CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2014; **26**:952–971
Published online 7 August 2013 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3078

SPECIAL ISSUE PAPER

SAC/C formulations of the all-pairs N-body problem and their performance on SMPs and GPGPUs

Artjoms Šinkarovs [1], Sven-Bodo Scholz [1,*,†], Robert Bernecky [2], Roeland Douma [3] and Clemens Grelck [3]

[1] *Heriot-Watt University, Edinburgh, UK*
[2] *Snake Island Research Inc, Toronto, Canada*
[3] *University of Amsterdam, Netherlands*

SUMMARY

This paper describes our experience in implementing the classical N-body algorithm in SAC and analysing the runtime performance achieved on three different machines: a dual-processor 8-core Dell PowerEdge 2950 (a Beowulf cluster node, the reference machine), a quad-core hyper-threaded Intel Core-i7 based system equipped with an NVidia GTX-480 graphics accelerator and an Oracle Sparc T4-4 server with a total of 256 hardware threads. We contrast our findings with those resulting from the reference C code and a few variants of it that employ OpenMP pragmas as well as explicit vectorisation. Our experiments demonstrate that the SAC implementation successfully combines a high level of abstraction, very close to the mathematical specification, with very competitive runtimes. In fact, SAC matches or outperforms the hand-vectorised and hand-parallelised C codes on all three systems under investigation without the need for any source code modification. Furthermore, only SAC is able to effectively harness the advanced compute power of the graphics accelerator, again by mere recompilation of the same source code. Our results illustrate the benefits that SAC provides to application programmers in terms of coding productivity, source code, and performance portability among different machine architectures, as well as long-term maintainability in evolving hardware environments. Copyright © 2013 John Wiley & Sons, Ltd.

Received 19 February 2012; Revised 24 May 2013; Accepted 24 May 2013

KEY WORDS: single assignment C; data parallelism; functional programming; high productivity; auto-parallelisation; vectorisation; high performance; graphics cards

1. INTRODUCTION

The SICSA MultiCore Challenge is a long term initiative that aims at evaluating the state of the art in programming language support for multi-core systems. Since 2010, two programming challenges have been identified; researchers have been invited to contrast programming languages of their choice against a given reference implementation on a given reference system. For details on the SICSA MultiCore Challenge see [1].

This paper focuses on the second SICSA challenge, themed around the *N-body problem*. The N-body problem is that of predicting the motion of a group of celestial objects, interacting with each other gravitationally. As formulated in 'Philosophiae Naturalis Principia Mathematica'

*Correspondence to: Sven-Bodo Scholz, School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, EH14 4AS, UK.

†E-mail: S.Scholz@hw.ac.uk

by Sir Isaac Newton, the N-body problem can be described as a system of the following differential equations:

$$\forall j \in \{1, 2, \ldots, n\} : m_j \ddot{q_j} = G \sum_{k \neq j}^{n} \frac{m_j m_k (q_k - q_j)}{\left|q_k - q_j\right|^3}, \tag{1}$$

where $m_1, m_2, \ldots m_n$ are point masses of the planets, $q_1, q_2, \ldots, q_n$ are 3D vector functions of time variable t describing the positions of the point masses and G is the universal gravitational constant. It is assumed that initial positions $q_i(0)$ and velocities $\dot{q_i}(0)$ are defined, and $q_i(0) \neq q_j(0)$ if $i \neq j$.

The classical way to approximate $q_i(t)$ for all i and a given interval for t is to use numerical integration, separating the second-order differential equation into two first-order equations and to apply the Euler method of order one. This leads to the following equations, where k indicates the discretised time, whereas v_i and a_i denote $\dot{q_i}$ (velocity) and $\ddot{q_i}$ (acceleration), respectively.

$$\overset{k+1}{q_i} = \overset{k}{q_i} + \overset{k+1}{v_i}\, dt \tag{2}$$

$$\overset{k+1}{v_i} = \overset{k}{v_i} + \overset{k+1}{a_i}\, dt \tag{3}$$

$$\overset{k+1}{a_i} = \sum_{j \neq i}^{n} \frac{m_j \left(\overset{k}{q_j} - \overset{k}{q_i}\right)}{\left|\overset{k}{q_j} - \overset{k}{q_i}\right|^3} \tag{4}$$

This discretised form straightforwardly leads to an algorithm of complexity $O(N^2)$, which is often referred to as *all-pairs N-body simulation*. In practice, however, this algorithm is rarely used as such because its quadratic complexity makes it prohibitively expensive as the number of bodies increases. More sophisticated techniques, such as Barnes–Hut [2], are more popular in practice. They are based on the observation that distant objects have negligible gravitational effect on each other and, hence, can be accounted for in a cumulative fashion. This brings down the complexity to $O(N \log(N))$. However, as part of the Barnes–Hut algorithm, the effect of all bodies within a given local scope are treated individually, typically by applying the simple all-pairs algorithm, as explained earlier. As a consequence, the all-pairs algorithm itself can nevertheless be seen as a suitable object for analysis. The ability to program and parallelise this algorithm effectively constitutes an important prerequisite for any effective Barnes–Hut implementation.

In this paper, we look at a SAC implementation of the all-pairs algorithm. SAC (Single Assignment C) is a purely functional programming language with a C-like syntax whose most prominent feature is genuine support for truly multidimensional and truly functional (state-free) arrays [3,4]. SAC aims at combining high programmer productivity with high performance across a range of multi-core architectures, a goal that aligns very well with the SICSA MultiCore Challenge.

