS &dst, float &rollingData,
                    int runupPosition, const IterationPosition &pos)
  { rollingData += src(runupPosition, pos.component()); }

  template <class SRC, class DST>
  void kernel(SRC &src, DST &dst, float &rollingData,
              const IterationPosition &pos)
  {
    float value = rollingData * _filterWidthInv;
    *dst = DSTACCESS::clamp(value);
    rollingData += float(src(_radius+1, pos.component()) - float(src(-_radius, pos.component()));
  }

protected :
  const int _radius;
  const float _filterWidthInv;
};
```

```
class BoxBlurKernel : public Kernel2<eComponentWise,
                                     AccessSpec<Ranged1DAccess, eRead, eComponentWise, ClampedEdge>,
                                     AccessSpec<TapAccess, eWrite, eComponentWise> >
                    , public Rolling<float>
{
  void initialiseRollingData(float &rollingData,
                             const IterationPosition &pos)
  { rollingData = 0; }

  template<class SRC, class DST>
  void rollingRunup(SRC &src, DSTACCESS &dst, float &rollingData,
                    int runupPosition, const IterationPosition &pos)
  { rollingData += src(runupPosition, pos.component()); }

  template <class SRC, class DST>
  void kernel(SRC &src, DST &dst, float &rollingData,
              const IterationPosition &pos)
  {
    float value = rollingData * _filterWidthInv;
    *dst = DSTACCESS::clamp(value);
    rollingData += float(src(_radius+1, pos.component()) - float(src(-_radius, pos.component()));
  }

