

Decoupled Access/Execute Software for Distributed Memory Architectures

Lee Howes¹ Anton Lokhmotov¹ Paul Kelly¹
Alastair Donaldson²

Imperial College
¹**London**



5th HiPEAC Industrial Workshop
4 June 2008

Architectures with software-managed memory

H. Peter Hofstee, Chief Architect of the Cell processor (2005)

“... managing memory locality becomes the main factor determining software performance, and writers of compilers and high-performance software alike spend much of their time reverse-engineering and defeating the sophisticated mechanisms that automatically bring data on to and off the chip. Given the large number of transistors devoted to these mechanisms this is an unsatisfactory situation.”

Architectures with software-managed memory

H. Peter Hofstee, Chief Architect of the Cell processor (2005)

*“... managing memory locality becomes the main factor determining software performance, and writers of compilers and high-performance software alike spend much of their time **reverse-engineering and defeating** the sophisticated mechanisms that **automatically** bring data on to and off the chip. Given the large number of transistors devoted to these mechanisms this is an unsatisfactory situation.”*

The problem

- Cell is designed with this in mind
- Codeplay wishes to automatically parallelise software
 - Sieves attempt to solve the parallelism and output problem
 - Inferring prefetching only works in simple cases
 - Software caching in other cases
- Back to fighting the hardware, but now Cell's DMA system

The problem

- Cell is designed with this in mind
- Codeplay wishes to automatically parallelise software
 - Sieves attempt to solve the parallelism and output problem
 - Inferring prefetching only works in simple cases
 - Software caching in other cases
- Back to fighting the hardware, but now Cell's DMA system

Managing memory hierarchy in software is painful!

Problem:

- local memory is scarce, *e.g.*
 - Cell SPE: 256KB for code & data
 - CSX600 PE: 6KB for data
- data-movement hardware constraints, *e.g.* alignment
- forces early optimisation (which is the root of all evil!), *e.g.*
 - choosing data transfer/buffer sizes
 - using double/tripple buffering schemes
- optimisation is not portable and disrupts code base

Solution:

- use a suitable description of memory access patterns
- generate efficient data movement code automatically

Managing memory hierarchy in software is painful!

Problem:

- local memory is scarce, *e.g.*
 - Cell SPE: 256KB for code & data
 - CSX600 PE: 6KB for data
- data-movement hardware constraints, *e.g.* alignment
- forces early optimisation (which is the root of all evil!), *e.g.*
 - choosing data transfer/buffer sizes
 - using double/tripple buffering schemes
- optimisation is not portable and disrupts code base

Solution:

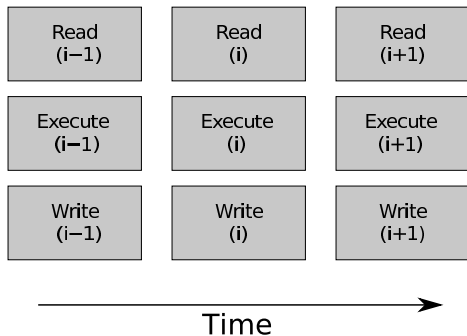
- use a suitable description of memory access patterns
- generate efficient data movement code automatically

Example: 1D simplification of image processing kernel

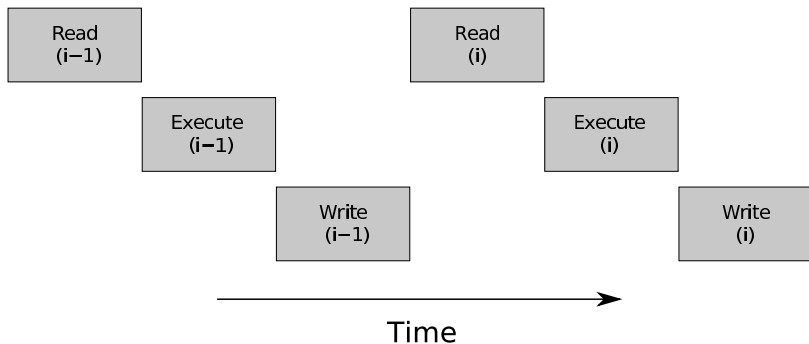
```
void f(float *p, float *q, float *r, int *o, int N)
{
    for(int i = 0; i < N; ++i)
        p[i] = q[i] + r[i + o[i]];
}
```

- array o contains N entries, each in the range $[-1, 2]$
- unless the compiler knows this, it must assume that each iteration accesses an unknown memory location from r , with no information to limit the range of potentially accessed locations and hence no possibility to perform prefetching

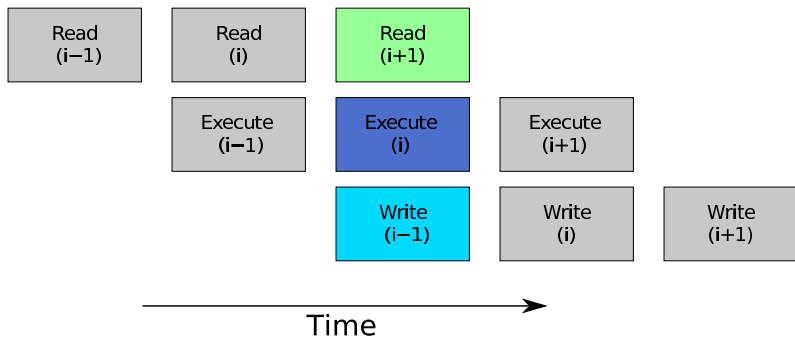
Processing in concept



Processing in reality



Pipelined processing



Managing memory hierarchy in software is painful!