SAC achieves high programmer productivity through extreme abstraction. In fact, SAC programs often remain very close to abstract algorithmic or even mathematical specifications. For instance, all memory management for aggregate data structures such as arrays is completely automatic. SAC programs permit a high level of code reuse across applications and across a range of parallel target platforms from multi-socket multi-core systems [5] to GPGPU-style graphics accelerators [6]. Fully automatic parallelisation, regardless of the chosen target architecture, is a fundamental characteristic of SAC. Thus, in contrast to the pragmas of OpenMP [7,8] or the explicit kernels of CUDA [9] and OpenCL [10,11] SAC programs are 100% target architecture agnostic.

At first glance, the design principles of SAC seem to be fundamentally at odds with the goal of high performance: highly efficient programs typically avoid unnecessary abstraction. They rather aim at minimising the materialisation of redundant computations and redundant data structures that often result from high-level program specifications. Furthermore, efficient programs are usually finely tuned to the executing machinery, in particular when it comes to targeting multi-core and other parallel systems.

DOI: 10.1002/cpe

SAC attacks this seemingly insuperable divide between high productivity and high performance through aggressive compiler optimisation exploiting the purely functional, side-effect-free semantics. Advanced code transformations, such as *with-loop folding* [12, 13], *with-loop fusion* [14] and *with-loop scalarisation* [15], systematically transform SAC programs from a highly problem-oriented representation into a radically different machine-oriented representation. From that representation, dedicated code generators derive tailor-made code for a variety of hardware architectures.

In this paper, we put the SAC approach to the test by looking at the all-pairs N-body algorithm. We derive an algorithmic specification as close as possible to the underlying mathematics and evaluate how it performed on three different contemporary multi-core machines featuring four different processor architectures: a two-processor 8-core Intel Xeon based SMP server (a single node of the Beowulf cluster), an Intel Core-i7 based multi-core system equipped with an NVidia GTX-480 graphics accelerator and an Oracle Sparc SuperCluster T4-4 server with 4 processors, 32 cores and 256 hardware threads in total. Given the embarrassingly parallel nature of the algorithm, we contrast the SAC run times with those of the given reference implementation in C. We also look at variants of the C code that aim at improving its multi-core performance: we investigate multi-core scalability by OpenMP annotations, and we explore the potential of explicit vectorisation.

The structure of our paper follows the guidelines for this special issue. The next section gives a brief introduction into the programming languages under consideration. It summarises SAC, the vector extensions we use to enforce vectorisation of the reference code and OpenMP, which we employ to hand-parallelise the reference implementation. Section 3 gives a more detailed account of the three systems we use for our experiments. In Section 4, we describe the various implementations that form the basis of our experiments. Section 5 presents the experiments themselves and their analysis. We primarily concentrate on wall-clock times achieved for the given input data for easier cross-paper comparisons within this special issue. However, we also look at scalability effects for increased problem sizes. We draw conclusions in Section 6.

2. LANGUAGES AND LIBRARIES

2.1. SAC — *Single Assignment C*

Single Assignment C – SAC for short – is an array language that looks like C, feels like C, but nonetheless is purely functional with execution driven by the principle of context-free substitution and data represented by immutable values. The main idea of SAC is to provide a framework to operate with arrays. Hence, types in SAC represent arrays with potentially unknown ranks (number of axes or dimensions) and shapes (extents along axes/dimensions). To keep the language functional, SAC rules out expressions with side-effects, undefined behaviour, pointers, etc. This allows the compiler to use implicit memory management and to make decisions about parallel execution of certain code parts without requiring programmer-specified annotations. In the following we merely introduce some key concepts of SAC to facilitate understanding of code examples. For more information the interested reader is referred to [3, 16].

As we mentioned before, all types in SAC are array types and the size of an array is a type attribute, rather than a variable attribute. For example, to declare an array of five integers, one would write the following declaration:

```
int[5] A;
```

In SAC, array types can also be specified without a static size but with a given dimension, for example, a 2D array of double of arbitrary size can be defined as following:

```
double[.,.] A;
```

Furthermore, SAC supports rank-generic programming through types that leave even the number of dimensions of an array open:

```
float[*] A;
```

DOI: 10.1002/cpe

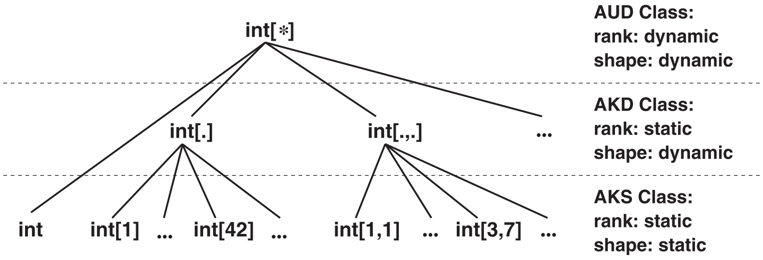

Figure 1. Type hierarchy of SAC.

For each base type, SAC features a hierarchy of array types, as illustrated in Figure 1, and supports overloading of functions accordingly. Built-in primitives allow programs to query for the ranks, shapes and elements of generic arrays.

One of the key language constructs in SAC is the with-loop. It is a data-parallel array comprehension construct, by which the programmer specifies how index sets are to be mapped into element values. Depending on the with-loop used, the computed values are subsequently either laminated to form an array or they are folded into a single value. The with-loop is used in SAC to express potentially parallel operations. As an example, consider generating a matrix A of shape M by N, where all elements at index position i,j are computed through some function foo(i,j). In SAC, this can be achieved by the following:

```
A = with {
      ([0,0] <= [i,j] <= [M,N]): foo (i,j);
    } : genarray ([M,N], 0);
```

Note that the zero in the end of this specification denotes a default element that would be used instead of foo(i,j) if the index range would not be fully covered.

