Compile-time and Run-time Issues in an Auto-parallelisation system for the Cell BE Processor

Alastair F. Donaldson[1], Paul Keir[2], and Anton Lokhmotov[3]

[1] Codeplay Software, 45 York Place, Edinburgh, EH1 3HP, UK
[2] Department of Computing Science, University of Glasgow, 18 Lilybank Gardens, Glasgow, G12 8QQ, UK
[3] Department of Computing, Imperial College London, 180 Queen's Gate, London, SW7 2AZ, UK

Abstract. We describe compiler and run-time optimisations for effective auto-parallelisation of C++ programs on the Cell BE architecture. Auto-parallelisation is made easier by annotating *sieve scopes*, which abstract the "read in, compute in parallel, write out" processing paradigm. We show that the semantics of sieve scopes enables data movement optimisations, such as re-organising global memory reads to minimise DMA transfers and streaming reads from uniformly accessed arrays. We also describe run-time optimisations for committing side-effects to main memory. We provide experimental results showing the benefits of our optimisations, and compare the Sieve-Cell system with IBM's OpenMP implementation for Cell.

1 Introduction

The Cell Broadband Engine (BE) processor [4] is a heterogeneous multi-core chip, which consists of a Power Processing Element (PPE) and eight Synergistic Processing Elements (SPEs). To avoid memory bottlenecks, each SPE is equipped with 256KB of fast local memory, which can be viewed as an extended register file for intensive calculations, and accesses main memory via DMA transfers. This approach allows scalable parallelisation over SPEs for suitable algorithms. Abandoning the convenient shared memory paradigm, however, makes the Cell processor difficult to program correctly and efficiently: the programmer needs to write separate programs for the PPE and SPEs, pack data into vectors for SIMD processing, and orchestrate data movement explicitly using untyped DMA transfers.

Codeplay's Sieve C++ [2, 5] is a C++ extension to aid automatic parallelisation. The principal language construct is the *sieve block* – a lexical scope prefixed with the `sieve` keyword. By placing code inside a sieve block, the programmer instructs the compiler to *delay* writes to memory locations defined outside the block (global memory) and apply them *in order* on exit from the block. Conceptually, global memory is read on entry to the block and written to on exit from the block. Thus, the compiler is free to re-order computation within a sieve block, if it has no dependences on memory locations defined within the block (local memory). Restricting dependence analysis to local memory makes C++ code more amenable to deterministic automatic parallelisation.

In the context of the Cell processor, a sieve block makes explicit the notion of separate memory spaces: code outside a sieve block runs on the PPE and accesses main memory as usual; code inside a sieve block is a candidate for parallelisation over SPEs, with local variables to be placed in local store. The sieve semantics enables streaming between global memory and local store.

Sieve blocks are similar to *tasks* in Stanford's Sequoia [3] and BSC's CellSs [1] in that they specify a fragment of code to be executed on SPEs. Unlike a task, a sieve block leaves unspecified the working set of code, making optimisation of sieved code more challenging. However, while a task (a leaf task in Sequoia) is intended for execution on a single SPE, the sieve semantics parallelisation of a sieve block across multiple SPEs. The sieve construct is similar to the bulk-synchronous parallel (BSP) model [6]: it separates computation (on data brought into local memory) and communication (of results into global memory). Communication in BSP, however, is non-deterministic.

We have previously described Sieve C++ and its other constructs facilitating automatic parallelisation via software thread-level speculation [2]. Our contributions in this paper are: a discussion of the components of the Sieve-Cell system (§2); the description of optimisation techniques concerning the movement of data between main memory and local store (§3); and a comprehensive experimental evaluation showing the speedups afforded by our optimisations and comparing the Sieve-Cell system with IBM's OpenMP implementation for Cell (§4). We conclude (§5) with an outline of future work.

2 Sieve Overview

We illustrate the sieve concept and its advantages using a molecular dynamics example (§2.1), and describe a sieve implementation for the Cell BE processor (§2.2).

