Imperial College London

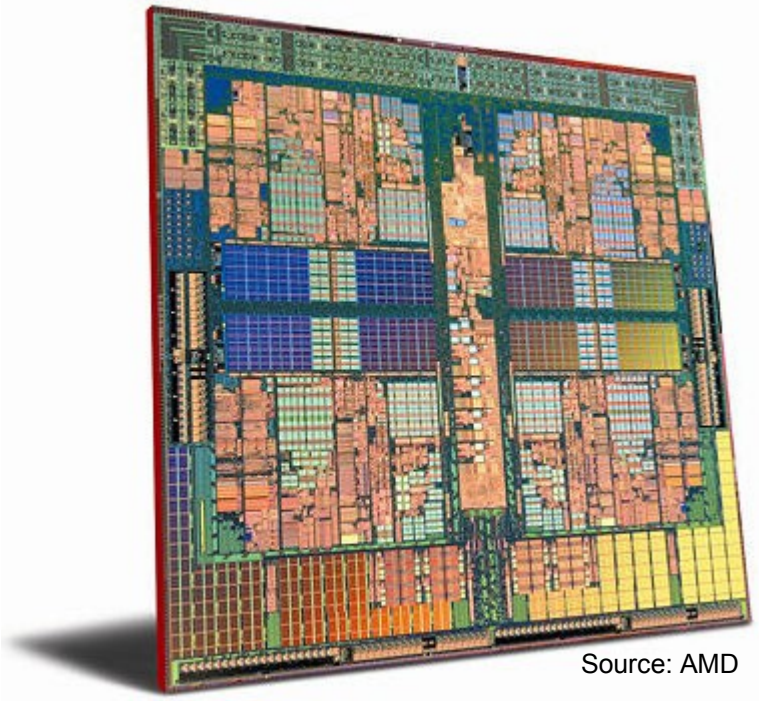
Deriving Efficient Data Movement From Decoupled Access/Execute Specifications

Lee W. Howes, Anton Lokhmotov,
Alastair F. Donaldson and Paul H. J. Kelly
Imperial College London and Codeplay Software

January 2009

Multi-core architectures

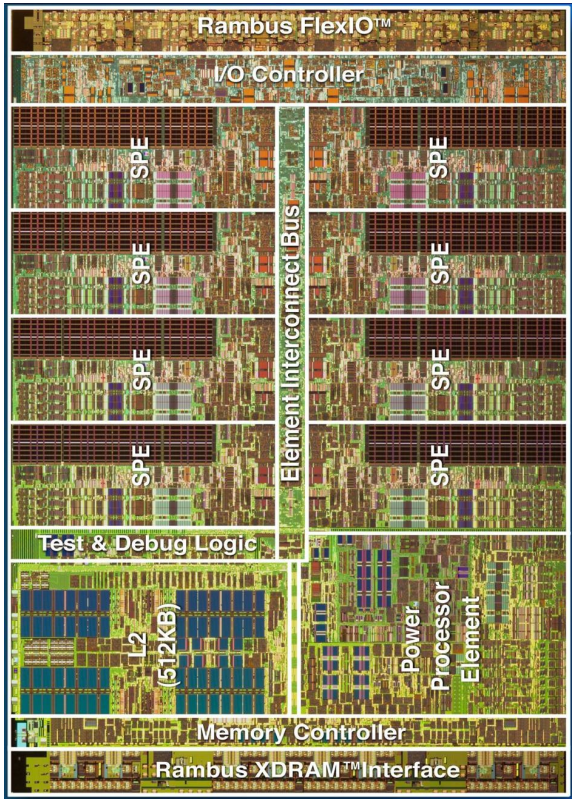
- Require parallel programming
- Must divide computation
- Must communicate data
- High-throughput computation
  - Efficient use of memory bandwidth essential



Source: AMD

Cell's hardware solution

- Target the memory wall:
  - Distributed local memories: 256kB each
  - Separate data movement from computation using DMA engines
- Bulk transfers increase efficiency
- Increased programming challenge:
  - Must write data movement code
  - Must deal with alignment constraints
- Premature optimisation
  - Platform independence is lost



Source: IBM

Mainstream programming models

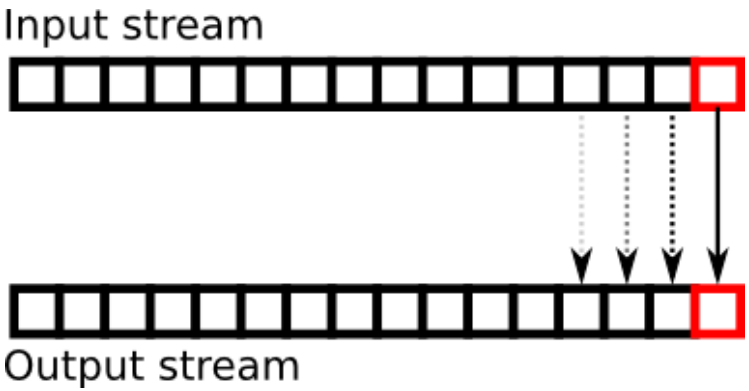
- No explicit support for separation of computation from data access
- Freely mix computation and data movement
- Complexity of compiler analysis => Difficult to extract separation
- Orthogonal issues:
  - extracting parallelism
  - creating data movement code

The proposal

- Allow the programmer to express explicitly:
  - Separation between data communication and computation
  - Parallelism of the computation

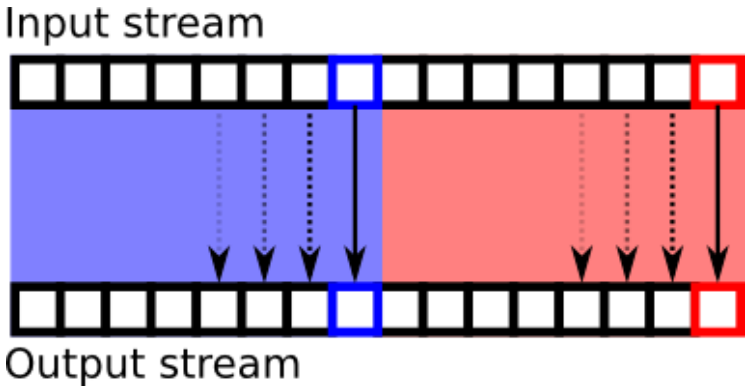
Streams

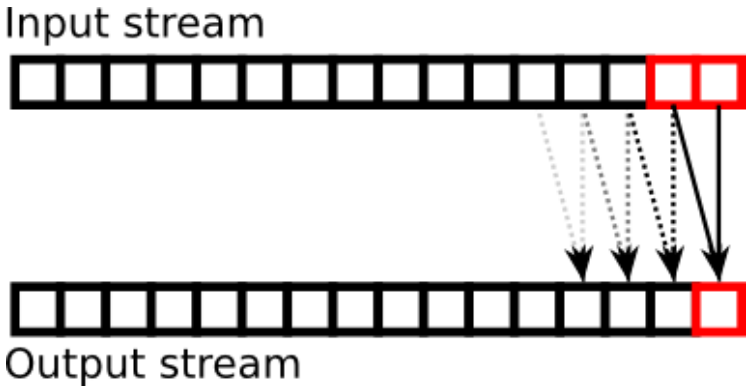
- Approaches the separation ideal
- Simple kernel applied to each element of a data set
- Each element of stream typically independent of others
  - No feedback as a parallel processing model
  - Dependencies only on input and output elements



