If arrays are first class values, programmers can specify in a concise manner operations that are to be performed in parallel across whole arrays of data without the need for explicit loop structures. As a result array languages lend themselves to reasonably direct compilation onto parallel hardware.

2 The Machines

Our compilers are targeted at the Cell broadband engine and at Intel multi-core machines using the new AVX instruction sets. These machines pose interesting problems for automatic parallelisation because in the first case a heterogeneous multi-core processor is involved, and the second case the compiler has to wrestle with making the best use of both multi-core and SIMD parallelisation.

2.1 Cell Broadband Engine

The first generation Cell Broadband Engine [13,9] includes a 64-bit RISC PowerPC processor element (PPE), augmented by 8 accelerators: SIMD synergistic processor elements (SPE). The SPEs and PPE are all clocked at 3.2GHz. As shown on figure 1 along with a memory controller, and bus interface controller, the PPE and SPEs are interconnected through an on-chip element interconnect bus (EIB) with 96 bytes/cycle bandwidth. Rambus XDR DRAM memory delivers 12.8 Gb/s per 32-bit memory channel giving a total bandwidth of 25.6 Gb/s. The PPE has 32KB first-level instruction and data caches, and a 512KB second-level cache, and implements IBM's Amazon PowerPC AS ISA. PowerPC support for virtualisation and large page sizes is inherited, enabling the PPE to support multiple operating systems. The PPE provides dual, in-order instruction issue and is fed by two simultaneous hardware threads. The PPE has 32 general purpose registers, and 32 floating-point registers; both are 64-bit. A vector multimedia extension unit (VMX) also provides the PPE with its own set of 32, 128-bit wide, SIMD registers using an Altivec SIMD ISA.

The SPEs are mainly designed for manipulating data. They are independent simple SIMD RISC architectures that are very similar in function to vector processors in which a single instruction operates on multiple data elements [1]. An SPE consists of two independent main components: a Synergistic Processing Unit (SPU) and a Memory Flow Controller (MFC). Each SPU has 128 x 128-bit registers and supports only 16-byte load and store operations and operates on 128-bit vector data types. It also has two parallel execution units and a 256KB

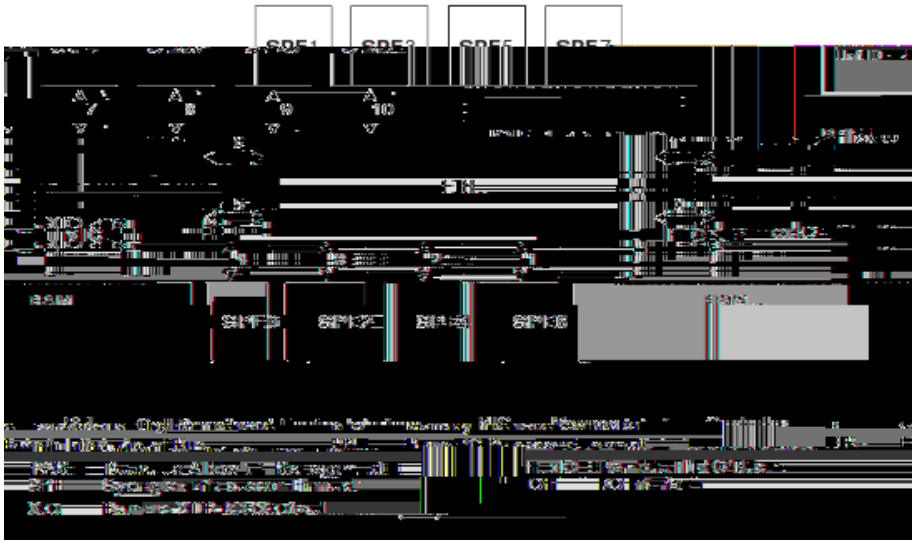

Fig. 1. Architecture of the Cell Broadband Engine.

(Cell SDK) installed. The Cell SDK includes two versions of GCC 4.1.1 targeting the PPE and SPE respectively. Of the Cell's eight SPEs, only 6 are available to us. One is lost due to increased fabrication yields; the other hosts the execution of a hypervisor.