SAC does not provide built-in arithmetic array operations, but the standard library defines a plethora of array operations making use of rank-generic programming, shapely overloading and the previous with-loop construct. For example, it is possible to add two objects of the same type and shape, in which case the operation is applied element-wise. Arithmetic operations of different types are also supported, for example, it is possible to multiply an arbitrarily shaped array by a scalar or vice versa. The usual reduction operations are likewise provided.

A thorough introduction to programming in SAC can, for instance, be found in [17].

2.2. *Portable vectorisation in gcc*

Despite the omnipresence of SIMD accelerators in almost all mainstream processors, dedicated use of these through a C program proves surprisingly difficult. Most modern compilers are aware of the SIMD capabilities of a target architecture, and most of the mainstream C compilers do come with some auto-vectorisation capabilities [18–20]. This would be an ideal solution, as it requires minimum effort by the user. Unfortunately, if the auto-vectoriser fails, the user cannot easily help it along. The alternative is to either use intrinsic operations or to use inline assembly, both of which make the code non-portable. So the question is: How can we help the compiler while retaining portability?

As a potential solution to the problem, we have implemented a set of OpenCL-compatible vector operations in the context of GNU gcc compiler. We do not provide a general introduction to the framework in this paper, but we will mention the key points of it. Details can be found in [21].

To force a C compiler to generate SIMD operations, one has to use operations on vector types. To declare a vector type, the notion of attributes is used. Consider the following example:

```
int __attribute__ ((vector_size (16))) var;
```

The `int` type specifies the base type of the vector, and the attribute specifies the length of the vector type, measured in bytes. In the aforementioned declaration, given that `int` is a 32-bit type, we define a vector of four integers.

Language-wise, a vector type does not differ from a scalar type: one can have an array of pointers of vector types, arrays of vector types, typecasting and so on. Most of the operations applicable to scalar types can be used on vector types. Vector types allow indexing of the inner elements by using array indexing notation, support arithmetic operations written as scalar/vector or vector/scalar, and support vector-specific operations such as shuffle.

2.3. OpenMP

The OpenMP API provides a set of annotations, recognised by C/C++ and Fortran, which allow a programmer to mark a code to be executed in parallel, using threads. The main advantage of this approach is its simplicity of use. A programmer does not have to deal with any complications introduced by thread programming, such as mutexes, locking and potential deadlocks. The only thing one has to provide is an annotation and the scope of application. As OpenMP defines a standard only, a programmer is free to choose which implementation to use.

A common scenario of using OpenMP is loop-parallelization, which could be achieved by using the following annotation:

```
#pragma omp parallel for shared (a,b) private (i)
for (i = 0; i < N; i++)
  a[i] = foo (b[i]);
```

In the example given, we inform the compiler that the loop should be executed in parallel, that the variables a, b are shared between the threads and that the variable i is private to the individual threads. Being the iteration variable, i for any given thread would range over a slice from the overall range. For more details on OpenMP, we refer the interested reader to other sources such as [7, 22].

3. EXPERIMENTAL INFRASTRUCTURE

3.1. Machines and operating systems

We run our experiments on three different machines exposing four distinct computer architectures. The first system is the reference machine for the SICSA MultiCore Challenge, in our case a single node of the Beowulf cluster at Heriot-Watt University: an 8-core Intel Xeon based Dell PowerEdge 2950. Next, we employ an Intel Core-i7-based system equipped with an NVidia GTX-480 GPU, where we investigate the performance both by using solely the CPU and by using the CPU and the GPU together. Finally, we look at a 256-way hardware threaded Sparc SuperCluster T4-4 system.

3.1.1. Beowulf cluster: Dell PowerEdge 2950. The Beowulf cluster at Heriot-Watt University is the reference machine for the SICSA MultiCore Challenge. It consists of 64 Dell PowerEdge 2950 nodes. Each node has two quad-core Intel Xeon E5504 processors with 4 MB Intel Smart Cache running at 2 GHz. The operating system is CentOS 5.8. Because the tool chains under consideration in this paper do not support cluster architectures for the time being, we restrict ourselves to a single node.

On this system, we demonstrate the fully automatic parallelisation technology of the SaC compiler [5] and compare them with OpenMP explicit parallelisation directives. Furthermore, we investigate the impact of explicit vectorisation instructions as provided by the gcc version we used.

3.1.2. Intel Core-i7. Our second benchmark system is representative of high-end consumer-level hardware. It is equipped with a quad-core Intel Core-i7 930 processor. Each of the four cores is two-way hyper-threaded. The system is clocked at 2.8 GHz and comes with 64 KB L1 cache and an 8 MB L2 cache. It runs a 32-bit Ubuntu 10.04 operating system.

DOI: 10.1002/cpe

We effectively repeat our experiments from the Beowulf cluster node. The differences between the server hardware in the Dell PowerEdge 2950 (i.e. two independent processors and eight fully-fledged cores) and the consumer hardware here (i.e. only a single processor, hyper-threaded cores) lead to interesting insights.

3.1.3. Intel Core-i7 + NVidia GTX-480. The aforementioned Core-i7 system is further equipped with an NVidia GTX-480 GPGPU. This graphics accelerator features 480 (simple) cores running at 1.4 GHz. The peak memory bandwidth between the host and the GPU is 8 GB/sec, whereas the bandwidth between the GPU cores and GPU memory is 177.4 GB/sec. The graphics card comes with 1.5 GB of memory and 16 KB cache per 32 cores. We use CUDA 4.0.