Parallelism in stream programming

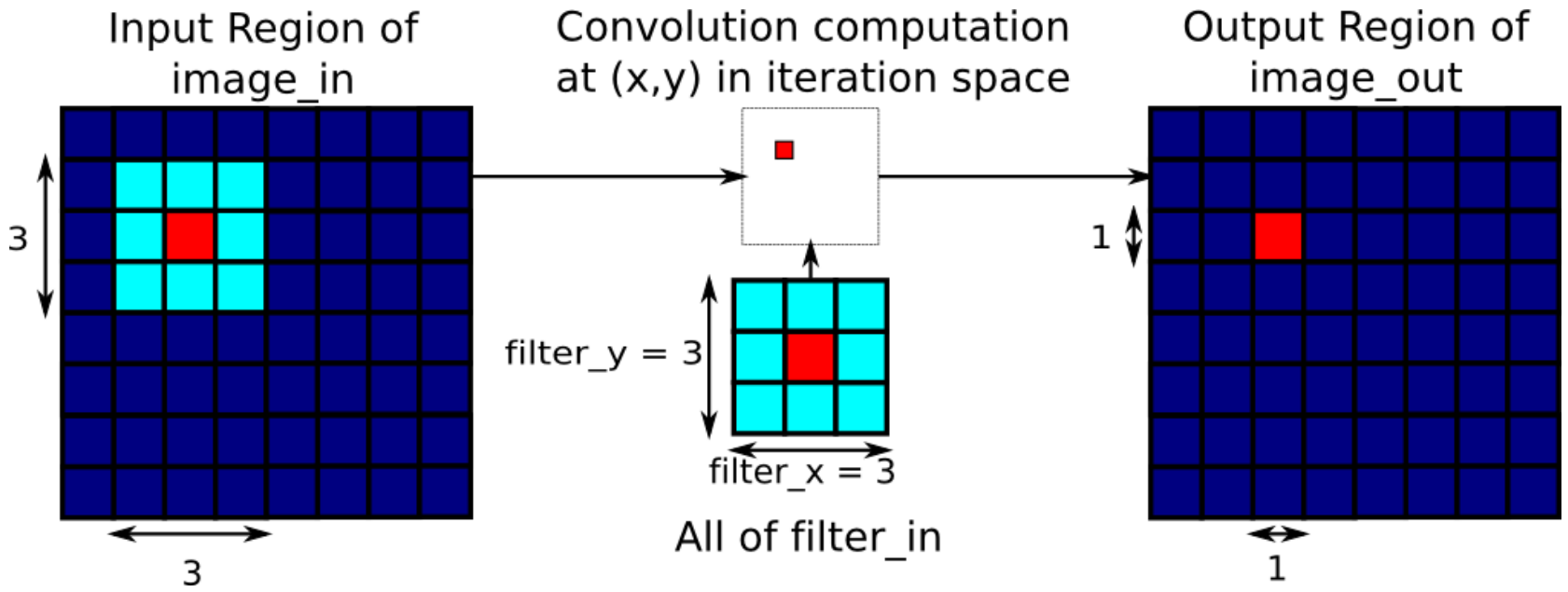
- Independence of executions => simple inference of parallelism
- Sliding windows of elements on inputs
  - access multiple elements
  - parallelism still predictable
- AMD, NVIDIA use a stream model for parallel hardware





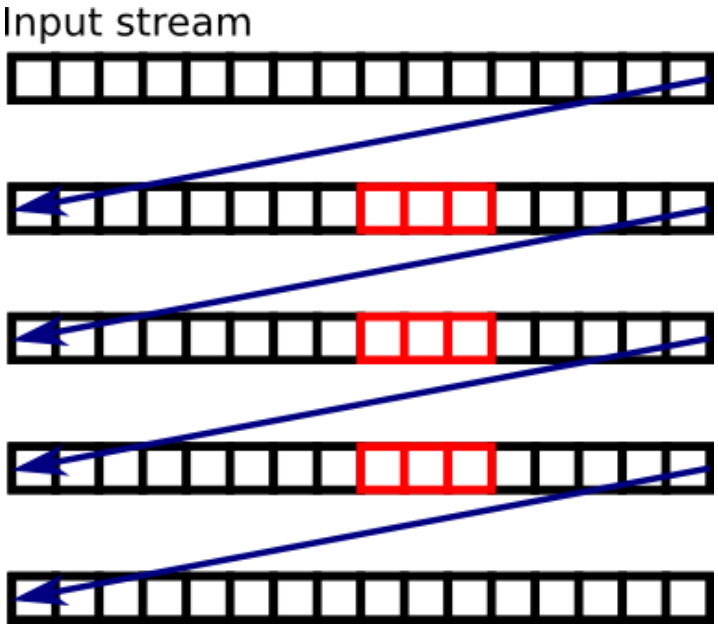
Streams? A 2D convolution filter

- Reads region of input
- Processes region
- Writes single point in the output



Representing convolution as 1D streams

- One option: flatten 2D dataset
  - Requires multiple sliding windows or long FIFO structures
- Mapping 2D structures to 1D streams is untidy



Representing convolution as 2D streams

- Stanford's Brook language uses stencils on 2D shaped streams

```
floats x;
floats2 y;
streamShape(x,2,32,32);

streamStencil(y, x, STREAM_STENCIL_CLAMP, 2, 1, -1, 1, -1);
kernel void neighborAvg(floats2 a, out floats b) {
    b = 0.25*(a[0][1]+a[2][1]+a[1][0]+a[1][2]);
}
```

Representing convolution as 2D streams

- Stencil stream passed to kernel
- Treated as if it is a small set of accessible elements
- Limited addressing capabilities

```
floats x;
floats2 y;
streamShape(x,2,32,32);

streamStencil(y, x, STREAM_STENCIL_CLAMP, 2, 1, -1, 1, -1);
kernel void neighborAvg(floats2 a, out floats b) {
    b = 0.25*(a[0][1]+a[2][1]+a[1][0]+a[1][2]);
}
```

Generalising streams

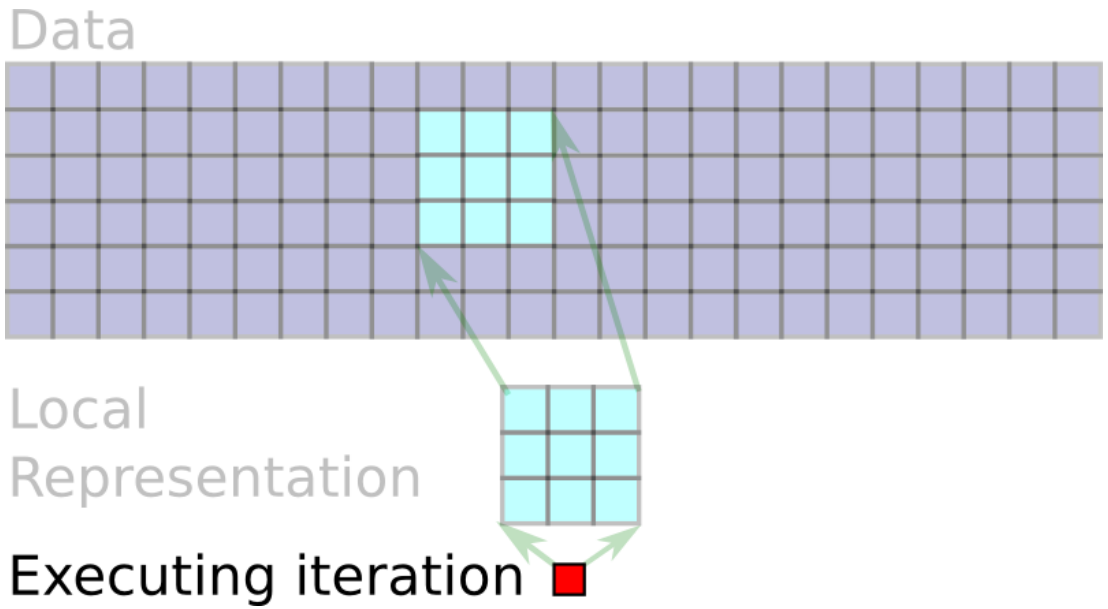
- View streams as:
  - A kernel, executed separately on each data element
  - A simple mapping of that kernel onto the data – elementwise or moving windowed
- This is a simplistic separation of access from execution, hence the Decoupled Acess/Execute (***Æcute***) model

Æcute as a generalisation of streams

- Take a similar kernel-per-element declarative programming model
- View in terms of an iteration space that is independent of the data sets
- With a separate, flexible mapping to the data
- Mapping allows clean descriptions of complicated data access patterns
- Simpler kernel implementations with localised data sets