2.2 The Intel Sandybridge

The hardware for the Intel test used an Intel Sandy Bridge[29] with 4 compute cores and one Graphics Processing Unit. Each core was hyperthreaded giving a maximum of 8 simultaneous threads supported in hardware and the clockspeed was 3.1 GHz.. Each core has its own level 1 caches for instruction and data each of 32 kB, these are backed up with 256K of level 2 cache for each core. There is a level 3 cache shared by all caches and synchonised via a high speed inter processor ring bus. This ring bus architecture is similar to that used on the Cell and the Larrabee[25].

From the standpoint of the compiler writer the most signi cant feature of the chip is that it is the rst production machine to run the new AVX instructions[10]. These introduce a number of novel features the most signi - cant of which are:

The introduction of 3 address instructions as a general feature of the architecture. Up until now 3 address instructions on Intel chips had only been used in integer multiplication.

The extending of the XMM registers used in the SSE instructions to 256 bits. The full length registers are now called YMM registers. This greater

length allows, for instance, the parallel addition of 8 single precision oating point values in a single instruction.
The introduction of unaligned multi-media addresses. In the original SSE speci cation memory addresses of 128 bit vectors had to be 16 byte aligned. The withdrawal of this restriction allows more freedom in compiling parallel operations on array slices.

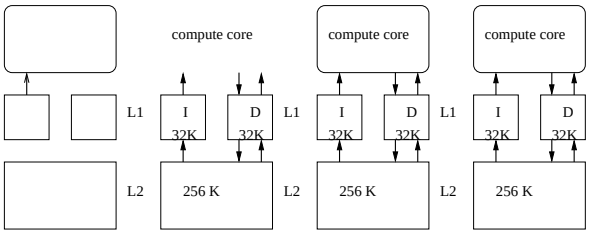

Fig. 2. Architecture of the Sandy Bridge chip.

The machine was running Linux and the C reference tests were compiled with GCC version 4.1.2

3 The Compilers

We will describe the approaches taken by two compilers: an F compiler written in Haskell that targets the IBM Cell, and a Pascal compiler implemented in Java that targets both the IBM Cell and the Intel Sandybridge processor. We will look in particular at the code generation and parallelisation strategies of the two compilers.

3.1 Glasgow Pascal

The Glasgow Vector Pascal compiler is implemented in Java allowing a single executable jar le to be distributed for all platforms. The machine code to be output is selected by a commandline ag. This ag selects one of a library of code generator classes included in the jar le. The optimising code generators

themselves are java classes which are automatically produced by a code generator generator from formal speci cations of the target processor instruction-sets[5].

The compiler will attempt to parallelise array statements across two dimensions. It will attempt to distribute calculations of the rows of the result across di erent cores and attempt to parallelise the column results of each row using SIMD instructions. The degree of parallelism achieved depends both on the width of the SIMD registers available, and on the number of cores available.

For example the executable statement in the following code:

```
var b,d:array[1..n,1..m] of real;
    a:array[1..m] of real;
.....
   b:= 1.5*(d+a);
```

is equivalent on a processor with no parallel hardware to the following loop using the implicit index vector `iota`:

```
 for iota[0]:=1 to n  do
  for iota[1]:=1 to m   do
   b[iota[0] ,iota[1] ]:= 1.5*(d[iota[0],iota[1] ]+a[iota[1]]);
```

but if we were using a machine with 8 element vector registers and 4 cores the translation would produce something equivalent to the following pseudo-code:

```
 parfor p :=0 to 3 do
 for iota[0]:=1 to n step 4 do
  for iota[1]:=1 to (m div 8)*8 step 8 do
   b[iota[0]+p,iota[1]..iota[1]+7]:=
    1.5*(d[iota[0]+p,iota[1]..iota[1]+7]+a[iota[1]..iota[1]+7]);
  for iota[1]:=1+(m div 8)*8  to m   do
   b[iota[0] ,iota[1] ]:= 1.5*(d[iota[0],iota[1] ]+a[iota[1]]);
```

Although this above is pseudo-code, it captures the basic parallelisation mechanism which allocates di erent rows of the problem to threads and then uses SIMD parallelism on the inner loop. Note the residual sequential code to handle rows that are not a multiple of the vector length.

