Deriving Efficient Data Movement from Decoupled Access/Execute Specifications

Lee W. Howes[1], Anton Lokhmotov[1], Alastair F. Donaldson[2], and Paul H.J. Kelly[1]

[1] Department of Computing, Imperial College London
180 Queen's Gate, London, SW7 2AZ, UK
[2] Codeplay Software, 45 York Place, Edinburgh, EH1 3HP, UK

Abstract. On multi-core architectures with software-managed memories, effectively orchestrating data movement is essential to performance, but is tedious and error-prone. In this paper we show that when the programmer can explicitly specify both the memory access pattern and the execution schedule of a computation kernel, the compiler or run-time system can derive efficient data movement, even if analysis of kernel code is difficult or impossible. We have developed a framework of C++ classes for decoupled Access/Execute specifications, allowing for automatic communication optimisations such as software pipelining and data reuse. We demonstrate the ease and efficiency of programming the Cell Broadband Engine architecture using these classes by implementing a set of benchmarks, which exhibit data reuse and non-affine access functions, and by comparing these implementations against alternative implementations, which use hand-written DMA transfers and software-based caching.

1 Introduction

Architectures with software-managed memories can achieve higher performance and power efficiency than traditional architectures with hardware-managed memories (*e.g.* caches), but place additional burden on the programmer. For a traditional architecture, the programmer typically designs a computation kernel and specifies the order in which the kernel traverses the iteration space. To off-load the kernel to a co-processor equipped with local memory, the programmer must additionally manage data movement, to ensure that data is smoothly streamed to and from local memory.

This additional step sounds easier than it actually is. The performance-conscious programmer needs to consider issues such as the optimal data transfer sizes, alignment constraints, exploiting data reuse, *etc.* Moreover, when the working set of a processor is too large to fit in its local memory, the programmer has to use low-level optimisation techniques such as double buffering to overlap computation and communication. Unfortunately, this harms code portability and maintainability.

In this paper, we introduce the decoupled Access/Execute (Æcute—pronounced "acute") programming model, which allows the programmer to express explicitly both the memory access pattern and the execution schedule of a computation kernel, similar to programming traditional architectures. We show that in many cases the compiler or run-time system can derive efficient data movement even if analysis of kernel code is

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 168–182, 2009.

difficult or impossible, thus removing from the programmer the additional complexity of managing data movement.

In the remainder of this paper we argue that decoupling access and execute is natural when programming architectures with software-managed memories (§2) and introduce decoupled Access/Execute specifications (§3). We discuss the prototype Æcute framework (§4) and use examples adapted from linear algebra and signal processing (§3.1 and §5) to show the ease of programming using the specifications. We present experimental results for our examples (§6) obtained on a Cell Broadband Engine (BE) processor and compare them against alternative implementations, which use hand-written DMA transfers and software-based caching. We review related work (§7) and conclude with an outline of future work (§8).

2 Background

Since the 1980s, microprocessor designers have worked hard to preserve the *illusion* of fast memory by providing hardware-managed caches. Sadly, increasing the number of transistors dedicated to caches has been found to achieve diminishing effects on performance. Moreover, optimising software for the memory hierarchy has become the principal activity of performance-conscious programmers and compiler writers, who "spend much of their time *reverse-engineering and defeating* the sophisticated mechanisms that automatically bring data on to and off the chip" [1].

Given this unsatisfactory situation, designers have turned their attention to software-managed memory hierarchies, where data is copied between memories under explicit software control. Examples include the Cell BE architecture from Sony/Toshiba/IBM [1], the CSX SIMD array architecture from ClearSpeed [2], and massively parallel architectures from NVIDIA and ATI (still habitually called graphics processing units, GPUs).

Local memory is typically cheap to access (*e.g.* 6 cycles on Cell), and thus is akin to an extended register file. One some architectures (*e.g.* on Cell and CSX), processing elements can only access local memory, and need to invoke expensive data transfer mechanisms to access remote memory. On other architectures (*e.g.* on GPUs), exploiting local memory is not obligatory but is essential to performance.

Efficient programs are naturally separated into two parts: remote memory access to copy operands in and to copy results out (often asynchronously), and execution in local memory to produce the results. The access and execute parts can be thought of as two concurrent instruction streams. For example, on Cell the execute part runs on an SPE, while the access part is serviced by its DMA engine (Memory Flow Controller).