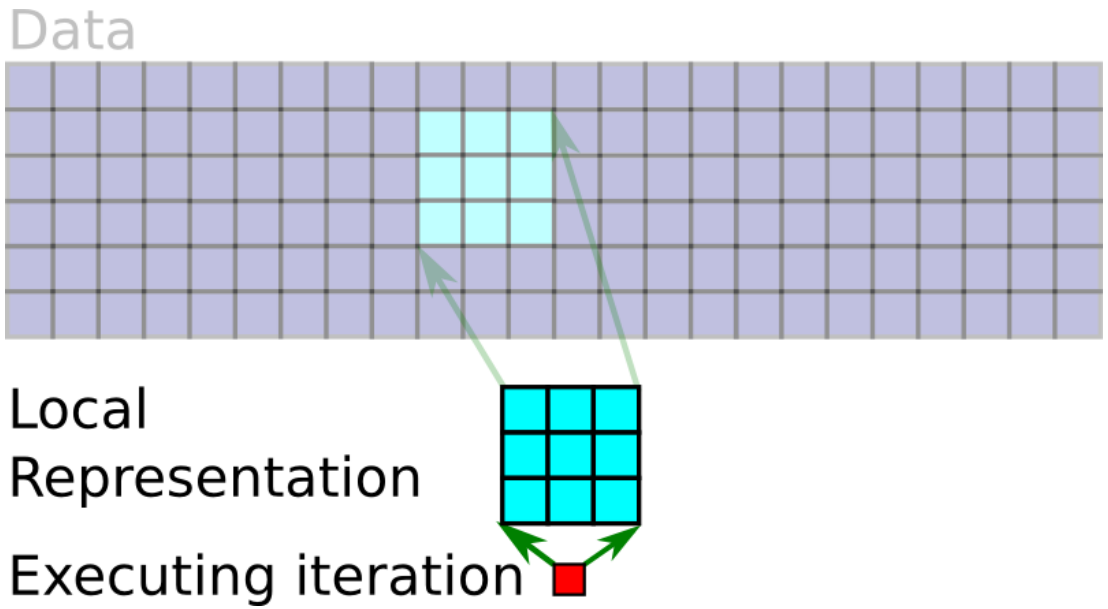
Execute

- Define an iteration space (e.g. as polyhedral constraints)
- Execute a computation kernel for each point in the iteration space



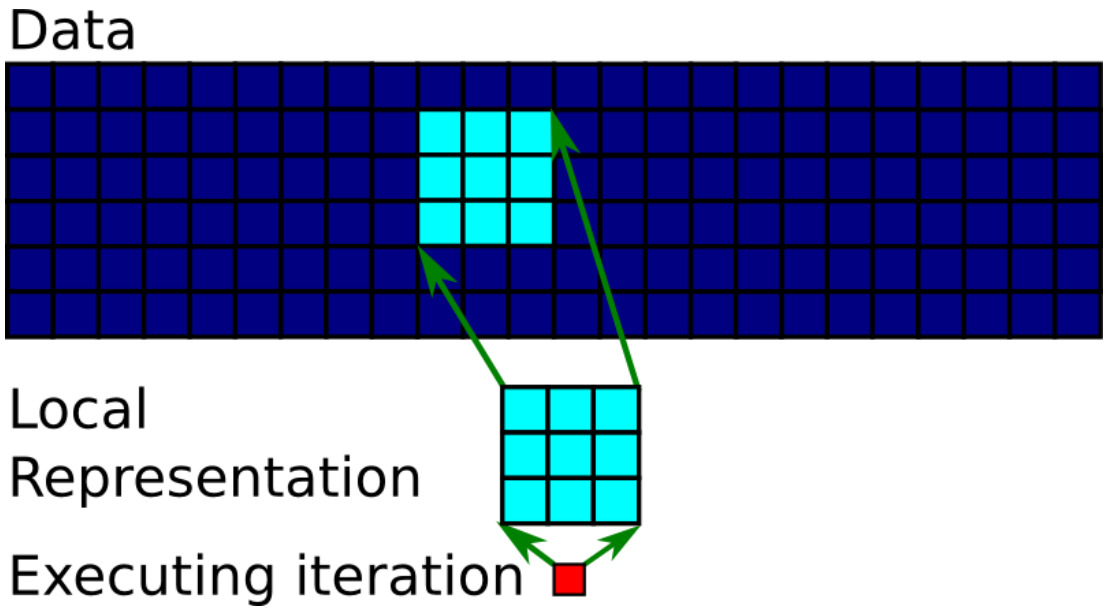
Data access

- On each iteration, the kernel accesses a set of data elements
- Accessed elements treated as local to the iteration
- Eases programming of the kernel



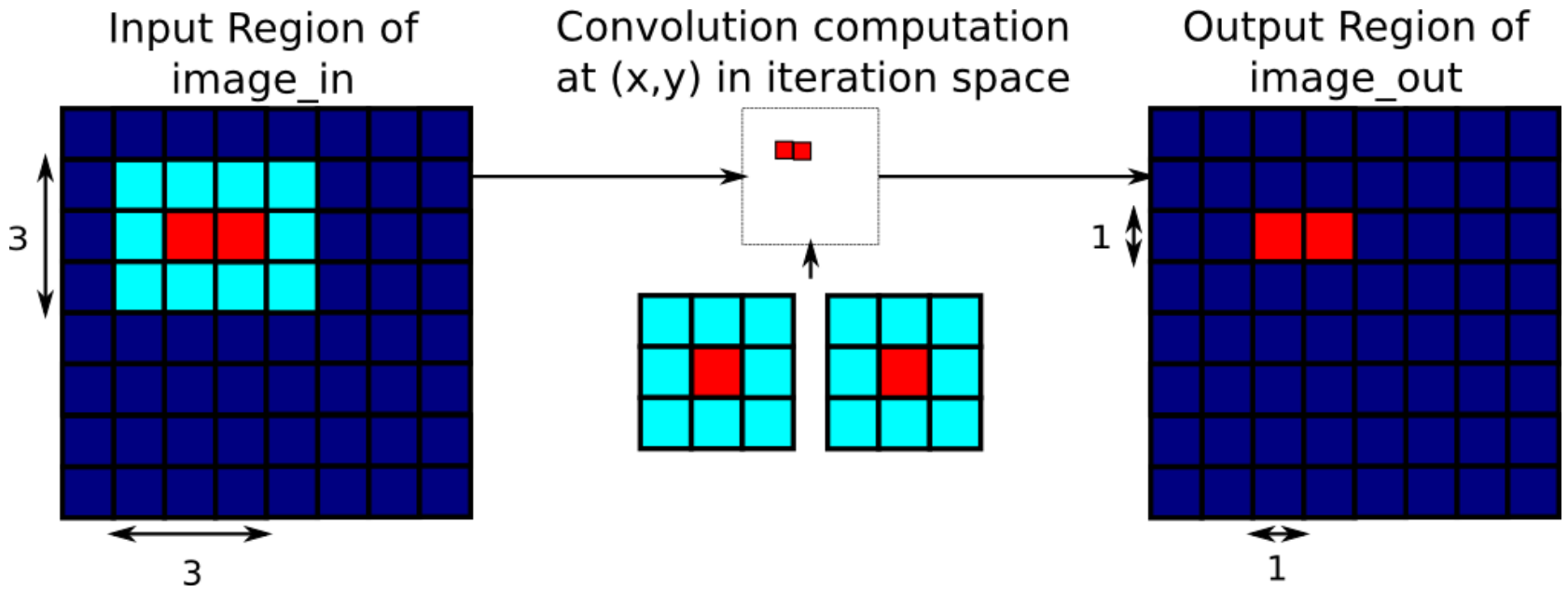
Decoupled access/execute

- Decouple **access** to remote memory from local **execution**
- Separate mapping of local store to global data