Problem:

- local memory is scarce, *e.g.*
 - Cell SPE: 256KB for code & data
 - CSX600 PE: 6KB for data
- data-movement hardware constraints, *e.g.* alignment
- forces early optimisation (which is the root of all evil!), *e.g.*
 - choosing data transfer/buffer sizes
 - using double/tripple buffering schemes
- optimisation is not portable and disrupts code base

Solution:

- use a suitable description of memory access patterns
- generate efficient data movement code automatically

Decoupled Access/Execute Software

Access/Execute ($\mathcal{A}E$ cute)?

- access: fetch operands/store results
- execute: produce the results

Old hardware idea (Smith, 1984); how about software?

- manually partitioning the program into access and execute parts, or
- running the same program on two different processors

Decoupling is becoming natural, *e.g.* on Cell

- local store access is cheap (6 cycles), *cf.* large register file
- main memory access is expensive (involves DMA)
- decouple **access** to main memory from **execution** in local memory

Decoupled Access/Execute Software

Access/Execute ($\mathcal{A}E$ cute)?

- access: fetch operands/store results
- execute: produce the results

Old hardware idea (Smith, 1984); how about software?

- manually partitioning the program into access and execute parts, or
- running the same program on two different processors

Decoupling is becoming natural, *e.g.* on Cell

- local store access is cheap (6 cycles), *cf.* large register file
- main memory access is expensive (involves DMA)
- decouple **access** to main memory from **execution** in local memory

Decoupled Access/Execute Software

Access/Execute ($\mathcal{A}E$ cute)?

- access: fetch operands/store results
- execute: produce the results

Old hardware idea (Smith, 1984); how about software?

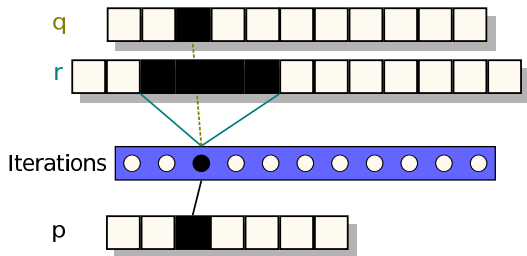
- manually partitioning the program into access and execute parts, or
- running the same program on two different processors

Decoupling is becoming natural, *e.g.* on Cell

- local store access is cheap (6 cycles), *cf.* large register file
- main memory access is expensive (involves DMA)
- decouple **access** to main memory from **execution** in local memory

Example: memory access of a single iteration

```
void f(float *p, float *q, float *r, int *o, int N)
{
  for(int i = 0; i < N; ++i)
    p[i] = q[i] + r[i + rand(-1, 2)];
}
```

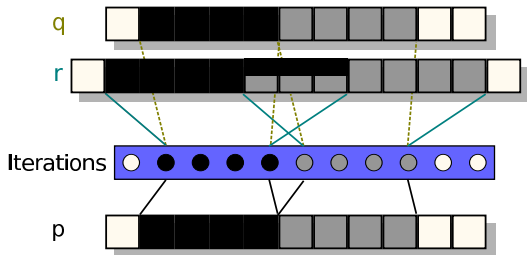


Example: coarse-grain iteration partitioning

```

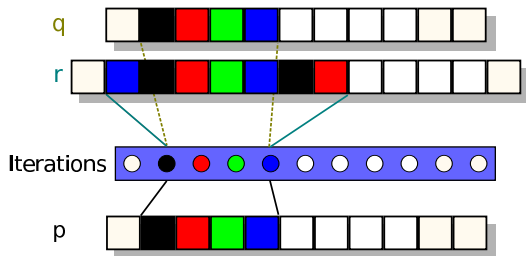
void f(float *p, float *q, float *r, int *o, int N)
{
    for(int i = 0; i < N; ++i)
        p[i] = q[i] + r[i + rand(-1, 2)];
}

```



Example: fine-grain iteration partitioning

```
void f(float *p, float *q, float *r, int *o, int N)
{
  for(int i = 0; i < N; ++i)
    p[i] = q[i] + r[i + rand(-1, 2)];
}
```



Example C++ code

```
// Data sets
Array1D < float > p( p_ptr, p_len );
Array1D < float > q( q_ptr, q_len );
Array1D < float > r( r_ptr, r_len );

// Iteration space
IterationSpace1D iterSpace( 0, N-1 );

// Access regions
Point1D_W pRegion( p, iterSpace );
Point1D_R qRegion( q, iterSpace );
Range1D_R rRegion( r, iterSpace, range(-1, 2) );

// Call kernel, with iteration space and access
// regions abstracting the access pattern
kernel( iterSpace, pRegion, qRegion, rRegion );
```

Decoupled access specification

Part of the interface specification of a kernel:

- sufficient information to infer relationships between points in the iteration space, such as data dependences and reuse
- can be inferred automagically in some cases (although magic can't be relied upon in real programs!)

Potential benefits:

- allows the compiler to synthesise data movement code
- can be used to improve scheduling of the iteration space for efficient hardware mapping
- can be used to infer relationships between distinct kernels (to allow inter-kernel optimisation such as fusion)

Work in progress: C++ version

- Proof of concept
- Embedded within templated data access objects
 - Objects encapsulate data movement management
 - Prompted by Codeplay's interest in self prefetching structures
 - Single kernel codebase for Cell PPE and SPE
- Extends data parallel stream concept

Future work

- Compiler support
 - Inter-kernel flow analysis
 - Aggressive fusion
- Speculation for iteration spaces with dynamic bounds
- Iteration space partitioning methods
- Prove concepts with benchmarks

Summary

- Separation of iteration space and data space
- Each iteration maps to data it accesses
- Iteration space partitioning causes data partitioning
- Low level optimisations driven by partitioning