Parallelisation is entirely automatic, no pragmas or directives are needed in the source code. Command line ags indicate what hardware resources are available to achieve parallelisation. So to compile a programme `nbody.pas` to Sandybridge code one would issue the shell command:

```
vpc nbody -cpuAVX32 -cores4 -opt3
```

to specify the source le, the processor, the number of available cores and the optimisation level to use. The

in the s vecor n1ody.pas

When compiling for the Cell we use the same compiler as for the Sandy Bridge but use a di erent code generator class. In this case we specify that the code generator has only one core available to it, but the core has very long SIMD registers capable of holding 1024 oating point numbers. The long SIMD registers are emulated at run time by a Virtual SIMD Machine(VSM) that runs in parallel on the SPEs. The code generator plants code to initialise the virtual machine on the SPEs at programme startup. It then generates PowerPC assembly instructions that correspond to the sequential source code excluding array expressions. Array expressions are translated into VSM opcodes using the basic pattern shown earlier. The virtual SIMD instructions are implemented by the Power PC writing virtual machine opcodes down hardware FIFOs to parallel interpreters runing on the SPEs. The scheduling of operations on the SPEs and the data transfers to them is thus done using the standard register allocation mechanism of the code generator.

3.2 E*]*

The diagram in gure 3 presents an overview of the executable components of the toolchain associated with the E*]* compiler [20] at an operational level. E*]*, the O oad C++ compiler, and the GNU toolchain all run under Cygwin on Windows and produce a 32-bit ELF executable suitable for execution on the PS3 under Linux. As shown, the SPU object les are embedded within a PPU object le using the GNU ppu-embedspu tool. The compilation stages performed by the GNU tools are fully con gurable; and all such components are potentially interchangeable.

The E*]* compiler accepts individual source programs written in the `F' programming language, before translating them into the extended C++ dialect, O oad C++ [6]. The purpose of this transformation is to extract and partition suitable `F' array expressions into multiple o oad blocks, later compiled by the O oad compiler for parallel evaluation. Of course E*]* must also convert the base language from `F'. The output language, O oad C++, statically assigns each pointer with an integer depth, 0 or 1, corresponding to a target in main memory or in local store; *outer* or *inner*. This aspect transforms many `F' procedures into template functions, parameterised by this *o oad depth*.

The output by the E*]* compiler is in fact con gurable using embedded `C' preprocessor macros embedded. With the appropriate (default) macros set, the extended O oad C++ portions of the generated code may either be enabled, facilitating parallel execution, or disabled, providing serial C++ code may be compiled by any C++ compiler. The existence of this serial reference has proced highly useful for debugging, testing, and also performance pro ling.

The O oad compiler may then be provided with extended C++ output from the E*]* compiler as input, and may need access to a Fortran runtime library compatible with the PPU, the SPU, or both. E*]* has developed a templated C++ class, ArrayT, parameterised by o oad depth, and providing a cross-vendor ABI-compatible array interface to all Fortran runtime libraries, using the low-level Chasm Interoperability Tools [22].