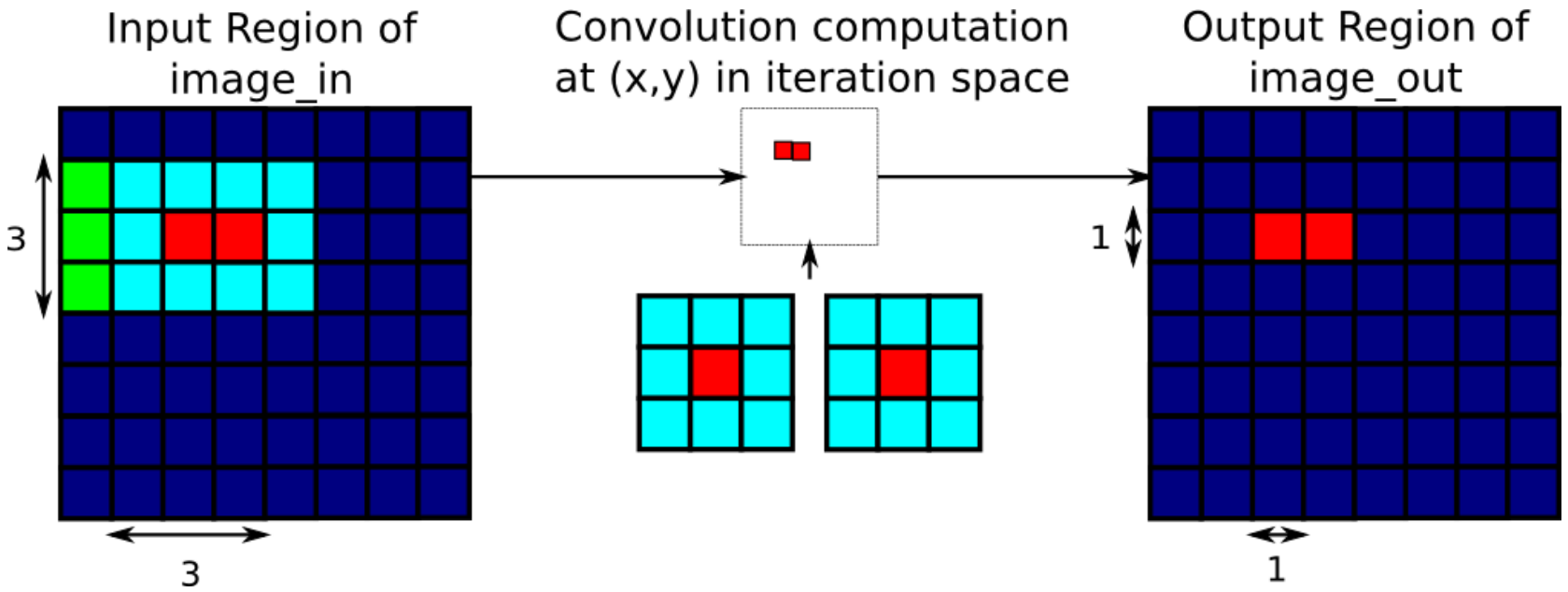
Multiple iterations

- Decouple access and execute for multiple iterations for efficiency
- Manually supporting this flexibility can be challenging



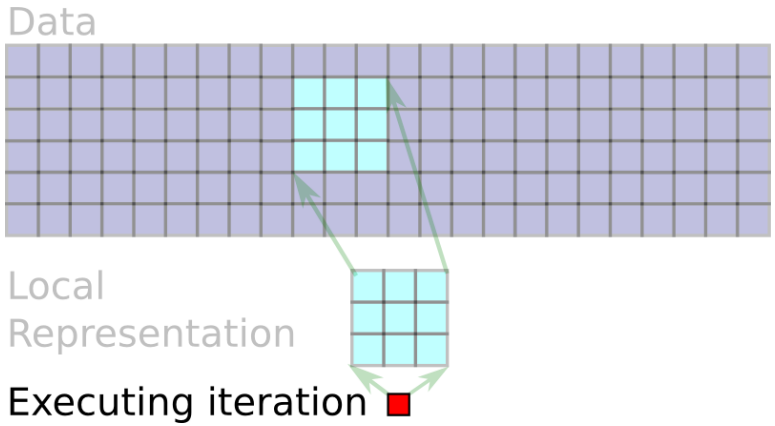
Add in alignment issues

- DMAs must be adapted to correct for alignment
- Data can often be read with alignment tweaks to fix performance



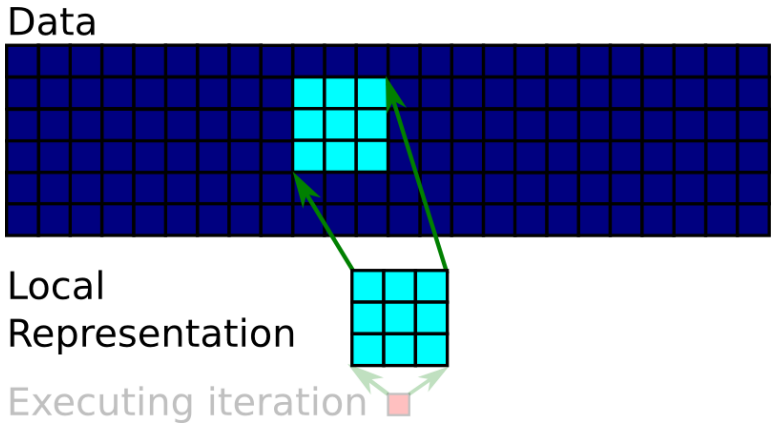
In code: The iterator

```
Neighbourhood2D_Read inputPointSet(iterationSpace, input, K);
Point2D_Write outputPointSet(iterationSpace, output);
...
void kernel( const IterationSpace2D::element_iterator &eit ) {
  // compute mean
  rgb mean( 0.0f, 0.0f, 0.0f );
  for(int w = -K; w <= K; ++w) {
    for(int z = -K; z <= K; ++z) {
      mean += inputPointSet(eit, w, z); // input[x+w][y+z]
    }
  }
  outputPointSet( eit ) = mean / ((2K+1)(2K+1));
}
```



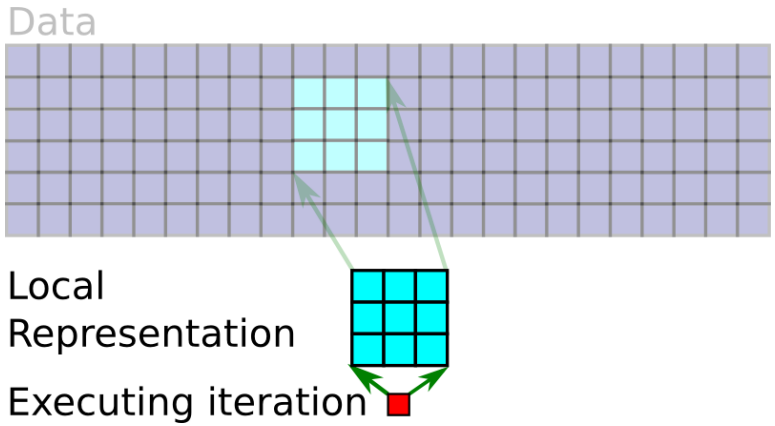
In code: Use of access descriptors

```
Neighbourhood2D_Read inputPointSet(iterationSpace, input, K);
Point2D_Write outputPointSet(iterationSpace, output);
...
void kernel( const IterationSpace2D::element_iterator &eit ) {
  // compute mean
  rgb mean( 0.0f, 0.0f, 0.0f );
  for(int w = -K; w <= K; ++w) {
    for(int z = -K; z <= K; ++z) {
      mean += inputPointSet(eit, w, z); // input[x+w][y+z]
    }
  }
  outputPointSet( eit ) = mean / ((2K+1)(2K+1));
}
```



In code: Computation in the kernel

```
Neighbourhood2D_Read inputPointSet(iterationSpace, input, K);
Point2D_Write outputPointSet(iterationSpace, output);
...
void kernel( const IterationSpace2D::element_iterator &eit ) {
  // compute mean
  rgb mean( 0.0f, 0.0f, 0.0f );
  for(int w = -K; w <= K; ++w) {
    for(int z = -K; z <= K; ++z) {
      mean += inputPointSet(eit, w, z); // input[x+w][y+z]
    }
  }
  outputPointSet( eit ) = mean / ((2K+1)(2K+1));
}
```



Æcute iteration spaces

- Define an n-dimensional iteration space
- Specify sizes for each dimension
  - can be run time defined
- For example:
  - IterationSpace<2> iSpace(0, 0, 10, 10);
- Over which we can iterate using fairly standard syntax:
  - for(IterationSpace<2>::iterator
    it = iSpace.begin().....){...}
- Can treat the iterator loop much as an OpenMP blocked look

Æcute access descriptors

- Define a mapping from an iteration space to an array
- Specify shape and mapping functions
- For example:
  - Region2D<Array<rgb,2>,IterationSpace<2>>
    inputPointSet(iSpace, data, RADIUS);
- Which we can access using an iterator
  - InputPointSet(it,1,0).r = 3;