The separation is reminiscent of decoupled access/execute architectures [3], which run (conceptually or physically) separate access and execute instruction streams. Another point of reference is data-prefetching in virtual shared memory, a shared memory abstraction for computers with physically distributed memories [4]. A program runs on two processors: the access processor prefetches remote data into local memory by performing a partial evaluation of the program; the execute processor performs the full evaluation. The scout-threads in Sun's upcoming Rock processor [5] manifest the same idea, by reading the instruction stream ahead during a memory access stall.

Ideally, to hide the memory latency the access stream runs well in advance of the execute stream. Occasionally the streams need to synchronise, for example, when the execute stream computes an address required by the access stream. Topham *et al.* [6] describe special compiler techniques to minimise the frequence of such *loss of decoupling* events.

Decoupled architectures use either a single original program or two programs derived (manually or automatically) from the original program. We observe, however, that deriving access and execute instruction streams from programs written in mainstream programming languages such as C/C++ is hard, in particular, because of the difficulty of dependence analysis in the presence of aliasing.

3 Decoupled Access/Execute Specifications

We propose a declarative programming model that allows the programmer to annotate a computation kernel with both the execute (§3.2) and access (§3.3) metadata.

3.1 Motivating Example: The Closest-to-Mean Image Filter

The *Closest-to-Mean* (CTM) filter [7] is an effective mechanism for reducing noise in near Gaussian environments, preserving edges more effectively than linear filters whilst offering better performance than computationally expensive median-based filters. For a sample set of vectors V with distance metric δ, the output for the CTM filter is given by the following formula:

$$CTM(V) = \arg\min_{x \in V} \delta(x, \overline{x}),$$

where $\overline{x}$ denotes the sample average value, and $\arg\min_{x \in V}(\textit{expr})$ denotes a value of x that minimises *expr*.

The CTM filter can be applied to a digital $W \times H$ image by mapping each pixel to a CTM value for a $(2K+1) \times (2K+1)$ square sample of neighbouring pixels (for some $K > 0$). Fig. 1 shows a CTM filter implementation in our prototype C++ framework.[1] The class method `kernel` closely resembles the filter's original kernel code, except that accesses to arrays have been replaced with uses of Æcute access descriptors (§4.1).

3.2 Execute Metadata

Definition 1. *Execute metadata for a kernel is a tuple $E = (I, R, P)$, where:*

- $I \subset \mathbb{Z}^n$ *is a finite, n-dimensional* iteration space*, for some $n > 0$;*
- $R \subseteq I \times I$*, is a precedence relation such that $(i_1, i_2) \in R$ iff iteration i_1 must be executed before iteration i_2.*
- P *is a* partition *of I into a set of non-empty, disjont iteration subspaces.*

[1] Note that in all our examples we have compacted construction of member fields into their declarations, to save space.

```
class CTMFilter : public StreamKernel {
  Neighbourhood2D_Read inputPointSet(iterationSpace, input, K);
  Point2D_Write outputPointSet(iterationSpace, output);

  CTMFilter( IterationSpace2D &iterationSpace,
    int K, Array2D &input, Array2D &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]
      }
    }
    mean /= (2*K + 1) * (2*K + 1);
    // compute closest to mean
    rgb closest = inputPointSet(eit, 0, 0); // input[x][y]
    for(int w = -K; w <= K; ++w) {
      for(int z = -K; z <= K; ++z) {
        rgb curr = inputPointSet(eit, w, z); // input[x+w][y+z]
        if( dist(curr, mean) < dist(closest, mean) )
          closest = curr;
      }
    }
    outputPointSet(eit) = closest; // output[x][y]
  }
}
```

Fig. 1. Æcute implementation code for the CTM filter

```
const int K = 2; // 5x5 filter

// 2D iteration space is equivalent to a doubly nested loop:
// parallel for (int x = K; x < W-K; ++x)
//   parallel for(int y = K; y < H-K; ++y)
IterationSpace2D iterationSpace( K, W-K, K, H-K );

// 2D array descriptors
Array2D < rgb >  inputArray( W, H,  &input[0][0] );
Array2D < rgb > outputArray( W, H, &output[0][0] );

// Filter class instantiation
CTMFilter filter( iterationSpace, K, inputArray, outputArray );

// Filter invocation
filter.execute();
```

Fig. 2. Æcute setup and invocation code for the CTM filter

The precedence relationship R specifies constraints on the execution schedule: if iterations i_1 and i_2 are in the relationship, i_1 must be executed before i_2; otherwise, i_1 and i_2 can be executed in any order.