2.1 Sieve scopes and outer pointers in a molecular dynamics example

In addition to marking sieve blocks, the **`sieve`** keyword can be used as a function qualifier indicating that the function may be called from a sieve block (or other sieve functions) and therefore should be compiled with the semantics of delayed writes to global memory. (Sieve blocks and functions constitute sieve *scopes*.)

The **`outer`** qualifier applied to a pointer declared within a sieve scope indicates that the pointer points to data in global memory. (Pointers declared outside a sieve scope are outer by default.) Hence, writes via outer pointers occuring in a sieve scope get delayed.

The following listing shows a function, `computeForces`, which takes input arrays representing masses and positions for each particle in a system, and computes an output array representing forces exerted upon each particle, according to the law of gravity:

```
extern int Size;
sieve float3 rNormalised(outer float3 *Pos, int i, int j);

void computeForces(float3 *Forces, float3 *Pos, float* Mass) {
  sieve {
    for(int i=0; i<Size; ++i) {
      float3 Potential = { 0.0f, 0.0f, 0.0f };
      for(int j=0; j<Size; ++j)
```

```
        Potential -= rNormalised(Pos, i, j) * Mass[j];
      Forces[i] = Potential * Mass[i]; // Delayed write
  } } // Side-effects to Forces[] committed here
}
```

Listing 1. Molecular dynamics code, annotated with a sieve block.

We do not show code for the `float3` class (a floating point vector class with standard operations), or for the sieve function `rNormalised` which, given `Pos`, `i` and `j`, returns $(0,0,0)$ if `i`$=$`j`, and $(\texttt{Pos[i]}-\texttt{Pos[j]})/|(\texttt{Pos[i]}-\texttt{Pos[j]})|^3$ otherwise.

Consider the standard C++ code obtained by removing the **`sieve`** and **`outer`** keywords from Listing 1. In this form, the code is hard to automatically parallelise for the following reasons. First, the compiler must conservatively assume that the arrays `Forces` and `Mass` may overlap, which would lead to a carried dependence on the outer loop due to the write to `Forces`. Second, note that both loops are bounded by the external global variable `Size`. The compiler has to assume that `Size` may be modified during calls to the `rNormalised` function (for which the compiler does not necessarily have source code), which would destroy the regular structure of the loops. Similarly, the array `Pos` is passed as a parameter to `rNormalised`. The compiler must therefore assume that memory accessed via `Pos` could be modified by this function call, which would introduce carried dependences on both loops in the nest.

For the molecular dynamics example, this conservativeness is due to the mechanical compiler lacking domain-specific information. In a sensibly-written application, `computeForces` will be called with arrays that do *not* overlap, and `Size` and `Pos` will *not* be modified by the pure function `rNormalised`. The intention that the input parameters refer to distinct memory regions can be stated using the C99 **`restrict`** qualifier, but restricted pointers do not help with a possible modification by the called function.

The sieve block of Listing 1 tears down these barriers to parallelisation. The `Pos`, `Mass` and `Forces` parameters are outer pointers, since they are declared outside the sieve block. As a result, during sieve block execution, any modification to data via these pointers is delayed until the end of the block. This change in semantics means that there are no loop-carried dependences even if the arrays do overlap. The compiler knows that `rNormalised` is a sieve function, compiled with delayed semantics, so that if the function (or a function it calls in turn) writes to global variable `Size`, or via outer pointer `Pos`, these writes will be delayed, having no effect on sieve block computation.

Clearly, the semantics of the sieve block will depart from the conventional semantics if the code does involve a write to, followed by a read from, a global memory location. While parallelising such code with standard semantics would result in nondeterministic, undefined behaviour, the sieve semantics mean that parallel code is deterministic, regardless of how many cores are employed. If sieved code does not behave as expected (due to a programmer error, or to a sieve block being applied to code which involves read-after-write dependences by design) the guarantee of determinism means that the programmer can debug their multi-core application on a *single* core.