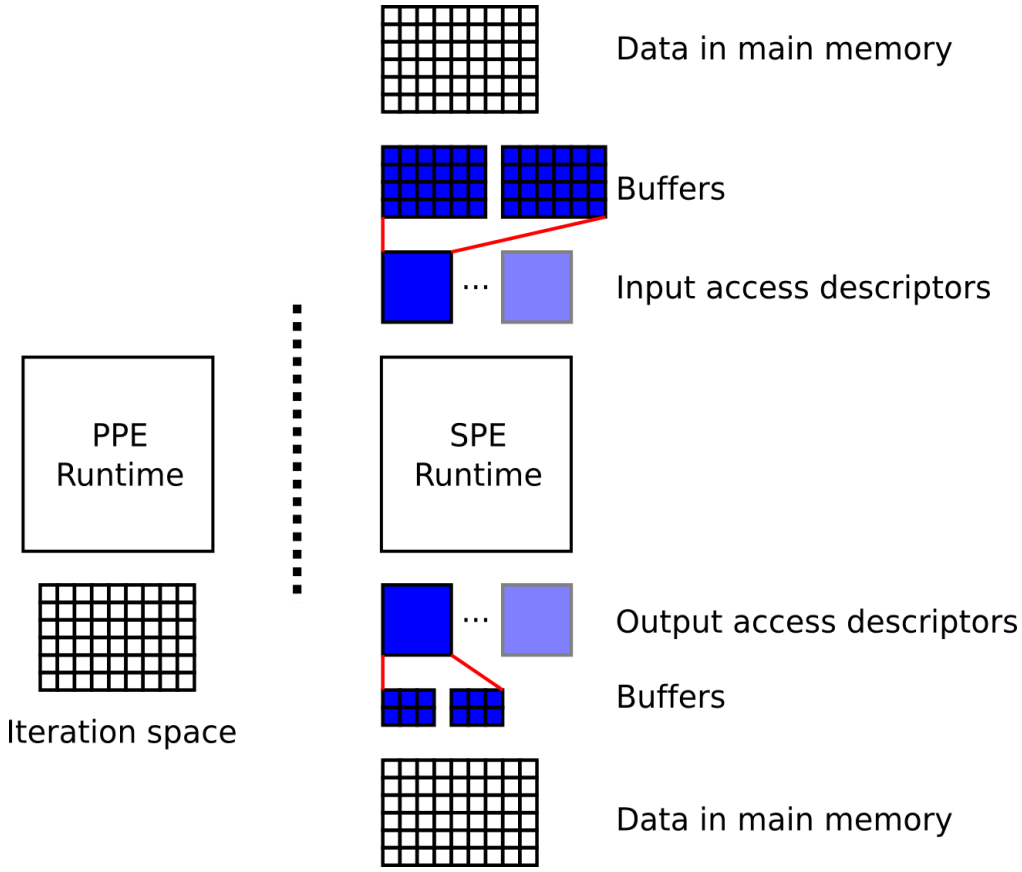
Æcute address modifiers

- Base address of a region combines:
  - iterator address in its iteration space
  - address modifier function
- A modifier, or modifier chain, is applied (optionally) to each access descriptor:
  - `Point2D< Project2D1D< 1, 0 > >`

    `inputPointSet( iSpace, data, RADIUS );`
  - Projects a 2D address into a 1D address to access a 1D dataset