The partition P indicates sets of iterations that it is sensible to execute on the same processing element (*e.g.* a set of iterations that exhibit data reuse). In this paper, we assume that the working set of each $p \in P$ fits into local memory, assuming a set number of buffers (*e.g.* two for double buffering); the programmer either partitions the iteration space manually or opts to use a simple automatic partitioning method which computes the maximum iteration subspace size based on this constraint.

In the CTM filter example, the iteration space is a two-dimensional rectangle congruous with the image dimensions; if the input and output images are disjoint, the execution schedule can be unconstrained, maximising parallelism; and the partition can be tiling into rectangular $w \times h$ tiles, maximising locality:[2]

- $I = \big\{(x, y) : K \leq x < W - K, K \leq y < H - K\big\}$
- $R = \emptyset$
- $P = \big\{\{(x, y) \in I : w(i-1) \leq x - K < wi, h(j-1) \leq y - K < hj\} :$
  $1 \leq i < (W - 2K)/w, 1 \leq j < (H - 2K)/h\big\}$

3.3 Access Metadata

Let M be a set of memory locations.

Definition 2. *Access metadata for a kernel is a tuple $A = (M_r, M_w)$, where:*

- $M_r : I \to \mathcal{P}(M)$ *specifies the set of memory locations $M_r(i)$ that may be* read *on iteration $i \in I$;*
- $M_w : I \to \mathcal{P}(M)$ *specifies the set of memory locations $M_w(i)$ that may be* written *on iteration $i \in I$.*

Often, the set of memory locations accessed on a given iteration is a function of the *iteration vector* (in which case we say that the set is *indexed* by the iteration vector); the set can also include locations that are independent of the iteration vector such as scalars.

In the CTM filter example, the input and output memory locations are indexed:

- $M_r = \big\{\mathrm{input}[x][y] : (x, y) \in I\big\}$;
- $M_w = \big\{\mathrm{output}[x + w][y + z] : (x, y) \in I, -K \leq w, z \leq K\big\}$.

3.4 Æcute Specifications

Definition 3. *An Æcute specification for a kernel is a tuple $S = (A, E)$, where A and E are its access and execute metadata.*

Access metadata 'knows' about memory locations that may be accessed on each iteration, while execute metadata 'knows' about iteration subspaces that are to be executed.

[2] We assume, for simplicity, that the iteration space contains a whole number of tiles.

Given an iteration subspace $p \in P$ and access metadata, we can (over) approximate the set of memory locations that the subspace may read and write: $M_r(p) = \{M_r(i) : i \in P\} \in \mathcal{P}(L)$ and $M_w(p) = \{M_w(i) : i \in P\} \in \mathcal{P}(L)$. Combining execute and access metadata in the form of Æcute specifications enables powerful optimisations such as software pipelining and exploiting data reuse.

In the CTM filter example, Æcute specifications can be used to trigger data prefetching of image rows into local memory, to ensure that the data is delivered in time for processing.

4 Æcute C++ Framework

We have developed a prototype framework to support the Æcute concept, consisting of a set of C++ descriptor classes (§4.1) and a run-time system (§4.2), which compile for the Cell BE architecture.

4.1 The Æcute C++ Classes

The formal iteration space I is specified via an instance of an `IterationSpace` class, which records the number of dimensions and size of each dimension, as in Fig. 2. Practically useful timestamp functions (T) are available in our prototype via the definition of serialised dimensions on iteration spaces, *e.g.* using the `COLUMN_SERIAL` directive. Partitioning of the iteration space is performed in the current prototype with a call to the `setBlockSize` function, which is parameterised with the size of a partition in each dimension of the iteration space.

A kernel class contains a main kernel method parameterised by an iterator to be executed on each point in the iteration space (*e.g.* see Fig. 1). The iterator is used to access indexed memory locations.

The memory mappings M_r and M_w are defined by *access descriptor* classes. An access descriptor object is created for each input or output associated with a kernel, and is invoked from the kernel code, parameterised by an iterator, to gain access to data. The prototype implementation supports the following access descriptor classes. For each member of the iteration space:

- `Point`: returns a single element of the data structure.
- `Neighbourhood`: returns a set of memory locations within a given radius of a primary address.
- `Buffer`: returns a set of points with per point addressing into the data structure based on a combination of the primary address and buffer offset.

In each case the primary address is computed from the iteration space coordinates provided by the Æcute iterator. To these coordinates we may apply a conversion function. In the examples of §5 we see `Project`, `ReAddress`, `Identity` and `BitRev`. `Project` is an affine scaling function. `ReAddress` is a proxy for applying separate conversions to each dimension: in our examples, the identity function `Identity` and a custom bit-reversal function, `BitRev`. The prototype framework can be extended with custom conversion functions for specific applications.