2.2 The Sieve-Cell System

The Sieve concept fits neatly with systems having multiple levels of memory hierarchy, in particular, the Cell BE processor. The programmer uses a sieve block to specify that

a portion of code should be distributed across the SPEs. Variables declared inside sieve scopes reside in SPE local store; variables declared in standard scopes are located in main memory, and data structures in main memory can be traversed on an SPE via outer pointers. A read inside a sieve block from global memory results in the transfer of data into local store via DMA; a write to global memory results in an entry being appended to a queue of side-effects, to be applied at the end of the sieve block. The Sieve compiler and run-time system take care of the low-level details associated with data movement. The programmer is able to write a unified application for the Cell processor, rather than being forced to write separate PPE and SPE programs with explicit communication via mailbox messages and DMA transfers (which cannot be typechecked). Annotating blocks of code, as opposed to outlining code into functions, also aids productivity, allowing Sieve versions of serial C++ codes to be developed quickly.

Sieve Compiler The compiler processes a Sieve C++ application and outputs a set of ANSI C files, which we refer to as OutputC files, together with a makefile. The makefile uses third party C compilers for the PPE and SPE processors to compile these C files, linking the resulting object files with the Sieve run-time library and other PPE/SPE libraries to produce a Cell executable. The Sieve compiler provides full support for PPU and SPU vector intrinsics.

The **`sieve`** and **`outer`** keywords allow type-checking across the PPE/SPE boundary, ensuring for example that SPE code does not accidentally de-reference a pointer to PPE data. The occurrence of a sieve block in a Sieve C++ application causes a call to a function named `runSieve` to be generated at a corresponding position in the OutputC code. This function is part of the Sieve-Cell PPE run-time library. Given a pointer to an SPE program for the associated sieve block, the `runSieve` function manages the distribution of sieve scope execution across available SPEs.

PPE Runtime After loading each SPE with the sieve block program,[4] the PPE runtime issues each SPE with a speculative work unit. A work unit consists of a program point in the sieve block at which execution should begin, called a *split point*, together with speculated values for variables which are live at the given split point, and an integer specifying how many further split points should be crossed before execution of the work unit is completed [2]. Execution of a work unit also completes if the end of the sieve block is reached before the given number of split points is crossed. To ease speculation (*e.g.* for automatic loop parallelisation), the compiler requires that the only variables live across split points are *iterators* – instances of special user-defined classes having methods for prediction of class state after traversal of a given number of split points.

After issuing work units, the PPE run-time sleeps, waking up when an SPU thread interrupts to indicate completion of work. On receiving a completed work unit from an SPE, the PPE issues the SPE with a fresh work unit, then attempts to *validate* the completed work – checking that the predicted iterator values for the work unit match the actual values computed by the previous work unit. A completed work unit contains a queue of associated side-effects. Side-effects for a valid work unit are kept until the end of the sieve block; side-effects for an invalid work unit are discarded. At the end of

[4] A run-time check ensures that SPEs are not needlessly re-issued sieve block code if the same sieve block is executed in succession.

the sieve block, the PPE run-time invokes a function, `runSideEffects`, which takes the side-effects generated by all valid work units (by which time all side-effects have been transferred to main memory), and commits these side-effects *in order*.

SPE Runtime The SPE program for a sieve block consists of a small run-time system loop together with code for work unit execution, which must all fit within the SPE local store. Each SPE has an outbound interrupt mailbox, which it uses to request a work unit from the PPE. Using the interrupt mailbox means that the PPE can sleep when not servicing an SPE, avoiding a busy-wait loop. The PPE responds to the SPE, via the SPE's inbound mailbox, with a pointer to a work unit. The SPE uses this pointer to fetch the work unit via DMA, after which the SPE executes the work unit.

By default, reads from main memory are via an SPE software cache, based on an implementation provided by IBM as part of the Cell SDK for Linux. Alternative methods for reading data from main memory are discussed in §3. For each write to main memory inside a sieve scope, the compiler generates a call to the `delayedStore` function in the SPU run-time. This function takes pointers to a SPU source address and PPU destination address, and an integer specifying the size of the data to be stored. The function adds a (*PPU address*, *size*, *data*) triple to a queue on the SPE. When an SPE's local side-effect queue reaches an upper bound (specified at compile-time), the SPE transfers the contents of its queue to a temporary location in main memory, and continues execution with an empty local queue. The PPE run-time is responsible for managing SPE requests to allocate main memory for temporarily storing side-effects.