Combining a multi-core CPU with a high-end graphics accelerator is a typical hardware scenario these days, from consumer-level systems to high performance computing installations [23]. On this system, we evaluate the automatic CUDA code generation capabilities of the SAC compiler [6].

3.1.4. Oracle Sparc SuperCluster T4-4. Our third benchmark system is an Oracle Sparc T4-4 server [24]. The Sparc T4 processor is the latest generation of a series of highly parallel, throughput-oriented processors developed by SUN MicroSystems, now Oracle, under the code name Niagara. The T4-4 server is a cache-coherent SMP system with four Sparc T4 processors running at 2.85 GHz. Each T4 processor consists of eight V9 cores, each of which is eight-way hardware threaded. Each V9 core is equipped with two out-of-order integer execution pipelines, one floating point unit, branch prediction and hardware data prefetch. Furthermore, each core has 16 KB of data and instruction cache each and 128 KB private unified L2 cache. Each processor has a 4 MB eight-banked 16-way associative L3 cache shared across all eight cores. The operating system is Solaris 10 (Beta).

The T4-4 server provides a total of 256 hardware threads and thus exactly the same level of hardware concurrency as the entire Beowulf cluster. We use this system specifically to validate the automatic parallelisation feature of the SAC compiler when confronted with levels of concurrency that exceed those of the more main stream x86-based systems by way more than an order of magnitude.

3.2. Compilers

Throughout the experiments, we use the SAC compiler sac2c v1.00-beta (revision 17726) and the SAC standard library (revision 1558). On the two x86-based systems we use gcc 4.7 checked-out from the repository (rev 183874) both for compiling the various C sources and as backend code generator for sac2c. This version of gcc came with the GNU implementation of OpenMP and explicit vectorisation instructions. For harnessing the GTX-480 GPGPU, we use CUDA 4.0 and the NVidia C compiler nvcc as backend code generator for sac2c. Finally, we use the Oracle/Sun Studio C compiler 12.3 on the Sparc T4-4 server again both as a backend code generator for sac2c and for compiling the reference C implementation of the N-body problem.

4. IMPLEMENTATION OF ALGORITHMS

In this section, we present all implementations used in this study. We also manifest important aspects of each implementation, as they are relevant to performance evaluation. We begin with the SAC implementation because it is very similar to the mathematical specification as presented in the formulae (2) – (4). Afterwards, we discuss several C-based parallel implementations.

4.1. SAC

The SICSA N-body challenge simulates the movement of 1024 bodies in 3D space over a certain period. Each body is characterised by its mass, treated as a single point, and by a directed velocity vector. As explained in the introduction, we look at the all-pairs implementation of the N-body problem. At each time step, each body experiences gravitational acceleration from all other bodies (4), which affects its velocity and its direction (3), and, consequently, its next position (2).

DOI: 10.1002/cpe

```
double
sicsaL2Norm( double[.] x)
{
  return sqrt(sum (x^2) + 0.01);
}

double[3]
acceleration( double[3] pos1, double[3] pos2, double mass)
{
  return (pos2 - pos1) * mass / (sicsaL2Norm(pos2 - pos1) ^ 3);
}

double[3]
acceleration( double[3] pos, double[.,.] positions, double[.] masses)
{
  acc = [0.0, 0.0, 0.0];

  for (i = 0; i < length(masses); i++) {
    acc += acceleration (pos, positions[i], masses[i]);
  }

  return acc;
}

double[.,.], double[.,.]
advance( double [.,.] positions, double [.,.] velocities,
         double [.] masses, double dt)
{
  accelerations = with {
                    ([0] <= [i] < shape(masses)) :
                      acceleration (positions[i], positions, masses);
                  }: genarray (shape(masses), [0.0, 0.0, 0.0]);

  velocities += accelerations * dt;
  positions  += velocities * dt;

  return (positions, velocities);
}
```

Figure 2. Complete N-body code in SAC.

In SAC, we can straightforwardly turn the underlying physics into program code. Figure 2 shows the complete SAC code for computing one time step. The function `advance` takes four arguments: a vector of 3D positions, a vector of 3D velocities, a vector of masses and a time interval `dt`. From these arguments, it computes new positions and new velocities for the time after the given time interval has elapsed. To do so, we first compute for each body `i`, the accumulated effect of the acceleration as a result from the gravitational effect of all other bodies. This effect is captured as a vector of accelerations. By using this vector, we conveniently specify the effect of those accelerations on the bodies' velocities by means of an element-wise multiplication with `dt` and an element-wise addition to the previous velocities. In the same fashion, we compute the new positions as an effect of the new velocities over time.

The function `acceleration` computes the acceleration that results from the gravitational forces of either a single body or multiple bodies to a given body. It is implemented by overloading two function specifications, one to compute the acceleration due to an individual body (first definition in Figure 2) and a second one to add up the effects of several individual bodies (second definition in Figure 2). The actual computation of the gravitational force is almost identical to any textbook definition. Adjustments, such as the use of a slightly diffuse `L2 norm`, which ensures a non-zero minimum distance, are consequences of the problem formulation provided by the SICSA MultiCore Challenge.

4.2. Reference C implementation

The first C implementation we consider is the reference implementation of the second SICSA MultiCore Challenge, which originates from the Computer Language Benchmarks Game [25]. This code is used as a base line for all comparisons.We show the `advance` function in Figure 3.