4.2 The Æcute Run-Time System

The Æcute run-time system comprises two components: a PPE run-time and an SPE run-time, an instance of which runs on each active SPE.

The PPE run-time spawns an SPE run-time process on each available SPE. Based on an iteration space partitioning specified by the programmer, or via a partitioning obtained automatically at run-time by querying access metadata, the PPE run-time generates a list of partition identifiers. Partition identifiers are farmed out to the SPEs, which are responsible for executing the kernel iterations associated with each partition. Once all partitions have been assigned, the PPE run-time waits for completion reports from all SPEs before returning control to the main program.

On initialisation all access descriptors in the SPE instance create a series of buffers based on the maximum partition size. At least one buffer will be present in each input and output descriptor, and possibly more if the configuration specifies this.

An instance of the SPE run-time repeatedly receives a list of partition identifiers from the PPE. The SPE instance takes each partition identifier in turn and converts that into a set of iterations. The conversion is possible because the SPE code is constructed using the same iteration space data as the PPE code. This information is partially static, and partially based on parameters passed on construction from the PPE side.

The partition information is passed to the access descriptors assigned to the kernel, which select available data buffers and construct appropriate DMA operations to copy data in. When no buffer is available, a blocking call to wait on DMA writes is initiated to allow buffers to be cleared and reused. The kernel checks that the data it needs for a given partition identifier is available by querying the access objects, and will block on the DMA reads if it is not. On completion of computation, partition completion information is passed to the access objects which will perform DMA write backs and free buffers as appropriate.

Double or triple buffering naturally occurs through this system, as a fixed buffer set is automatically managed to ensure that data is always available, without additional programmer intervention. This multiple buffering enables dynamic software pipelining of the execution to improve the efficiency of memory access. In addition, the run-time system will maintain buffers without reloading, or without writing back early, if it detects that it already had the appropriate data resident in an appropriate buffer.

5 Further Examples

5.1 Matrix-Vector Multiply

A matrix-vector multiply $y = Ax$ can be implemented as a two-dimensional iteration space of the dimensions of matrix A. Vectors x and y are one-dimensional, so we *project* the iteration space to obtain the vector indices.

Æcute specification $S = ((M_r, M_w), (I, T, P))$:

- $I = \{(i, j) : 0 \leq i < H, 0 \leq j < W\}$
- $R = \{((i, j), (i, k)) : 0 \leq i < H, 0 \leq j < k < W\}$

- $M_r(i,j) = \{A[i][j], x[j]\}$
- $M_w(i,j) = \{y[i]\}$
- $P = \big\{\{(i,j) \in I : h(k-1) \leq i < hk, w(l-1) \leq j < wl, \} :$
  $1 \leq k < H/h, 1 \leq l < W/w\big\}$

(As before, we tile the iteration space, assuming local memory can hold the working set for a rectangular tile of $h \times w$ iterations.)

The precedence relation indicates that the loop indexed by i is parallel and the loop indexed by j is serial. This serialisation removes the requirement for PPE-side accumulation of partial results. If the += operator could be guaranteed to be associative then the j loop could also be specified as parallel, by setting $R = \emptyset$.

Æcute code. The kernel operates over the input matrix and vector and the output vector. Note that we specify that the column dimension is serial, which preserves the order of multiply-accumulate operations.

```
IterationSpace2D iterationSpace(W, H, COLUMN_SERIAL);
Array2D < float >  inMatrix(H, W, pInMatrix);
Array1D < float >  inVector(W, pInVector);
Array1D < float > outVector(H, pOutVector);
MatrixVectorMul matvec(iterationSpace, inMatrix, inVector, outVector);
// Matrix-vector multiply invocation
matvec.execute();
```

The `MatrixVectorMul` kernel class is roughly as follows:

```
class MatrixVectorMul : public StreamKernel {
  Point2D_Read inputMatrix( iterationSpace, inMatrix);
  Point2D_Read < Project2D1D< 1, 0 >  >
    inputVector( iterationSpace, inVector );
  Point2D_Write < Project2D1D< 0,1 >  >
    outputVector( iterationSpace, outVector );

  MatrixVectorMul( IterationSpace 2D iterationSpace,
    Array2D inMatrix, Array1D inVector, Array1D outVector ){...}

  void kernel( const IterationSpace2D::element_iterator &eit ) {
    outputVector( eit ) += inputVector( eit ) * inputMatrix( eit );
  }
};
```

where `Project2D1D` projects a 2D-space point onto a 1D-space point. For example, `Project2D1D<0,1>` projects (i,j) onto j.