3 Sieve-Cell optimisations

Efficient data movement is key to achieving high-performance on the Cell processor. We describe compiler and run-time data-movement optimisations in the Sieve-Cell system.

3.1 Streaming DMA Reads

A common DMA transfer optimisation involves fetching data from main memory in large chunks before processing. For example, given an SPU loop which on each iteration fetches and processes an element of main memory array `A[0..N-1]`, it is typically more efficient to transfer `A[0..N-1]` into local memory *before* executing the loop. This is due to high cost of initiating a DMA transfer compared with the cost of transferring additional bytes once a DMA has been initiated. If `N` is large then it may be worth overlapping communication with computation, streaming data from `A` in chunks and using double-buffering to process chunk n while fetching chunk $n+1$. Indeed, streaming may be necessary if `N` is large enough that `A[0..N-1]` does not fit into local store.

The Sieve compiler exploits the delayed semantics to generate efficient DMA streaming code when compiling regularly structured loops which read through **`outer`** pointers. We illustrate this using the molecular dynamics example of Listing 1 as follows:

```
void computeForces(float3 *Forces, float3 *Pos, float* Mass) {
  sieve {
    DMAStream<sizeof(float)> MassStr_i, MassStr_j;
    int LocalSize = Size;
```

```
    MassStr_i.start(Mass);
    for(int i=0; i<LocalSize; ++i) {
      float3 Potential = { 0.0f, 0.0f, 0.0f };
      MassStr_j.start(Mass);
      for(int j=0; j<LocalSize; ++j)
        Potential -= MassStr_j.read(j) * rNormalised(Pos, i, j);
      Forces[i] = Potential * MassStr_i.read(i);
    }
    MassStr_j.destroy(); MassStr_i.destory();
} }
```

Listing 2. Optimised molecular dynamics code.

The compiler spots that the outer loop of the sieve block includes a statement reading from the base address `Mass` with offset `i`. Since `i` can be identified as an index variable for the outer loop, the compiler generates a DMA stream object, `MassStr_i`, and replaces the read from `Mass[i]` with a read from `MassStr_i`. Similarly, the regular reads from `Mass` offset by index variable `j` in the inner loop are replaced with reads from a stream, `MassStr_j`.[5]

A DMA stream can be thought of as a window into an array in main memory. In our double-buffered implementation, a stream is an SPU-side record consisting of a pair of buffers, a pointer to the *current* buffer, a base address for the PPU array to which the stream corresponds, an address indicating the main memory address to which the current buffer refers, and a DMA tag to monitor completion of prefetching operations.

The declaration `DMAStream<`*elem_size*`>` *stream_name* declares a DMA stream with a fresh DMA tag, with the capacity to store *elem_num* elements of size *elem_size* (where the *elem_num* parameter may vary at run-time). A fresh tag means that simultaneous DMA requests for multiple streams can be issued simultaneously and managed independently. The *stream_name*`.start(`*base_address*`)` operation issues and waits for a DMA operation to copy *elem_size* $\times$ *elem_num* bytes of data into the *current* buffer for the stream. A DMA request to fill the other buffer with the next *elem_size* $\times$ *elem_num* bytes of data is also issued, but not waited for. The *stream_name*`.read(`*offset*`)` first checks that the current buffer contains sufficient data to return the element at the specified offset. If not, this data will reside in the other buffer, so the pending DMA request to fill the other buffer is waited for. A new DMA request is issued to re-fill the *current* buffer, and the buffer pointer is switched. Now that the required data is ready, *elem_size* bytes are copied from the *current* buffer and returned. To avoid stack corruption, the compiler must ensure that all pending DMA operations are completed before returning from a sieve function which uses DMA streams.

3.2 Pre-fetching Global Variables

Consider the global variable `Size` used to control the **`for`** loops in Listing 1. Since `Size` is in main memory, its value must be fetched using a software cache read. Although this is more efficient than simply fetching `Size` by DMA, `Size` is accessed

[5] For readability, we use C++ template notation to describe streams. Note that streams are generated in the Sieve compiler intermediate representation, and are not exposed to the programmer as our example suggests.

many times which results in frequent cache reads. The sieve semantics allow us to optimise further by *hoisting* the read from `Size` to the start of the sieve block. This is safe even though `Size` could be modified by the call to `rNormalised`: because `rNormalised` is a sieve function, any potential write to `Size` would be delayed.