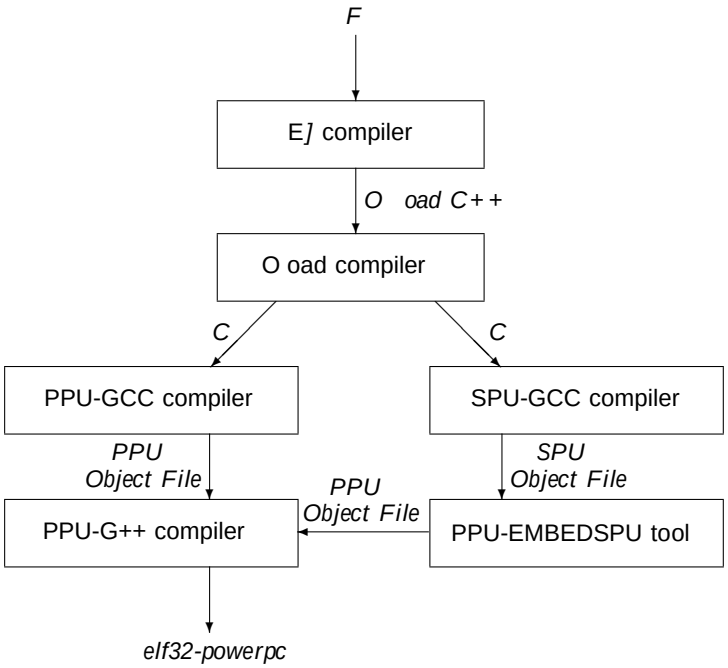

Fig. 3. The E] compiler and the system at-large.

Sony's Multicore Application Runtime System (MARS) libraries are used to implement the threading support required by the O oad runtime system.

The O oad compiler can also be invoked with the *-nomake* switch, in which case the generated source and make les are not deleted; and the GNU make tool is not applied. This option is essential when di erent tools, or versions other than the defaults, are required. For example, in the con guration shown in gure 3, the

4 An example application

We will take as an example of how these compilers work a benchmark programme for N-body simulation. We will show how the use of array operations allows automatic parallelisation of the code to di erent target architectures without the need for any pragmas or process directives. We will also present measurements of the performance gains that we have obtained with this approach. The N-body

Table 2. More detailed performance of Pascal on the Cell

N-body Problem	Performance (seconds) per Iteration				
	Pascal				C
Size	PPE	1 SPE	2 SPEs	4 SPEs	PPE
1K	0.381	0.105	0.065	0.048	0.045
4K	4.852	1.387	0.782	0.470	0.771
8K	20.35	5.715	3.334	2.056	3.232
16K	100.2	22.27	13.24	8.086	16.52

Tanikawa and collaborators [26] report a similar experiment the use of the AVX instructions to accelerate a version of the N-body problem, but this version is not directly comparable with the SICSA work both because it uses a more sophisticated Hermite technique and because their kernels were hand coded in assembler. The version reported by Sampson[23] is intermediate between a high level language and a low level one, since it required the explicit use of machine speci c SSE intrinsics. All other implementations reported in Table 1 use machine independent high level language source codes. Note that the parallel Pascal on the Cell gives only roughly the same speed as sequential C in table 1, but for $N > 1024$, Gdura[19] reported appreciable acceleration over C.

The N body problem is inherently of order N^2 in the number of planets since it is necessary to compute the gravitational force between each pair of bodies.

force matrix has to be computed, meaning that parallel code has to do more arithmetic than sequential code. The Pascal version of the advance function is

```
type coord = record pos: array[1..3] of realt;end;
procedure MAadvance(dt : realt);
var i,j :integer;
    dv:array[1..n,1..1] of coord;
begin
    dv:=computevelocitychange(iota[0],dt);
    for i:= 1 to N do
        for j:= 1 to 3 do v^[j,i]:=v^[j,i]+ dv[i,1].pos[j];
(*! Finally update positions. *)
    x^ := x^ + v^ *dt;
end;
```