5.2 Bit-Reversal

Many radix-2 FFT algorithms start or end their processing with data permuted in bit-reversed order. The reordering is typically done by a special subroutine, called

bit-reversed data copy (often abbreviated, if inaccurately, to *bit-reversal*). We assume that the subroutine reads an array $x[\,]$ of $N = 2^n$ elements and writes these elements into an array $y[\,]$ of N elements, such that x and y do not overlap, in bit-reversed order. That is, an element of the source array at the index written in binary as $b_0 \ldots b_{n-1}$, is copied to the target array at the index with reversed digits $b_{n-1} \ldots b_0$. The function $\sigma_n(i)$ reversing bits of index i having n bits can be implemented as [8]:

```
unsigned int reverse_bits(unsigned int n, unsigned int i) {
    i = (i & 0x55555555) << 1 | (i >> 1) & 0x55555555;
    i = (i & 0x33333333) << 2 | (i >> 2) & 0x33333333;
    i = (i & 0x0f0f0f0f) << 4 | (i >> 4) & 0x0f0f0f0f;
    i = (i<<24) | ((i & 0xff00)<<8) | ((i>>8) & 0xff00) | (i>>24);

    return (i >> (32 - n));
}
```

Few programmers will recognise that this sequence of bit-wise operations and shifts implies that $y[\,]$ will contain a permutation of $x[\,]$ and hence assignments can be performed in any order. One cannot expect that a compiler will recognise this either.

In addition to obscuring parallelism, bit-reversed indexing is unfriendly to hardware-managed caches: starting from a certain array size $N = 2^n$, each access to $y[\,]$ results in a cache miss. To avoid cache associativity problems inherent in bit-reversals of large arrays, the best approach, used by Carter and Gatlin in the so-called Cache Optimal BitReverse Algorithm (COBRA) [9], introduces a cache-resident buffer.

If the buffer holds B^2 elements, the iteration space is partitioned into N/B^2 independent subspaces. For each subspace, B source blocks of B elements each are copied into the buffer, permuted *in place*, and then copied out from the buffer into B target blocks of B elements each.

The permute kernel of COBRA can be off-loaded to a co-processor having local memory. The challenge is to implement the copy in and copy out loops, where the copy out loop uses a non-affine access function $\sigma_n(i)$.

Somewhat surprisingly, implementing data movement code can take longer than implementing the kernel proper (according to the experience of one of the authors). Again, a desired alternative is to derive data movement from Æcute specifications.

Æcute specification $S = \big((M_r, M_w), (I, R, P)\big)$:

- $I = \{t : 0 \le t < N/B^2\}$
- $R = \emptyset$;
- $P = \big\{\{t\} : t \in I\big\}$
- $M_r(t) = \{x[u.t.v] : t \in I, 0 \le u < B, 0 \le v < B\}$
- $M_w(t) = \{y[u.\sigma_n(t).v] : t \in I, 0 \le u < B, 0 \le v < B\}$.

The precedence function indicates that the one-dimensional iteration space is unordered. In this case each partition is a single element of the iteration space, because the blocks are disjoint and fairly large. In the Æcute code below we see that the programmer can manually set the partition size.

Ækute code As a result of the $B \times B$ blocking, it is natural to think of the input and output arrays of N elements as two-dimensional, having N/B rows of B elements each.

```
IterationSpace1D iterationSpace(N/(B*B));
Array2D <float> inputData(B, N/B,  pInputData);
Array2D <float> outputData(B, N/B, pOutputData);
BitReversal bitrev(iterationSpace, inputData, outputData);
bitrev.iterationSpace.setBlockSize( 1 );
// Bit-reversal invocation
bitrev.execute();
```

We iterate over independent subspaces $t \in I$, copying rows numbered as $u.t, 0 \leq u < B$, into the local buffer, applying the kernel, and copying rows numbered as $u.\sigma_n(t), 0 \leq u < B$, from the local buffer.

```
class BitReversal : public StreamKernel< BitReversal > {
    Buffer2D_Read
        input(iterationSpace, inputData, B);
    Buffer2D_Write < ReAddress2D< Identity, BitRev > >
        output(iterationSpace, outputData, B);

    BitReversal( IterationSpace 2D iterationSpace,
        Array2D input, Array2D output ) {...}

    // Do in place permutation
    void kernel(const IterationSpace2D::element_iterator &eit){
        ...
    }
};
```

`ReAddress` takes the (i, j) coordinate formed from the iteration space point and the buffer coordinates and applies the specified pair of functions to i and j respectively. `BitRev` reverses bits of the j value to correctly address the destination for the row by calling the `reverse_bits` function shown earlier.