We observe several important differences between the C implementation and the SAC solution in Figure 2. First of all, we see that the C implementation uses a vector of records as a representation of the bodies, instead of using individual vectors for positions, velocities and masses. We can also observe that there is no dynamic memory management whatsoever. By the way the code is specified, static memory reuse between time steps is guaranteed. In contrast, the SAC implementation does not enforce this at all: The decisions as to whether or not the vectors for the positions, velocity and masses can be reused, are inferred by the compiler, or decided dynamically by the runtime system.

```
1   struct planet
2   {
3     double x, y, z;
4     double vx, vy, vz;
5     double mass;
6   };
7
8   struct planet bodies[N];
9
10  void advance( int nbodies, struct planet *bodies, double dt)
11  {
12    int i, j;
13    for (i = 0; i < N; i++)
14      {
15        struct planet *b1 = &(bodies[i]);
16        for (j = i + 1; j < N; j++)
17          {
18            struct planet *b2 = &(bodies[j]);
19            double dx = b1->x - b2->x;
20            double dy = b1->y - b2->y;
21            double dz = b1->z - b2->z;
22            double distance = sqrt(dx * dx + dy * dy + dz * dz + 0.01);
23            double mag = dt / (distance * distance * distance)
24
25            b1->vx -= dx * b2->mass * mag;
26            b1->vy -= dy * b2->mass * mag;
27            b1->vz -= dz * b2->mass * mag;
28            b2->vx += dx * b1->mass * mag;
29            b2->vy += dy * b1->mass * mag;
30            b2->vz += dz * b1->mass * mag;
31          }
32      }
33
34    for (i = 0; i < nbodies; i++)
35      {
36        struct planet *b = &(bodies[i]);
37        b->x += dt * b->vx;
38        b->y += dt * b->vy;
39        b->z += dt * b->vz;
40      }
41  }
```

Figure 3. Reference C implementation of the N-body problem.

Another difference we can observe in the C code is that no functional abstractions have been made, which obfuscate the relation to the mathematical specification. However, having the complete functionality within one function body enables a rather smart (hand-coded) optimisation. Instead of computing $N^2 - N$ accelerations, as in the SAC case, the C implementation only computes $\frac{N^2-N}{2}$ accelerations. This optimisation is based on the observation that for each pair i and j, the expressions under the sum sign in formula (4) for $\overset{k+1}{a_i}$ and $\overset{k+1}{a_j}$ share a computationally heavy part, as follows:

$$\frac{\left(\overset{k}{q}_j - \overset{k}{q}_i\right)}{\left|\overset{k}{q}_j - \overset{k}{q}_i\right|^3} = -\frac{\left(\overset{k}{q}_i - \overset{k}{q}_j\right)}{\left|\overset{k}{q}_i - \overset{k}{q}_j\right|^3}$$

Lines 25–30 in the C listing reflect this sharing of the computation as the velocities of both, `b1` and `b2` are updated adjacently. Although this measure substantially reduces the number of the computations, it does come at a price: the outer loop can no longer easily be parallelised, because the inner loop updates v_i and v_j, creating a potential for concurrent writes and thus race conditions. A similar optimisation could also be achieved in SAC. However, that would require the abstractions for the `acceleration` function to be changed, and it would require the `with`-loop, in lines 29–32 of the SAC listing, to be replaced by a `for`-loop. The latter would inhibit parallelisation of the outer loop in SAC.

4.3. Vectorisation

Looking at the reference C implementation, we observe that operations such as distance computation or velocity update are, in essence, performed on three-element vectors.

To investigate the effect of vectorisation of the C code, we augmented it by expressing these operations by using SIMD vectors, as shown in Figure 4. We use a set of gcc extensions that provide a portable interface to SIMD instructions across processor architectures. The SIMD units we use

```c
struct planet
{
  float x, y, z, padding0;
  float vx, vy, vz, padding1;
};

struct planet bodies[N];
float masses[N];

typedef float __attribute__ ((vector_size (16))) v4r;
#define vecptr(mem) ((v4r *)&(mem))

advance( int nbodies, struct planet *bodies, float dt)
{
  int i, j;

  for (i = 0; i < nbodies; i++)
    {
      struct planet *b1 = &(bodies[i]);
      for (j = i + 1; j < nbodies; j++)
        {
          struct planet *b2 = &(bodies[j]);
          register v4r d, d_2;
          float distance, mag;

          d = *vecptr (b1->x) - *vecptr (b2->x);
          d_2 = d * d;
          distance = sqrt(d_2[0] + d_2[1] + d_2[2] + 0.01);
          mag = dt / (distance * distance * distance);

          *vecptr (b1->vx) -= d * masses[j] * mag;
          *vecptr (b2->vx) += d * masses[i] * mag;
        }

      *vecptr (b1->x) += *vecptr (b1->vx) * dt;
    }
}
```

Figure 4. Reference C implementation augmented with explicit vectorisation.

during the experiments can operate on four single precision (Intel SSE) or four double precision (Intel AVX) floating point numbers. To compare absolute times, we used floats in all the vector implementations.

Intel architectures distinguish two load operations to vector registers: move from aligned or from unaligned memory[‡]. As an aligned move is noticeably faster than an unaligned move, we keep position triplets and velocity triplets at aligned addresses. We add two paddings in the structure, making the size of the structure a multiple of four floating-point numbers. This approach increases the total memory demand but ensures proper alignment. It further eliminates the problem of shifting and masking, which would be required if we performed vector operations by using the original representation. We define a vector type and exploit the fact that the address of the first element is also the address of the vector of all the positions of an individual element. The same holds for velocities. Note that the extensions allow us to mix scalar and vector operations and thus to express computations in a compact way.

4.4. OpenMP annotations

We consider two parallelisation strategies: inner loop parallelisation and outer loop parallelisation. The first approach preserves the number of computations used in the reference C implementation but changes the order of the computation to exploit more parallelism. The second approach mimics the SAC implementation: it uses twice as many computations but offers the chance for coarse-grain parallelisation on the outer level.