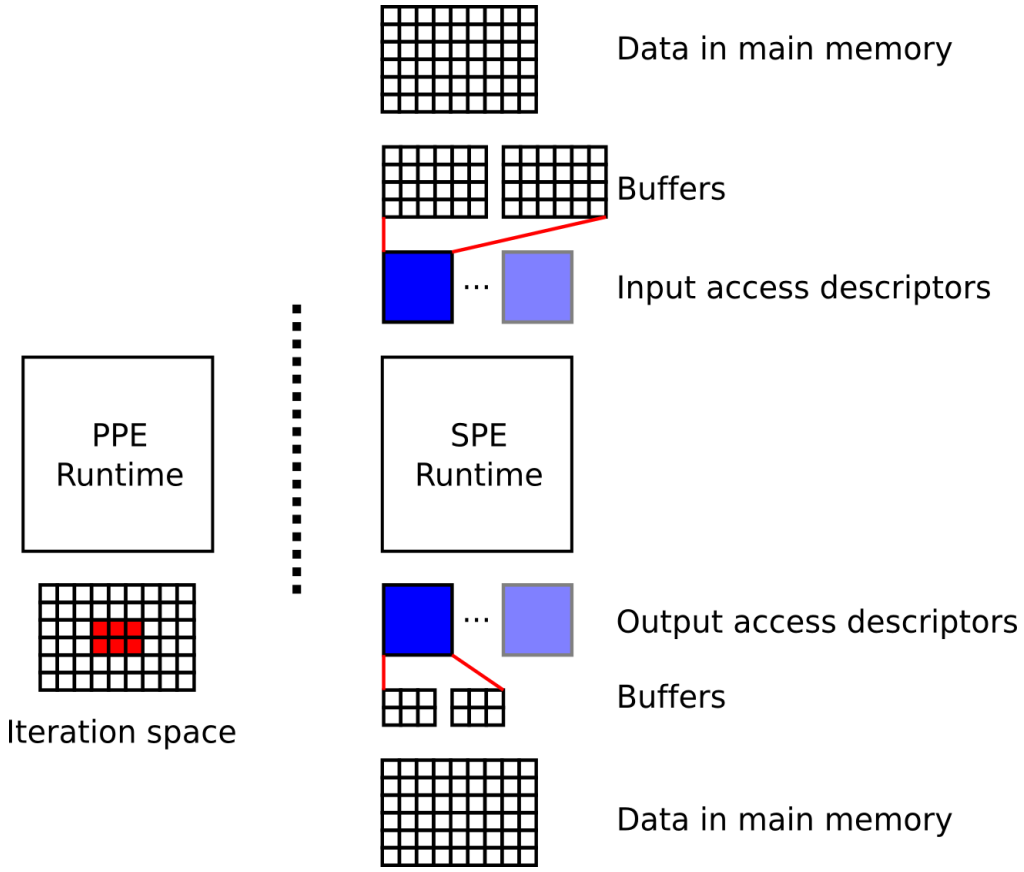
The Æcute framework

- Implementation of the Æcute model for data movement on the STI Cell processor



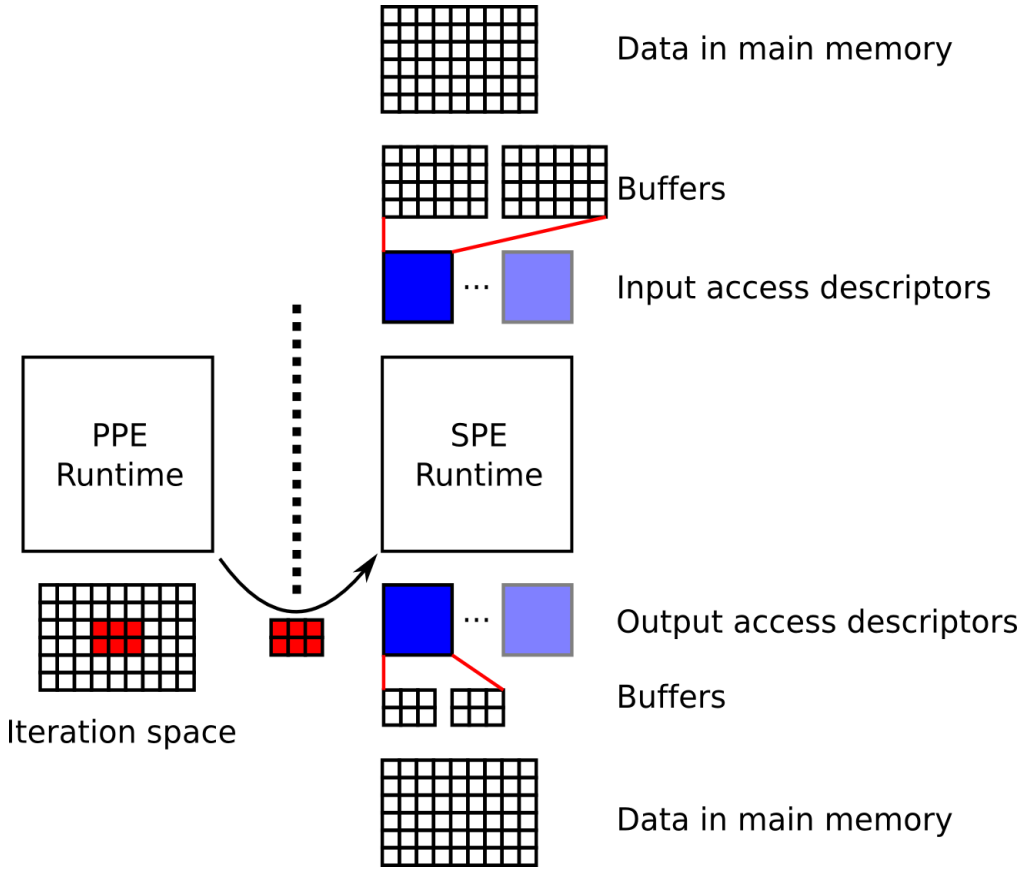
Iterating

- PPE takes a chunk of the iteration space
  - Blocking is configurable



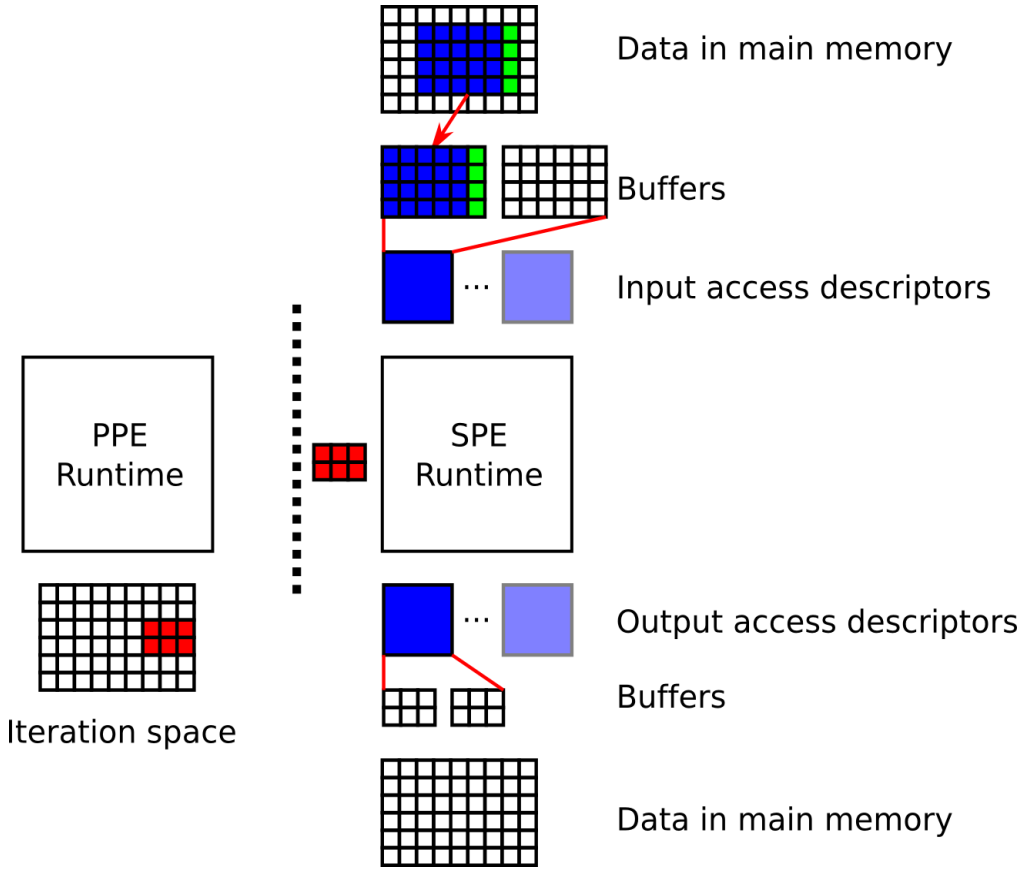
Delegation

- Transmits chunk to appropriate SPE runtime as a message



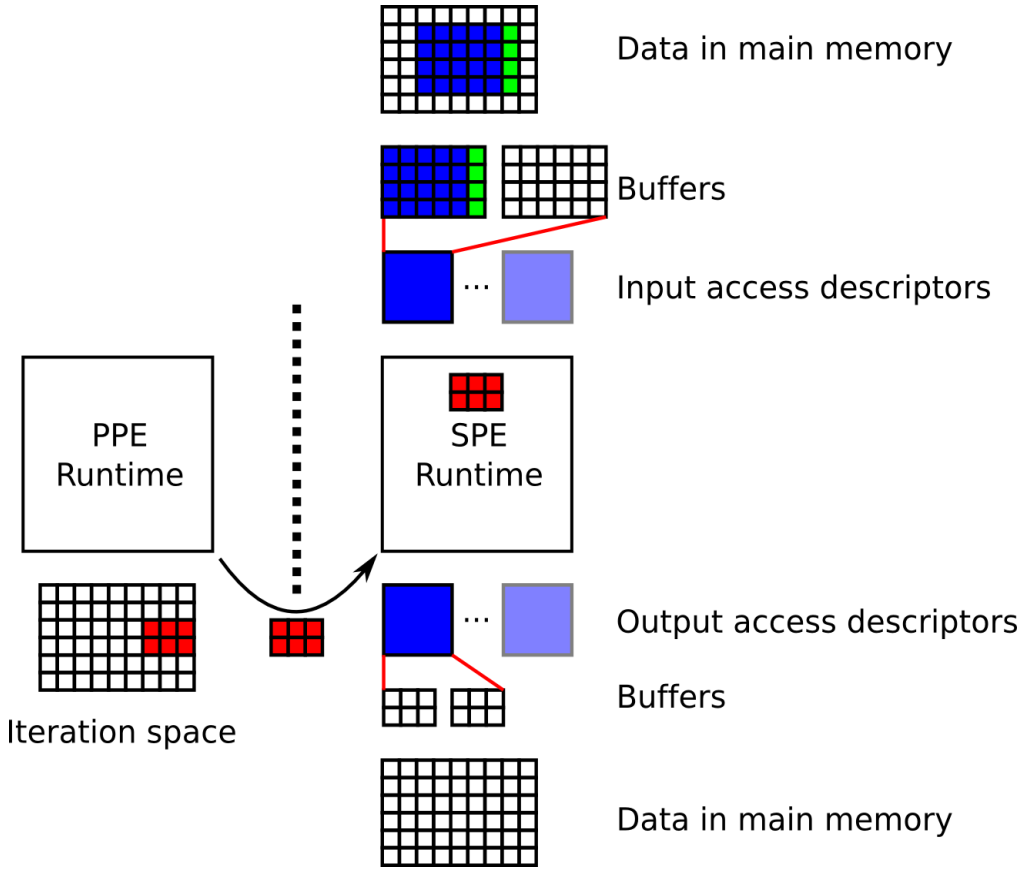
Loading data

- SPE loads appropriate data for the chunk into an internal buffer in each access descriptor object