6 Experimental Evaluation

We use a 3.2GHz Cell processor on a Sony PlayStation 3 console, running Fedora Linux (2.6.23.17-88.fc7), with IBM Cell SDK 2.1. We compiled the benchmark programs using the highest optimisation settings, and executed them on all six SPEs that are available to the programmer on a PlayStation 3.

6.1 Implementation

We evaluate our prototype Ækute framework against alternative implementations, which use hand-written DMA transfers and a software-based SPE cache. The cache allows remote data to be accessed in a familiar way, so that code can be quickly ported to run on an SPE. In our experiments, we use the standard cache implementation provided with SDK 2.1 [10]. We use a 4-way set associative cache with default "write-back" write policy and "round-robin" replacement policy, and vary the number of cache sets and line size on an application-specific basis.

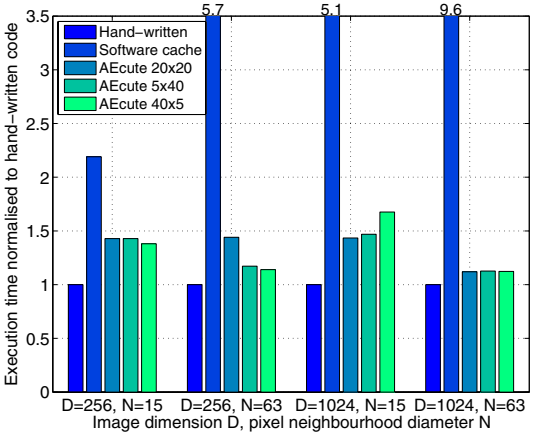

Fig. 3. Closest-to-mean filter

The kernel code is essentially the same, with minor changes to support the use of Æcute framework classes and software cache functions.

6.2 Benchmark Details

We evaluate the benchmarks described in §3 and §5.

Closest-to-mean filter (§3.1). Fig. 3 shows execution time normalised to code with hand-written DMA transfers. We consider two neighbourhood diameters N: 15 and 63, and two image sizes $D \times D$ where D is: 256 and 1024. These represent increasing computation workload. We also consider three different iteration space tile sizes: 20×20 (default square size, which is calculated automatically under the constraint that the tile footprint must fit into local memory); 5×40; and 40×5.

For $D = 256$ and $N = 15$, the best Æcute code performs within 40% of hand-written code; for $N = 63$, within 15%: the increased workload amortises the overhead of interpreting Æcute specifications. In contrast, the overhead of using the software cache grows with increasing neighbourhood size (which perhaps can be remedied by tuning the cache parameters). For $D = 1024$ and $N = 63$, the overhead drops to 12%.

We observe that no tile size was universally best. Given the simplicity of varying tile sizes, the best tile size could be found by iterative search. In contrast, it is usually more difficult to adapt code with hard-coded tile sizes.

Blocked DMA transfers, which are supported naturally by the partitioning and automated buffering in the Æcute system, and implemented in the hand written code, improve the efficiency of memory traffic and enable both hand-written and Æcute code perform far better than code using the software cache.

Matrix-vector multiply (§5.1). For this example we hand-vectorised the entire block computation for efficiency. The hand written and software cache based code are similarly vectorised for fair comparison. While the Æcute model looks promising for

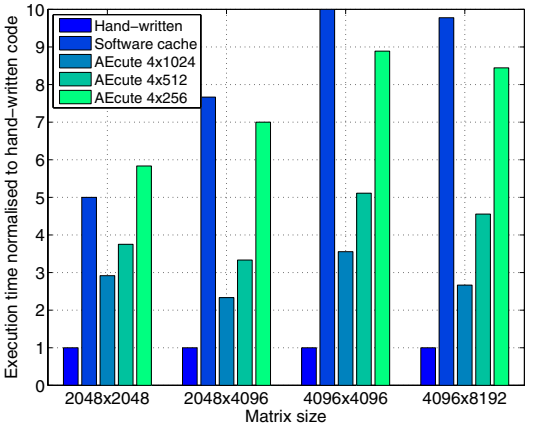

Fig. 4. Matrix-vector multiply normalised to execution time of hand written code

automatic vectorisation, it is important that the programmer retains full control over kernel optimisations should automatic optimisations fail.