Listing 2 illustrates this optimisation: the first executable statement in the sieve block copies the value of `Size` to a local variable `LocalSize`, and all reads from `Size` in the sieve block are changed to read from `LocalSize`.

3.3 Combining Delayed Writes

A write to global memory occurring in a sieve scope causes a call to a `delayedStore` function in the SPU run-time system. As discussed in §2.2, this function appends a side-effect node of the form (*PPU address*, *size*, *data*) to a local queue, streaming the local queue to PPE memory when full. A sieve scope with many side-effects will cause the SPE side-effect queues to fill up quickly, resulting in frequent DMA transfers to stream side-effects to main memory. In addition, at the end of the sieve block, the PPE must commit this large number of side-effects to memory individually.

If these side-effects are the result of small delayed writes (*e.g.* 4 bytes or less) then the 8 byte *PPE address* + *size* overhead associated with each side-effect may be the main reason why the local side-effect queue fills up. In many practical examples, delayed writes are to contiguous memory locations, *e.g.* elements of an array in global memory. Consider the delayed store to `Forces[i]` in the molecular dynamics code of Listing 1. It is clear that each iteration of the outer loop will write to an element of `Forces`. Since the loop uses stride 1 access, these writes are contiguous.

The Sieve-Cell system uses a run-time optimisation to take advantage of commonly occurring contiguous delayed writes. Suppose the `delayedStore` function is called with the side-effect node $(addr_n, size_n, data_n)$, and that the last side-effect in the queue is $(addr_{n-1}, size_{n-1}, data_{n-1})$. The run-time checks whether $addr_n = addr_{n-1} + size_{n-1}$. If this is the case then the new side-effect can be handled by replacing the last side-effect node with $(addr_{n-1}, size_{n-1} + size_n, data_{n-1}{+}{+}data_n)$, where ++ denotes concatenation of bits. Otherwise, the new side-effect node is added to the queue as usual. The contiguity check bears a (small) run-time overhead, and while effective when delayed stores *are* contiguous, may be onerous when delayed stores are fragmented.

We could extend the check to eliminate redundant side-effects by spotting cases where multiple stores are made to the same memory location; we have not found this to be a useful optimisation for practical examples.

4 Experimental Evaluation

4.1 Experimental Setup

We present experimental data for single-precision benchmark programs implemented in standard C++, Sieve C++, and OpenMP C++. The Sieve versions of the programs were developed from serial base codes by the addition of Sieve annotations, and elimination

of global variable updates within kernels. In all cases, a small number of preprocessor macros were then sufficient to allow OpenMP, Sieve, and serial code to coexist in one set of files. We also present results for standard and Sieve C++ versions of three further programs which, due to limitations of the (Alpha version of) XL C++, we were not able to implement using OpenMP; these benchmarks are marked † in the following list:

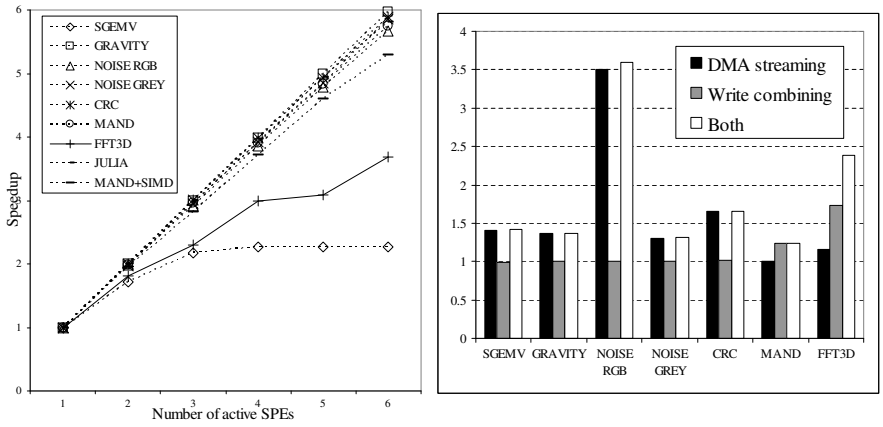
SGEMV: Matrix-vector multiply (8192×4096)
GRAVITY: An N-body molecular dynamics simulation of 8192 particles
NOISE RGB: Noise reduction filter applied to a 512×512 colour image
NOISE GREY: Noise reduction filter applied to a 512×512 greyscale image
CRC: Cyclic redundancy check on a random 8M (1M=2^{20}) word message
MAND: Calculates a 1024×1024 fragment of the Mandelbrot set
FFT3D: Fast Fourier transform of a complex 128^3 data set†
JULIA: Ray traces a 512×512 3D slice of a 4D quaternion Julia set†
MAND+SIMD: The MAND program optimised using SPU SIMD intrinsics†