OpenMP inner loop parallelisation. In this implementation, we parallelise the inner loop of the `advance` function, which uses $\frac{N^2-N}{2}$ pair computations. As we can see from the reference implementation, the computation shape is a triangular matrix of size $N \times N$. If we were to parallelize inner loops of that reference algorithm, then a problem would arise: the lines of a triangular matrix that contain few elements introduce more threading overhead than performance gain. Therefore, we would like to rearrange the computation to make it more rectangular, rather than triangular. To do

[‡] A memory address is considered aligned if it can be divided by 16 in the case of SSE or by 32 in the case of AVX.

that, we apply the following technique. Let us look at a six-body problem as an example. First, we enumerate all pairs of indices j, i we are interested in:

$$\begin{array}{llllll}
0,0 & & & & & \\
0,1 & 1,1 & & & & \\
0,2 & 1,2 & 2,2 & & & \\
0,3 & 1,3 & 2,3 & 3,3 & & \\
0,4 & 1,4 & 2,4 & 3,4 & 4,4 & \\
0,5 & 1,5 & 2,5 & 3,5 & 4,5 & 5,5
\end{array}$$

Now, we want to make sure that we construct a rectangular array of size $N \times \frac{N}{2}$, where each line could be executed in parallel, such that any element of any pair occurs only once per line. We may see that the half of a second diagonal of the matrix, $[(0,5),(1,4),(2,3)]$, has this property. By using this observation as a basis, we construct a table in the following way: following the direction of the second diagonal of the matrix, we join $[(0,0)]$ with $[(1,5),(2,4),\dots]$, $[(0,1)]$ with $[(2,5),\dots]$, $[(0,2),\dots]$ with $[(3,5),\dots]$ and so on, excluding pairs from the main diagonal. That gives us the following table:

$$\begin{array}{lll}
(0,1) & (2,5) & (3,4) \\
(0,2) & (3,5) & \\
(0,3) & (1,2) & (4,5) \\
(0,4) & (1,3) & \\
(0,5) & (1,4) & (2,3) \\
(1,5) & (2,4) &
\end{array}$$

Using this table as a pseudo-scheduler provides reasonably good workload balance. It also allows us to instruct OpenMP to apply static scheduling to avoid any overheads inflicted by dynamic

```
struct pair
{
  short i, j;
};

struct pair idxes[N][N/2];

static inline void
precompute_idxes( void)
{
  int i, p0, p1, cnt;

  for (i = 0; i < N; i++)
    {
      for (cnt = 0, i != N - 1 ? (p0 = 0, p1 = i+1) : (p0 = 1, p1 = i);
           cnt < N/2;
           cnt++, p0 = (p0 + 1) % N, p1 = (p1 - 1) % N,
           p0 > p1 ? (p0 = i+2, p1 = N-1) : p0)
        {
          idxes[i][cnt] = (struct pair){p0, p1};
        }
    }
}

void
advance( int nbodies, struct planet *bodies, real dt)
{
  int i;

  omp_set_num_threads (8);
  for (i = 0; i < nbodies; i++)
    {
      int cnt;
      #pragma omp parallel for shared(idxes, dt) private(cnt) schedule(static)
      for (cnt = 0; cnt < N/2; cnt++)
        advance_pair (idxes[i][cnt].i, idxes[i][cnt].j, dt);
    }

  #pragma omp parallel for shared(bodies) private(i) schedule(static)
  for (i = 0; i < nbodies; i++)
    {
      struct planet *b = &(bodies[i]);
      *vecptr (b->x) += *vecptr (b->vx) * dt;
    }
}
```

Figure 5. C reference implementation explicitly parallelised by using OpenMP pragmas following the inner loop approach.

```c
void
advance( int nbodies, struct planet *bodies, float dt)
{
  int i;

  omp_set_num_threads (8);
  #pragma omp parallel for shared(bodies,dt) private(i) schedule(static)
  for (i = 0; i < nbodies; i++)
    {
      struct planet *b1 = &(bodies[i]);
      int j;

      for (j = 0; j < nbodies; j++)
        {
          struct planet *b2 = &bodies[j];
          register v4r d, d_2;
          float distance, mag;

          d = *vecptr (b1->x) - *vecptr (b2->x);
          d_2 = d * d;
          distance = sqrt(d_2[0] + d_2[1] + d_2[2] + 0.01);
          mag = dt / (distance * distance * distance);

          *vecptr (b1->vx) -= d * masses[j] * mag;
        }
    }

  #pragma omp parallel for shared(bodies,dt) private(i) schedule(static)
  for (i = 0; i < nbodies; i++)
    {
      struct planet *b = &(bodies[i]);
      *vecptr (b->x) += *vecptr (b->vx) * dt;
    }
}
```

Figure 6. C reference implementation explicitly parallelised by using OpenMP pragmas following the outer loop approach.

scheduling. This appears to be relevant, as initial experiments relying on dynamic scheduling, rather than what the aforementioned indexing scheme actually showed slowdowns. The complete code is shown in Figure 5.

The function `precompute_idxes` in lines 8–22 initialises a static matrix `idxes`, which holds the table described earlier for the given number of 1024 bodies. The actual computation is lifted into a function `advance_pair`, which is called with the index pairs coming from the scheduling matrix `idxes`. This scheduling happens in lines 33–35. Note that we omitted the definition of `advance_pair` as it basically consists of a copy of lines 21–33 of the vectorised version without OpenMP pragmas.