The array dv is a column vector whose elements are coordinates. The assignment to dv in the rst line of the function computes in parallel the *v* values for all planets. These are then used to update the velocities. The nal line computes the changes in position for all bodies, again in parallel. The compiled code uses a mixture of SIMD and multi-core parallelism. The E*]* and Fortran version of the algorithm uses a more sophisticated decomposition based on [15] but it uses a Fortran elemental function mapped over an array in a similar manner.

```
elemental function calc_accel_p(pchunk) result(accel)
  type(pchunk2d), intent(in) :: pchunk
  type(accel_chunk)          :: accel
  real(kind=ki) :: dx, dy, dz, distSqr, distSixth, invDistCubed, s
  integer :: i, j
  accel%avec3 = vec3(0.0_ki, 0.0_ki, 0.0_ki)
  do i=1,size(pchunk%ivec4)
    do j=1,size(pchunk%jvec4)
      dx = pchunk%ivec4(i)%x - pchunk%jvec4(j)%x
      dy = pchunk%ivec4(i)%y - pchunk%jvec4(j)%y
      dz = pchunk%ivec4(i)%z - pchunk%jvec4(j)%z
      distSqr      = dx*dx + dy*dy + dz*dz + EPS
      distSixth    = distSqr * distSqr * distSqr
      invDistCubed = 1.0_ki / sqrt(distSixth)
      s = pchunk%jvec4(j)%w * invDistCubed
      accel%avec3(i)%x = accel%avec3(i)%x - dx * s
      accel%avec3(i)%y = accel%avec3(i)%y - dy * s
      accel%avec3(i)%z = accel%avec3(i)%z - dz * s
    end do
  end do
end function calc_accel_p
```

If we consider the general complexity of this problem under parallelism, one component of the execution time should shrink as the number of processors increases. During each round of the simulation, the program has to accumulate the gravitational forces imposed on each body by all other bodies. Since these calculations are independent, they can in principle be done using di erent processors in parallel. If p is the number of processors, this stage should have a cost $\frac{N^2}{p}$, for some constant . After this calculation has been done, all of the processors would have to ensure that all other processors have access to the same updated data on planetary positions. For a uni-processor this is unproblematic there is only a single state vector in memory. For multi-processors, however, depending on their design, this communications phase can be an appreciable overhead. If the communications is done naively, the data-transfer cost is $\underline{p^2 N}$

5 Conclusions

We have demonstrated that array versions of two classic imperative languages Fortran and Pascal can be a suitable tool when targeting modern multi-core processors. The addition of array mapping operations gives imperative languages the same advantages of concise notation that functional languages gain through list mapping. Such maps lend themselves well to expressing parallel operations. Because of the inherently simpler compilation model of imperative languages it is possible to translate simple data parallel expressions into fast code. On homogenous multi-core machines the array language gave performance markedly superior to other languages with explicit support for parallelism. Only by the use of SIMD intrinsics and explicit thread libraries was C++ able to give comparable results. On the hetrogenous Cell architecture, we have shown two distinct approaches to implementing array parallel operations. Whilst each of these

9. H. Peter Hofstee. Power E cient Processor Architecture and The Cell Processor. In *HPCA*, pages 258 262. IEEE Computer Society, 2005.
10. Intel. *Combined Volume Set of Intel 64 and IA-32 Architectures Software Developer's Manuals*, 2011.
11. ISO. *Pascal ISO 7185*, 1990.
12. K. Iverson. *A programming language*. Wiley, New York, 1966.
13. J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4.5):589 604, July 2005.
14. P. Keir. All-pairs n-body in Fortran for CellBE . SICSA Multi-core Challenge Phase II Workshop, May 2011.
15. Mark Harris Lars Nyland and Jan Prins. Fast n-body simulation with cuda. In Hubert Nguyen, editor, *GPU Gems 3*, pages 677 694. Addison-Wesley Professional, 2007.
16. Hans-Wolfgang Loidl. A C# implementation of the n-body problem . SICSA Multi-core Challenge Phase II Workshop, May 2011.
17. Iain McGinniss. Naive approaches to n-body parallelism using Google Go . SICSA Multi-core Challenge Phase II Workshop, May 2011.
18. M Metcalf and J Reid. *The F Programming Language*. Oxford Univesity Press, 1996.
19. Y. Gdura P. Cockshott. Vector Pascal implementations running on Nehalem and Cell processors . SICSA Multi-core Challenge Phase II Workshop, May 2011.
20. P. Cockshott P. Keir and A. Richards. Mainstream parallel array programming on cell. In *Proceedings of the 5th Workshop on Highly Parallel Processing on a Chip (HPPC 11)*, 2011.
21. R.H. Perrott, D. Crookes, P. Milligan, and W.R.M. Purdy. A compiler for an array and vector processing language.