Experiments are performed on a Sony PlayStation 3 console (on which only six of the SPEs are available to the programmer), running Fedora Core 7 Linux, with IBM SDK v3.0.0. The OutputC code produced by the Sieve compiler is compiled using `ppu-gcc` and `spu-gcc`, and Sieve results are compared against programs in standard C++ compiled with `ppu-g++` (all GNU compilers are v4.1.1). OpenMP examples are compiled using the IBM XL C/C++ Single-Source compiler (Alpha Edition, v0.9), and compared against serial code compiled with the same compiler, using the `-qnosmp` to specify that OpenMP directives should be ignored. Both serial versions run on the PPU.

Figure 1(a) plots the speedup of each Sieve C++ program relative to a single SPU when data movement optimisations are applied. Figure 1(b) demonstrates the effectiveness of these optimisations for code parallelised over 6 SPEs, showing speedups for each optimisation and for their combination relative to performance without data movement optimisations. Figure 2 shows the speedup of each Sieve/OpenMP application relative to serial code compiled with `ppu-g++`/XL C++. For the Sieve applications, the number of SPUs is indicated above the corresponding bar. For OpenMP, the number indicated is the number of active parallel threads, which ranges from 1 to 7 since the IBM OpenMP implementation supports parallelisation across the SPUs *and* PPU. The rules as to when a thread is spawned on the PPU rather than an SPU are undocumented; the results indicate that the distribution strategy varies between input programs.

4.2 Discussion

Figure 1(a) shows that six of our nine benchmarks scale almost linearly as the number of active SPUs increases, with GRAVITY showing the best scaling. Of the remaining three benchmarks, MAND+SIMD scales slightly worse than MAND: the gain from using vector intrinsics is high for a small number of SPUs, but more active SPUs leads to higher bus traffic due to side-effect transfer. This slows down the SPUs, making the impact of vector instructions less significant. Analysis using an in-house Codeplay profiler attributes the reasonable but sub-linear FFT3D scaling and the poor SGEMV scaling to high bus activity. These benchmarks involve a high rate of data movement in small chunks both to and from SPU local store.



(a) Scaling w.r.t. one SPE.

(b) Effectiveness of Sieve-Cell optimisations.

Fig. 1. Benchmark results for Sieve C++ programs.

Our data movement optimisations provide no performance improvement for JULIA and MAND+SIMD, which are therefore not shown in Figure 1(b). DMA streaming does not apply to these benchmarks (or to MAND) since they do not read data from global memory. Delayed write combining is effective for MAND and FFT3D, which involve frequent delayed stores. MAND+SIMD involves fewer delayed stores than MAND since the former benchmark writes back pixels in vector chunks. For NOISE RGB, delayed write combining has little effect alone but provides an improvement when combined with streaming. This works the other way for FFT3D, where streaming provides a more significant speedup when applied with write combining.