OpenMP outer loop parallelisation. As we mentioned earlier, the outer loop parallelisation gives us a coarser grained parallelism but doubles the number of computations. The code shown in Figure 6 uses very similar OpenMP annotations but this time on the outer for-loop. The scheduling is also static as the amount of work in every statically defined part is the same.

5. EVALUATION

This section presents our analysis of the performance of the SAC implementation of the N-body problem compared with the various C-based ones.

5.1. Experimental setup

Although the specification of the SICSA MultiCore Challenge suggests measuring 20 time steps of the N-body simulation, we decided to measure 200 time steps and to present average wall-clock runtimes per individual time step. Our experimental setup is motivated by the following observations: first, the runtime for 20 iterations, even when executed single-threaded, is too short for sufficiently accurate time measurements. Second, we see a perfectly linear relation between the number of iterations and the overall runtime. Third, with 200 iterations, we see very little fluctuation in the overall runtimes measured. In some critical cases, for example, highly parallel program runs on the Sparc T4-4 system, we rerun our experiments with 2000 iterations to confirm our measurements with 200 iterations.

DOI: 10.1002/cpe

We repeated every experiment 5 times and took the shortest execution time out of 5 successful program runs. We prefer shortest runtimes over, for instance, average runtimes because all codes are essentially deterministic. Any non-negligible difference in observed runtimes stems from purely coincidental activity of the operating system, which is both beyond our control and irrelevant for our findings. Taking average runtimes would potentially incorporate such coincidental activity and thus blur our observations.

Because on the given architectures vectorisation is only effective for single-precision floating point numbers, we first run all experiments with single-precision arithmetic and later conduct a separate analysis of the performance impact of double-precision floating point arithmetic for the non-vectorised test cases. Furthermore, we investigate the performance impact of increasing the problem size on the spatial domain. Starting out with the SICSA setup of 1024 bodies, we explore the effect of simulating 2048, 4096 and 8192 bodies, respectively.

5.2. SAC *vs C on Dell PowerEdge 2950 Beowulf cluster node*

Our first experiment runs the N-body problem, by using 32-bit floating-point arithmetic, on a single node of the Beowulf cluster. We relate the performance achieved by the SAC implementation to that of the C implementations discussed in the previous section: the plain reference code, the hand-vectorised variant and the two OpenMP hand-parallelised versions (inner and outer), each in a vectorised and in a non-vectorised form. Figure 7 shows average execution times for one time step of the N-body simulation, whereas Figure 8 visualises the same findings as speedups relative to the plain C reference implementation.

Let us first focus on sequential performance. Here, we must admit that the SAC program is about a factor of 2 slower than the C reference code. This performance difference can be attributed to essentially three independent factors. First of all, we can generally hold the high level of abstraction of the SAC code responsible for some performance impact when compared with any low-level C implementation. In this particular case, however, there are two additional aspects worth mentioning. For one, the SAC code does not exploit the symmetry of gravitation between any two bodies as the C reference implementation does. Consequently, it performs significantly more computations than the reference implementation. The similarly increased runtimes for the two OpenMP outer-loop parallelised versions nicely demonstrate the effect.

Last, but not least, the SAC code makes use of a different memory representation than the C reference implementation. Whereas the latter operates on a vector of 7 floating point numbers; in SAC, we properly separate positions, velocities and masses and use two vectors of three floating point numbers each for spatial information and a third vector for the masses. We made a small side

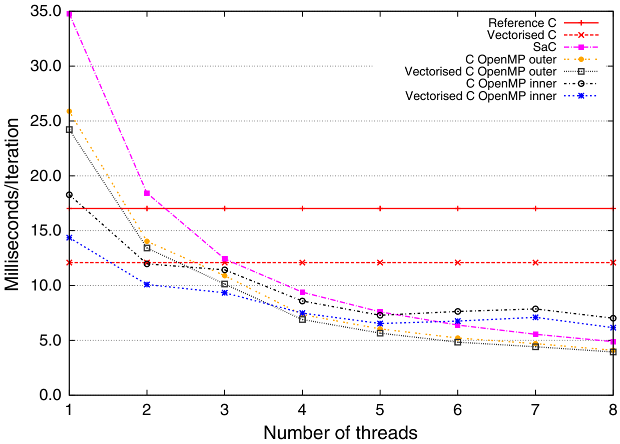

Figure 7. Average wall-clock execution times per N-body simulation step on a Dell PowerEdge 2950 Beowulf cluster node using single-precision floating point arithmetic.

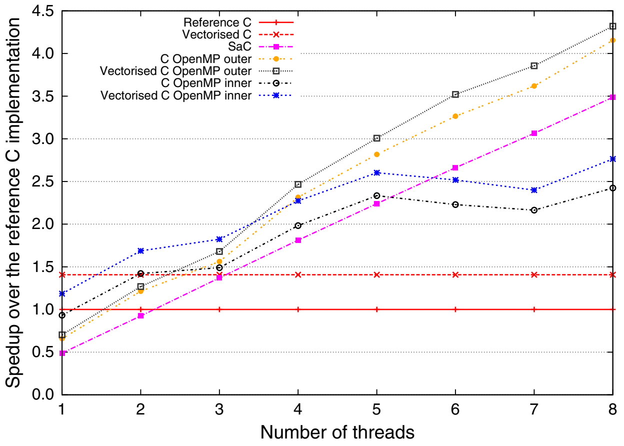

Figure 8. Speedups of average wall-clock execution times per N-body simulation step on a Dell PowerEdge 2950 Beowulf cluster node relative to the (sequential) C reference implementation using single-precision floating point arithmetic.

experiment to quantify the impact of these two different memory layouts. For this, we rewrite the C reference implementation to use a similar memory layout as SAC. This seemingly small change increased the execution time of the N-body simulation by about one third.