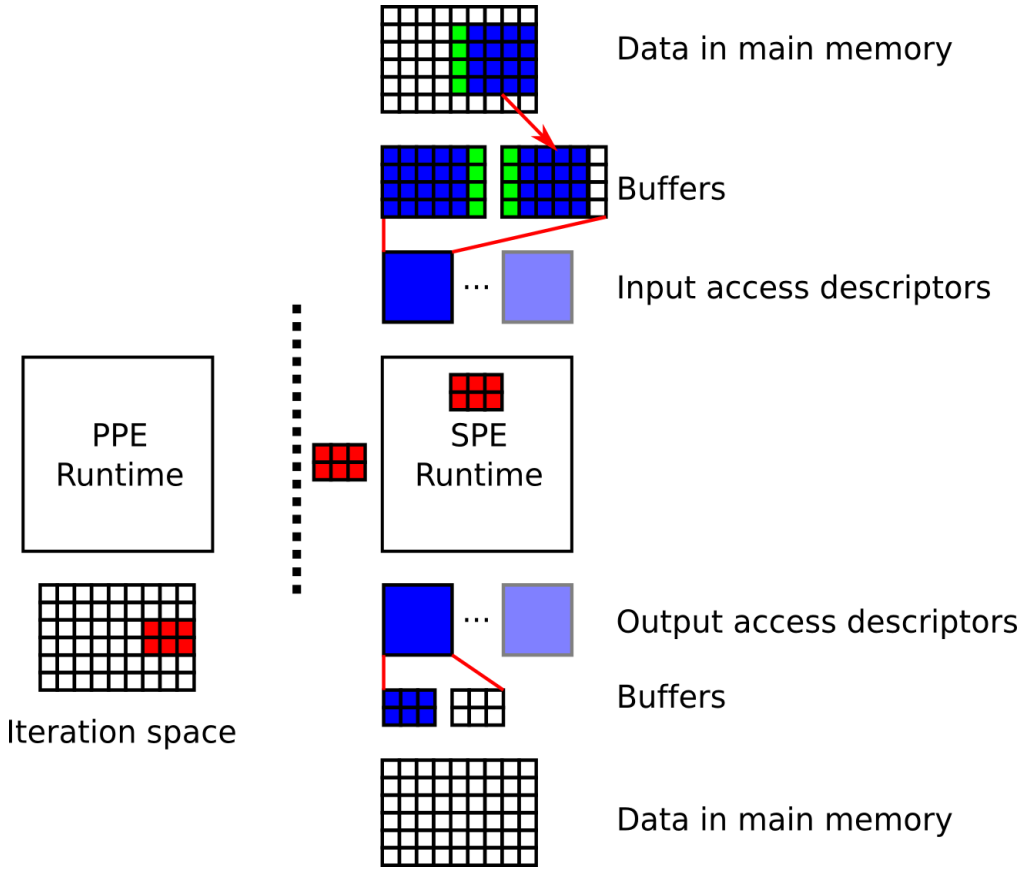
Loading data

- SPE processes one buffer set while receiving the next block to process



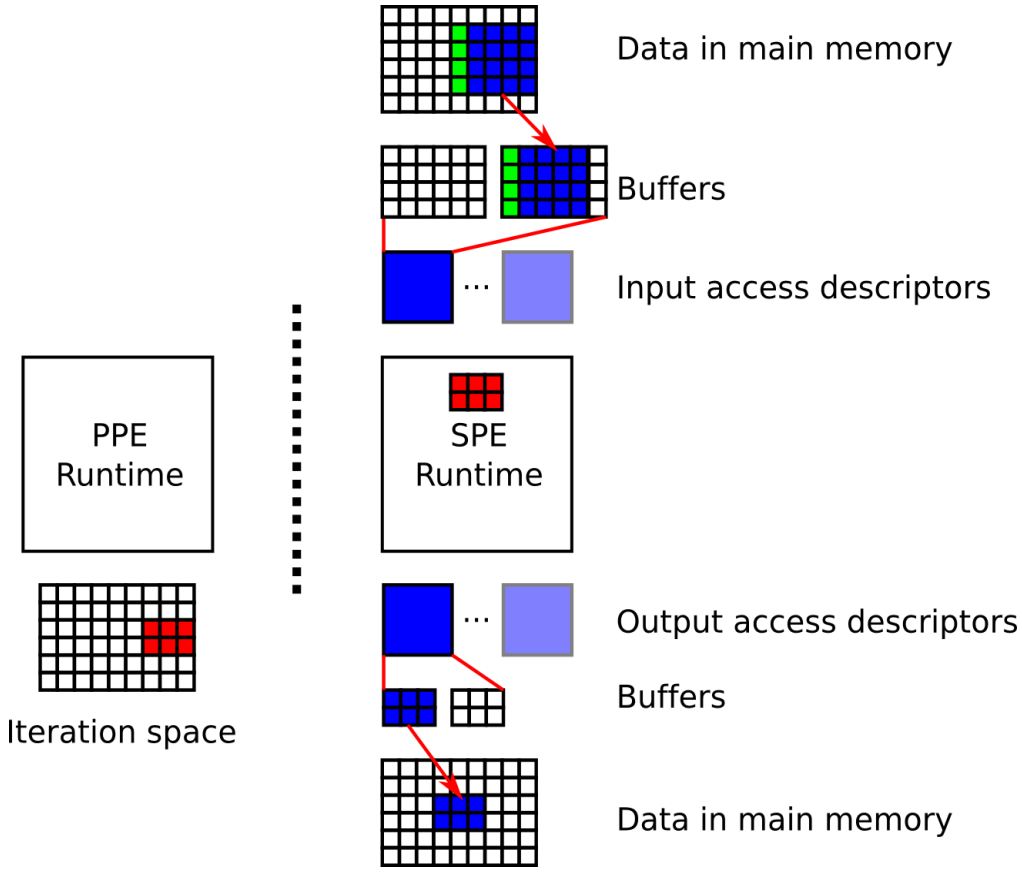
Loading data

- DMA loading next buffers operate in parallel with computation



Loading data

- On completion of a block, input buffers cleared, output DMAs initiated



Advantages

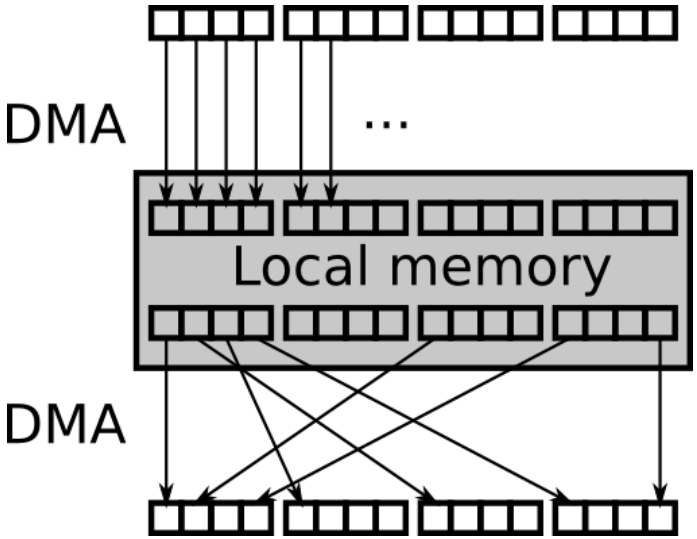
- Separation of buffering maintains simplicity
- Double/triple buffering comes naturally when there are no data dependent loads
- Remove complexity of manual software pipelining
- Complicated addressing schemes not precluded

Non-affine addressing

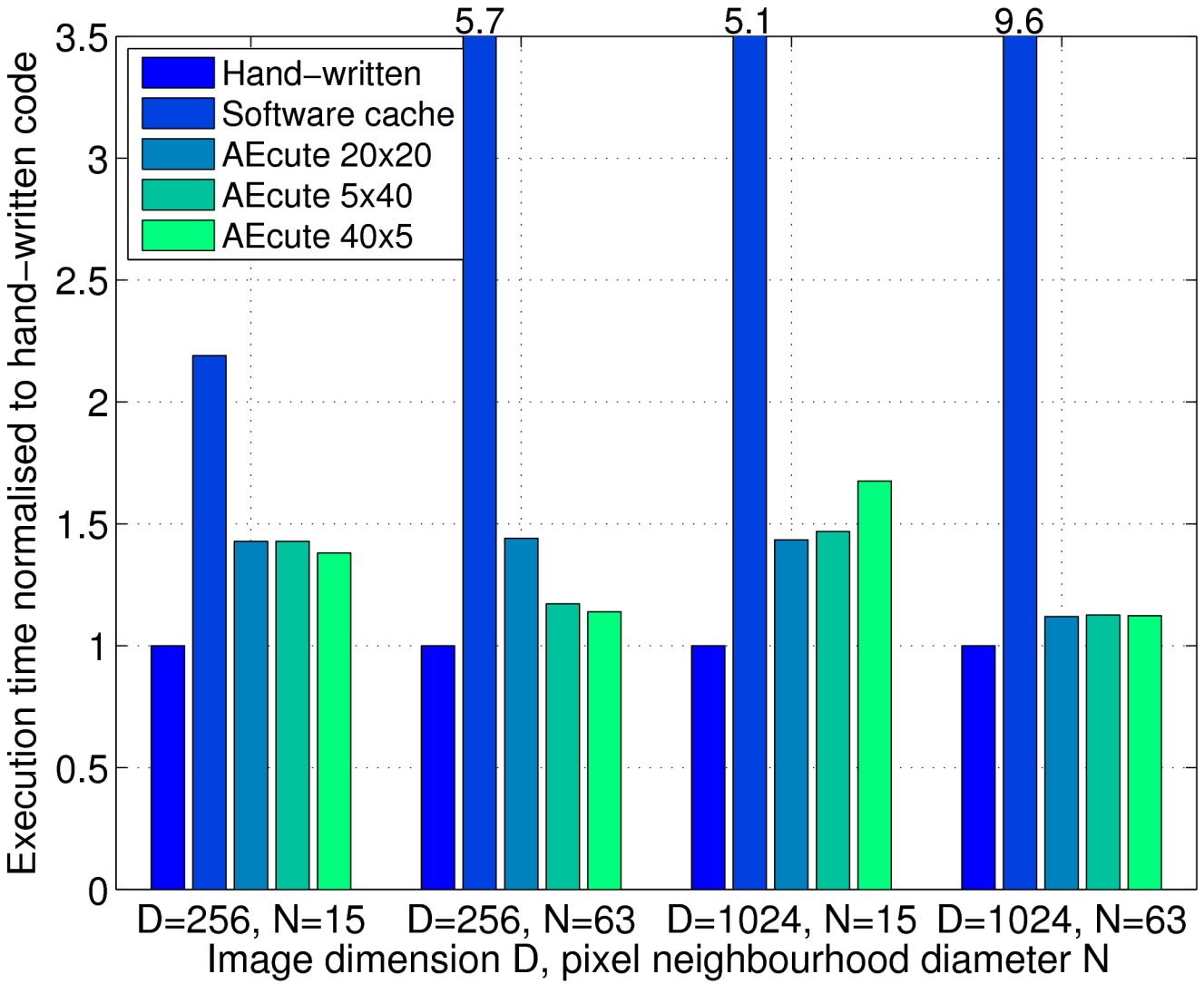
- Simple stencils inadequately flexible
- Partitioning of Iteration space defines parallelism
- Generating complicated addressing schemes is often necessary
  - Addressing can still be performed externally to the computation and automatically pipelined
  - Alignment may need to be on a per-element basis if relationship inference not possibile.