Fig. 4 shows normalised execution time for various matrix sizes. The best tile size is 2–3 slower than hand-written code, but considerably faster than the software cache implementation. The run-time overhead associated with the Æcute framework is significant for this example due to the low arithmetic density of the matrix-vector multiply operation. The hand-written implementation requires less SPE-PPE communication: the SPEs are able to compute results entirely independently.

Bit-reversal (§5.2). Fig. 5 plots execution time in milliseconds against $n = \log_2 N$, the bitwidth of the array index. We see smooth scaling of performance with the size of the dataset. In addition, the performance of the Æcute implementation tracks that of the hand-written implementation with a near-constant scaling. In this case, while remote memory accesses are inherently non-contiguous due to bit-reversed indexing in the algorithm, the system can construct efficient DMA list transfers from Æcute specifications.

7 Related Work

Recent work by Solar-Lezama *et al.* on sketching [11] aims to automate the optimisation of simple computation kernels. Where the Æcute model defines iteration spaces and memory access patterns declaratively to localise memory access, sketching supports a rough definition of an optimised implementation and attempts to search for a series of transformations to convert one to the other.

Saltz *et al.* [12] propose run-time parallelisation of loop nests that defy compile-time dependence analysis. At run-time, *inspector* procedures identify parallel wavefronts of loop iterations, which *executor* procedures then distribute among processors. In contrast, our approach relies on the programmer to supply information that the compiler may fail to extract from the program.

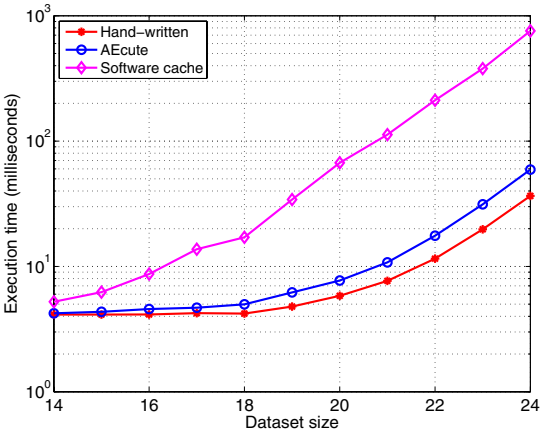

Fig. 5. Bit-reversal

The emergence of architectures having software-managed memories (in particular, of Cell) has spurred the development of high-level programming abstractions, addressing the issue of copying data between distributed memories.

Sequoia. The Sequoia language [13] from Stanford University abstracts parallelism and communication by introducing *tasks*: side-effect free methods using the call-by-value-result (CBVR) argument passing mechanism. The abstract machine model of Sequoia is a tree of (physical or virtual) memory modules. Each task runs at a single node of the tree and can directly access memory only at this node. Tasks can spawn subtasks on the same node or child nodes. Upon calling a subtask, input data from the caller's address space is copied into the callee's address space, output data is computed and then copied out into the caller's address space on return.

CellSs. CellSs [14] is a programming model for the Cell architecture from Barcelona Supercomputing Centre. Similar to Sequoia, CellSs annotations to C programs specify a task for execution on the SPEs and its arguments.

Sieve C++. In Sieve C++ [15][16], a C++ extension from Codeplay Software, the programmer can place a code fragment inside a *sieve scope*—a new lexical scope prefixed with the **`sieve`** keyword—thereby instructing the compiler to *delay* writes to memory locations defined outside of the scope (global memory), and apply them *in order* on exit from the scope. The semantics of sieve scopes can be considered as generalising to composite statements the semantics of the Fortran 90 single-statement vector assignments [17]. This semantics, named call-by-value-delay-result (CBVDR) [15], disallows flow dependences and preserves name dependences on data in global memory, and by restricting dependence analysis to data in local memory makes C++ code more amenable to automatic parallelisation.

8 Conclusion and Future Work

We have presented the concept of decoupled Access/Execute specifications and demontrated their convenience, flexibility and efficiency on three benchmark examples. Our Æcute implementation automates the data movement element of the accelerator programming task. The blocking of DMA transfers and construction of DMA lists enabled by separating the memory access from computation results in more efficient memory traffic.

We are looking into extending this work in several ways.

First, Æcute specifications may be thought of at a level of compiler intermediate representation rather than a high level programming language. Thus, we plan to investigate 'front-ends' that will derive Æcute specifications from higher-level abstractions, in particular, from the polyhedral model [18]. In addition, we wish to investigate 'back-ends' for other accelerator architectures, such as GPUs.