The results of Figure 2 show that automatic parallelisation using Sieve and OpenMP can lead to significant speedups over PPU-only code, and thus commends both systems. The Sieve results are competitive with those for OpenMP, showing better scaling for three benchmarks. Both approaches perform poorly compared with serial SGEMV. For the GRAVITY benchmark, although scaling is good, the performance of Sieve code with six SPUs is roughly equal the performance of serial code; OpenMP code using all seven parallel threads runs 1.3 times faster than the XL C++ serial version. Of the Sieve-only benchmark programs, MAND+SIMD shows excellent scalability, approaching the performance of hand-written parallel Cell code. Nevertheless, our 4-pixel SIMD execution path prevents an ideal quadrupling of performance. A modest 1.6 times speedup with 6 SPUs is achieved for JULIA. As with GRAVITY, Sieve code for FFT3D with 6 SPUs runs at roughly the same speed as the serial `ppu-g++` code. Further investigation of the weaker speedup responses has shown that programs with a high rate of data transfer between SPE and PPE memory are most strongly affected, due to the cost of DMA operations and the attendant overhead of the delayed store function. The final writeback of side-effects by the PPE does *not* make a significant contribution.

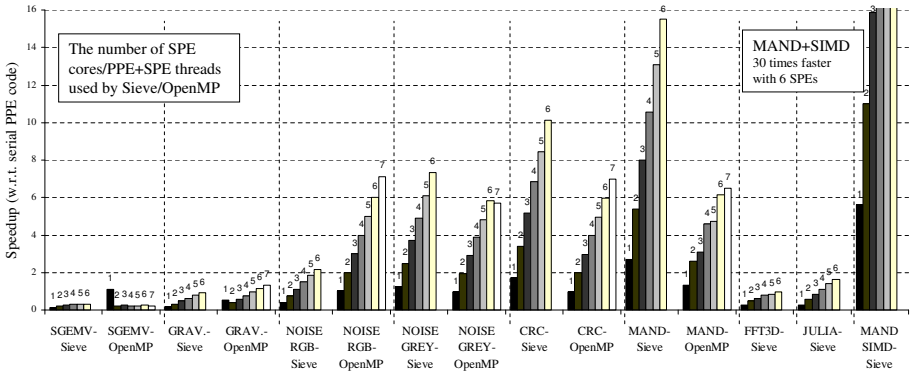


Fig. 2. Scaling for Sieve and OpenMP benchmarks with respect to serial PPE versions.

5 Conclusions and Future Work

We have presented the Sieve-Cell system – an auto-parallelisation system for the Cell BE processor based on the Codeplay Sieve C++ language. We have described compile-time and run-time data movement optimisations, and presented experimental results which show the effectiveness of these optimisations on a number of benchmarks, as well as the scaling of parallel code over multiple SPUs. Our experimental results also show that Sieve is competitive with IBM's OpenMP implementation for Cell.

Future work includes comparing the Sieve-Cell system against Sequoia and CellSs on a number of representative benchmarks, and performance comparisons with other architectures. Similar experiments on double-precision data using PowerXCell8i will also be attempted. The design and implemention of advanced data movement techniques for complex access patterns is also underway, as is the development of an overlay system to allow larger applications to be parallelised over SPUs.

Acknowledgements Thanks to all at Codeplay for their work on the Sieve-Cell system, Mike Houston for providing Sequoia benchmark source code, and the anonymous reviewers for their comments which have helped to improve the paper.

References

1. Bellens, P., Perez, J., Badia, R., Labarta, J.: CellSs: a programming model for the Cell BE architecture. In *Supercomputing'06*, page 86. ACM Press (2006)
2. Donaldson, A. F., Riley, C., Lokhmotov, A., Cook, A.: Auto-parallelisation of Sieve C++ programs. In *Euro-Par'07 Workshops*, pp. 18–27, LNCS 4854. Springer (2008)
3. Fatahalian, K. *et al.*: Sequoia: programming the memory hierarchy. In *Supercomputing'06*, page 83. ACM Press (2006)
4. Hofstee, H. P.: Power efficient processor architecture and the Cell processor. In *HPCA'05*, pp. 258–262. IEEE Computer Society (2005)
5. Lokhmotov, A., Mycroft, A., Richards, A.: Delayed side-effects ease multi-core programming. In *Euro-Par'07*, pp. 629–638, LNCS 4641. Springer (2007)
6. Valiant, L.: A bridging model for parallel computation. *Commun. ACM* 33(8):103-111 (1990)