The bit reversal

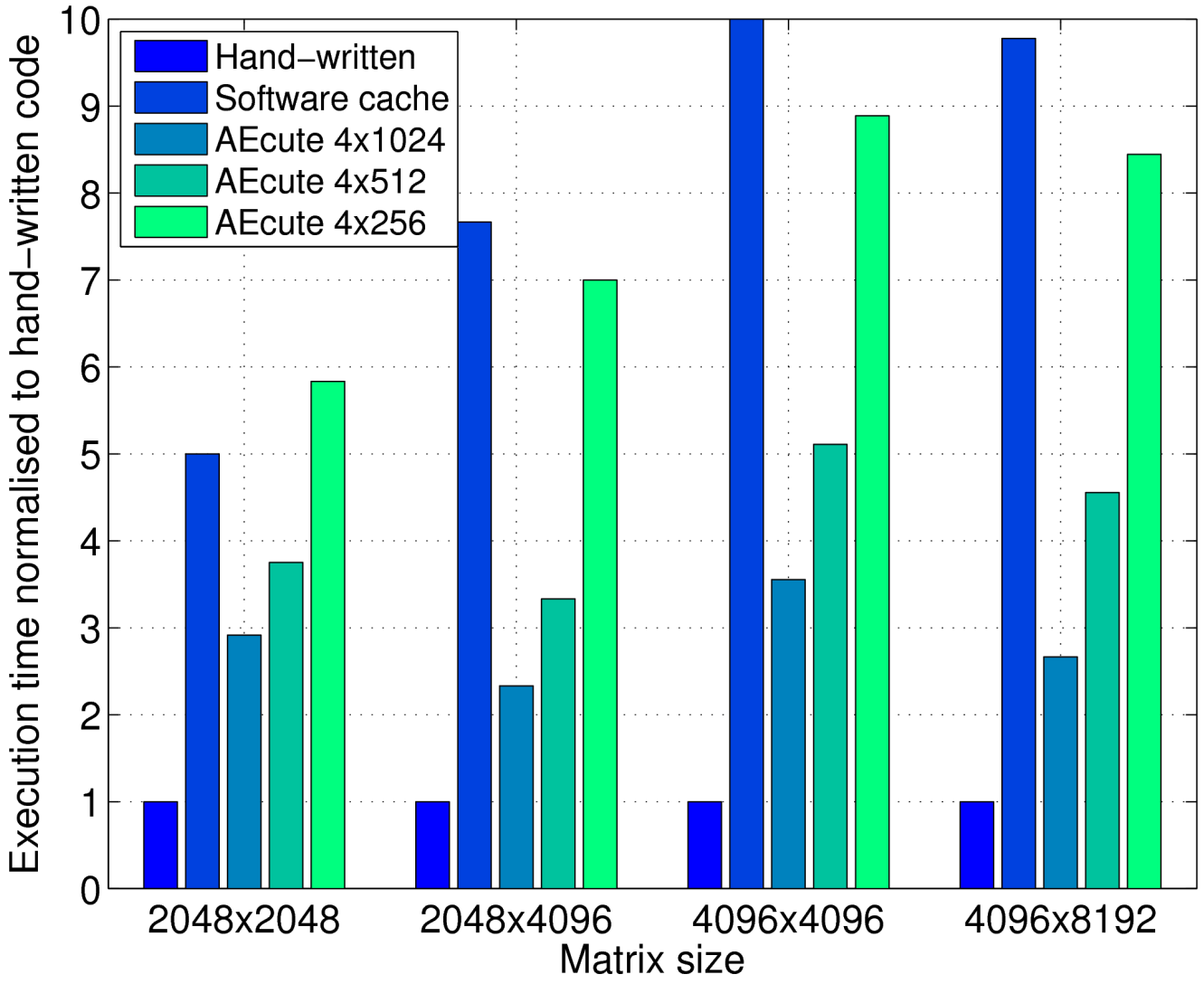
- As used in a radix-2 FFT:
  - Performs a complicated, but predictable, permutation of a data set
  - Input address with bits reversed => output address
- Access descriptors can wrap complicated addressing
  - Generate DMA lists



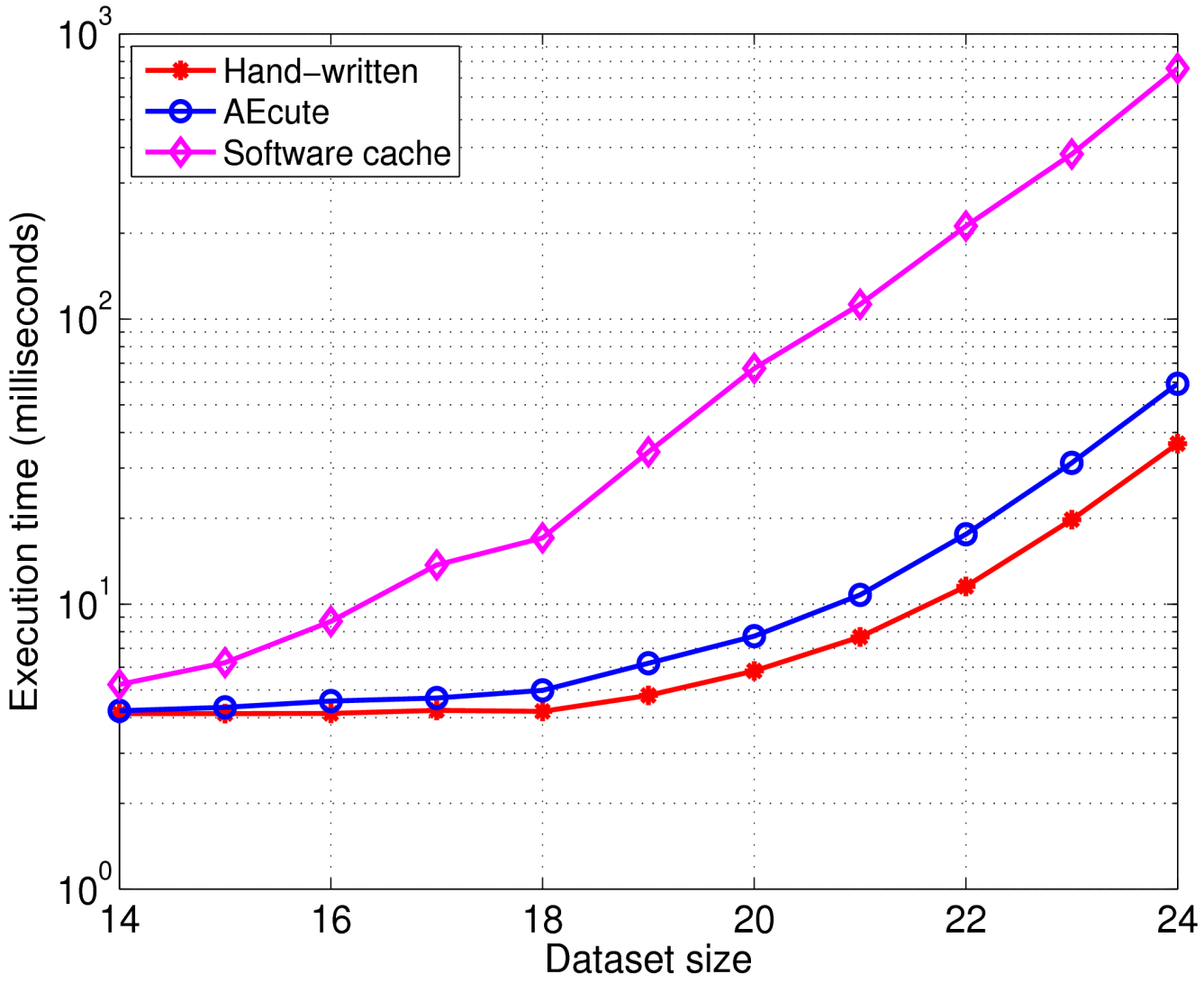
Æcute performance: CTM filter



Æcute performance: Matrix/vector multiply



Æcute performance: Bit reverse



Conclusions

- Programming model that generalises streaming
- Declarative mapping of computation to data
- Separate kernel implementation working on simple data subset
- Further work on:
  - Inference of inter-kernel dependencies
  - Merging of earlier kernel fusion work
  - Targetting different architectures: GPUs
  - Compiler support
  - Integration with the Sieve system
  - Investigating the limits of this kind of specification