protected :
  const int _radius;
  const float _filterWidthInv;
};
```

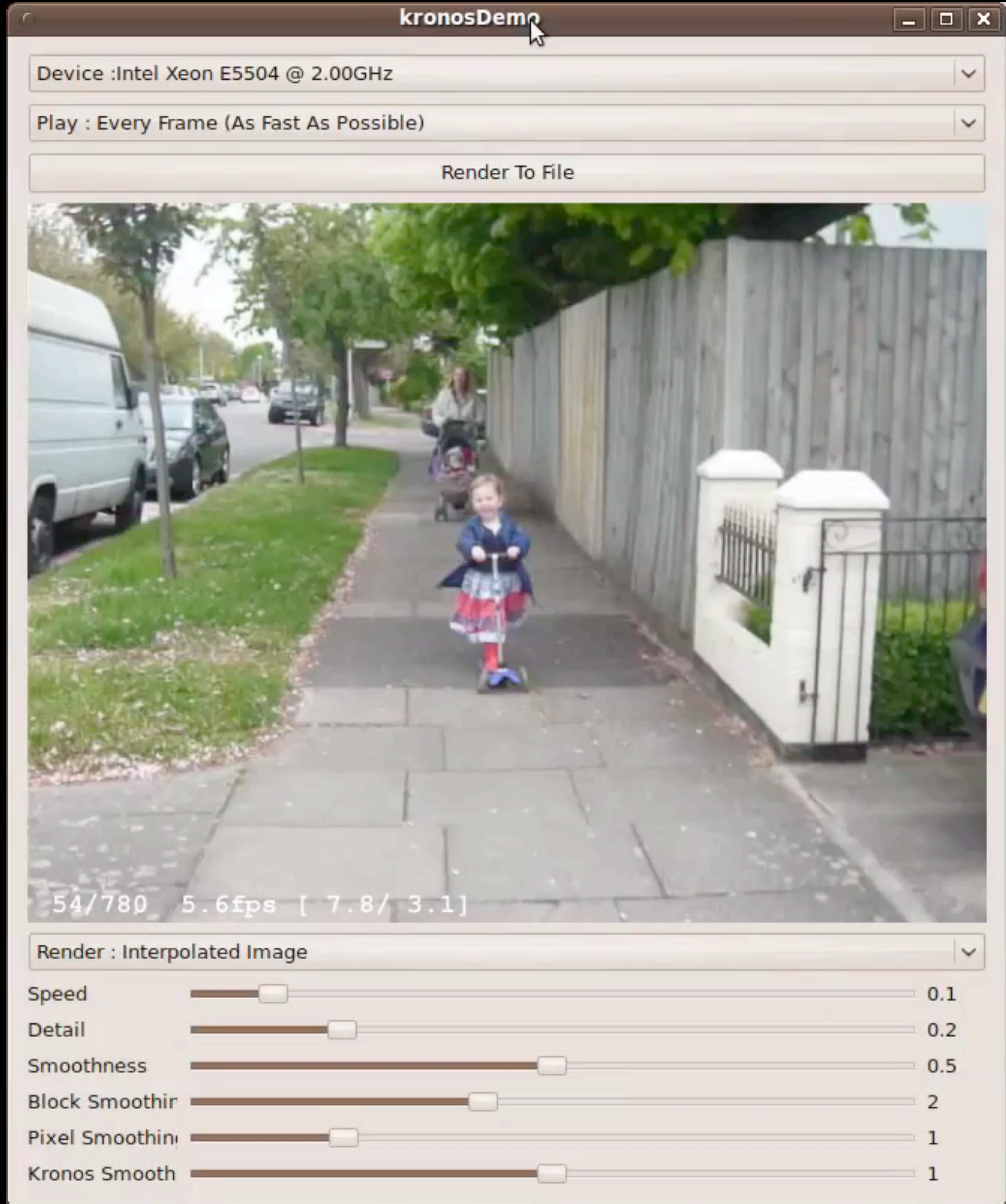
About Motion Estimation

- One of our core bits of IP
- Effectively per pixel tracking between images
  – you get a vector per pixel indicating inter-image displacement
- Allows you to solve lots of problems in 2D VFX
  – including 'in-betweening' to retime footage
- A large set of complex algorithms
- Impossible to do with GP-GPU techniques
- Implemented on RIP framework


Frame 1
Motion Vectors
Frame 2
=
Push All Pixels by 50%
50% Inbetween

Implementing Motion Estimation In RIP

- Complex set of algorithms that needed 33 RIP kernels to implement
- On a 4 Core Xeon E5504@2GHz,10:1 retime of SD footage achieve 5 fps
  – no SSE path yet, will go faster when we do
- On a Quadro 5000 we achieve 200fps!
  – including host to GPU device transfer
- Thorny issue, the pictures are different between CPU and GPU
  – because of a 'push' algorithm
  – which is a problem anyway on a multicore CPU
  – could fix via atomics, but at a large compute cost


kronosDemo
Device :Intel Xeon E5504 @ 2.00GHz
Play : Every Frame (As Fast As Possible)
Render To File
54/780 5.6fps [7.8/ 3.1]
Render : Interpolated Image
Speed 0.1
Detail 0.2
Smoothness 0.5
Block Smoothir 2
Pixel Smoothin 1
Kronos Smooth 1

Moving Bottlenecks

- In practice computation bottlenecks simply get moved
  – our retimer can compute SD at around 200fps on a Quadro 5000
  – as a plugin to After Effects, it peaks at around 15fps
- Amdhal's law has kicked in
  – for VFX, file i/o is a big part of the serial computation
- So do as much computation as possible while in memory
  – but CPU apps attempt to do that

Near Term Future Work

- Code translate all kernels type and accessors
  - all transformations and large reductions
- Complete implementation of a processing graph
  - to manage tiled image rendering for large data sets
  - as a harness for kernel fusion
- Complete SSE support on CPU

Future Research

- Run time translation of kernels
  - requires run time compilers for CPU and GPU
- Inter-kernel optimisations
  - data dependencies allows for 'simple' low level kernel fusion
    - which reduces memory traffic = higher performance
    - hard cases as well (eg: chained set of ranged access kernels)
      - e.g. loop fusion of ranged accessors via array contraction
  - Proof of concept via collaboration with Imperial College London

What We’ve Learnt

- We were much more ambitious that we thought
- Clang/LLVM rocks (basis of our parsing)
- You still need to know about the hardware
- Breaking CPU/GPU agreement is occasionally necessary
  - provided you know why and where you are doing it
- It is sometimes necessary to write separate GPU and CPU paths
- Run time compilation is essential for where we want to take this
- OpenCL on its own isn’t what you need