Second, we plan to integrate Æcute specifications into a compiler, to reduce both the overhead of interpreting Æcute specifications at run-time and the size of generated data-movement code, which must be minimised to conserve precious local memory. As in Gaster's streaming extension to OpenMP [19], compiler support can be layered on top of an extention and streamlining of the current Æcute classes, allowing the application to work correctly with or without compiler support.

Adding compiler support is related to our work on metadata-enhanced component programming [20], which uses Æcute-like metadata, describing the input-output interfaces of components, such that combining the components can optimise data flow at run-time. We aim to achieve similar optimisations by applying fusion optimisations to Æcute kernels.

Third, we plan to extend the expressivity of Æcute metadata to handle a larger set of kernels, associated with full scale applications. The current Æcute implementation supports only a limited range of partitioning options and mappings to data. We can extend this by using a hierarchical partitioning and improving the search options, *e.g.* for locality. In addition, we wish to support unstructured mesh based computations, such as fluid flow. For unstructured data we need to extend the memory read and write sets to support indirection while maintaining decoupling of access and execute.

Acknowledgements. We thank Mike Gist for his implementation of matrix-vector multiply, the anonymous reviewers and Alec-Angus Macdonald for their helpful comments, and the EPSRC for funding this work through grant number EP/E002412/1.

References

1. Hofstee, H.P.: Power efficient processor architecture and the Cell processor. In: Proceedings of the 11th International Conference on High-Performance Computer Architecture (HPCA), pp. 258–262. IEEE Computer Society, Los Alamitos (2005)
2. ClearSpeed Technology: The CSX architecture, `http://www.clearspeed.com/`
3. Smith, J.E.: Decoupled access/execute computer architectures. ACM Trans. Comput. Syst. 2(4), 289–308 (1984)

4. Watson, I., Rawsthorne, A.: Decoupled pre-fetching for distributed shared memory. In: Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS), Washington, DC, USA, pp. 252–261. IEEE Computer Society, Los Alamitos (1995)
5. Tremblay, M., Chaudhry, S.: A third-generation 65nm 16-core 32-thread plus 32-scout-thread CMT SPARC processor. In: Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC) (2008)
6. Topham, N., Rawsthorne, A., McLean, C., Mewissen, M., Bird, P.: Compiling and optimizing for decoupled architectures. In: Proceedings of Supercomputing (SC), p. 40 (1995)
7. Lau, D.L., Gonzalez, J.G.: The closest-to-mean filter: an edge preserving smoother for Gaussian environments. In: Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 2593–2596. IEEE Press, Los Alamitos (1997)
8. Warren, H.S.: Hacker's Delight. Addison-Wesley, Boston (2002)
9. Carter, L., Gatlin, K.S.: Towards an optimal bit-reversal permutation program. In: Proceedings of Foundations of Computer Science (FOCS), pp. 544–555 (1998)
10. Wright, C.: IBM software development kit for multicore acceleration. Roadrunner tutorial LA-UR-08-2819 (2008), `http://www.lanl.gov/orgs/hpc/roadrunner`
11. Solar-Lezama, A., Arnold, G., Tancau, L., Bodik, R., Saraswat, V., Seshia, S.: Sketching stencils. In: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI), pp. 167–178. ACM, New York (2007)
12. Saltz, J.H., Mirchandaney, R., Crowley, K.: Run-time parallelization and scheduling of loops. IEEE Trans. Comput. (5), 603–612 (1991)
13. Fatahalian, K., et al.: Sequoia: programming the memory hierarchy. In: Proceedings of Supercomputing (SC), pp. 83–92 (2006)
14. Bellens, P., Perez, J.M., Badia, R.M., Labarta, J.: CellSs: a programming model for the Cell BE architecture. In: Proceedings of Supercomputing (SC), pp. 86–96 (2006)
15. Lokhmotov, A., Mycroft, A., Richards, A.: Delayed side-effects ease multi-core programming. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 641–650. Springer, Heidelberg (2007)
16. Codeplay Software: Portable high-performance compilers, `http://www.codeplay.com/`
17. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann, San Francisco (2002)
18. Griebl, M.: Automatic Parallelization of Loop Programs for Distributed Memory Architectures. University of Passau, Habilitation Thesis (2004)
19. Gaster, B.R.: Streams: Emerging from a shared memory model. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 134–145. Springer, Heidelberg (2008)
20. Howes, L.W., Lokhmotov, A., Kelly, P.H., Field, A.J.: Optimising component composition using indexed dependence metadata. In: Proceedings of the 1st International Workshop on New Frontiers in High-performance and Hardware-aware Computing (2008)