As Figures 7 and 8 show, our hand-vectorised version of the C reference code runs about 30% faster than the original code.

By using multiple cores, we found that the SAC implementation, as well as the OpenMP outer-loop version, quickly catch up with the reference C code. In fact, only two threads/cores suffice to break even in the case of SAC and to noticeably outperform the sequential code in the case of OpenMP (outer loop). Only three cores are needed to equalise even the (single-core) hand-vectorised C code. At about five cores, both the SAC code and the OpenMP outer-loop code begin outperforming the initially much faster OpenMP inner-loop code.

When utilising all eight cores of the machine, we observe a minimum runtime of around 4 ms per N-body iteration, which is less than a fourth of the sequential runtime of the reference implementation. The best performance (3.9 ms) is achieved by OpenMP with outer-loop parallelisation. Although the vectorised version does perform marginally better than the non-vectorised one, one must conclude that the effect of vectorisation here is disappointing. Not disappointing at all is that the SAC implementation finishes third with 4.9 ms. Relative to the initial single core runtime of 35 ms, this constitutes a 7.15-fold performance increase and thus an almost ideal utilisation of the parallel computing resource.

Figure 9 shows the effects of using different representations for floating point numbers to quantify the effect of changing numeric precision. We compare 32-bit single-precision (float) arithmetic with 64-bit double precision (double) arithmetic. The results show that for any particular implementation, little difference in performance can be observed between these numeric types. However, the vectorised versions are only available for `float` numbers because the SIMD architecture would not be able to keep an entire set of `double` data in the SIMD registers.

Finally, we investigate how scalability is affected by the number of bodies in the simulation. Increasing the problem size has two effects in the context of the all-pairs N-body problem: on the one hand, we expect an increased demand on the memory bus, which may limit the observable speedup. On the other hand, we expect less scheduling overhead as the number of tasks and their granularity increases.

We present the speedup of the best SAC runtime over (i) the sequential SAC runtime and (ii) the reference C code runtime for four different problem sizes: 1024, 2048, 4096 and 8192 bodies in Figure 10. We see slight improvements with increasing number of bodies, but saturation occurs at about 4096 bodies for which we achieve an almost ideal speedup of 7.8.

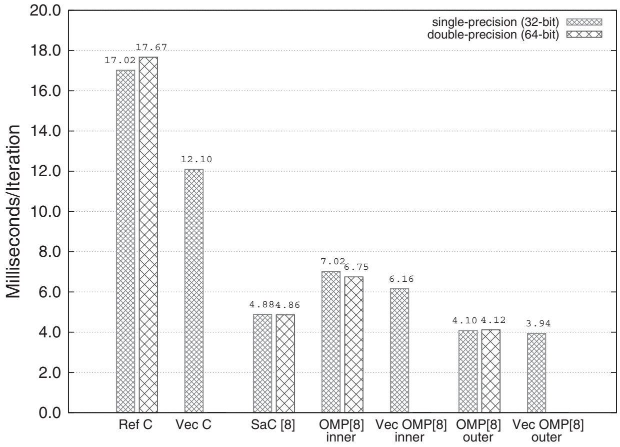

Figure 9. Performance impact of floating point arithmetic precision on a Dell PowerEdge 2950 Beowulf cluster node.

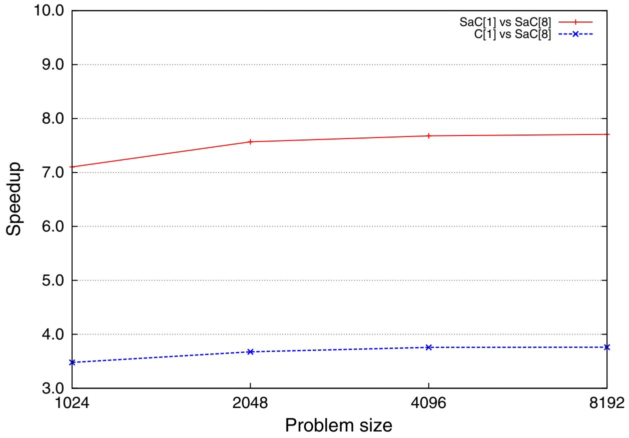

Figure 10. Performance impact of spatial problem size (number of bodies) on a Dell PowerEdge 2950 Beowulf cluster node.

5.3. SAC *versus C on an Intel Core-i7*

We repeated all the experiments described earlier in our second experimental environment: a quad-core hyper-threaded Intel Core-i7 system. Figure 11 shows average execution times for one time step of the N-body simulation, whereas Figure 12 again illustrates speedups relative to the plain C reference implementation.

At first glance, the results appear to be fairly similar to those obtained on the Dell PowerEdge. Having a closer look, we do observe two relevant differences, however.

First, performance encounters a hit when going from four threads to five threads, regardless of the code variant used. The hit is most noticeable for the two outer-loop OpenMP codes and least in the case of SAC, but the effect as such is uniform. What we observe here is the effect of four hyper-threaded cores as opposed to the eight fully-fledged cores of the Dell PowerEdge server system. With five threads used, hyper-threading is only effectively used on one of the cores and leads to load imbalance in the runtime systems of both SAC and OpenMP that expect a ‘real’ fifth core.

Second, we see that the OpenMP inner loop version for any number of cores outperforms the OpenMP outer loop version or the SAC code. We assume that this is an effect of the more modern Core-i7 design, which seems to cope better with the indexed memory accesses involved in that version.

DOI: 10.1002/cpe

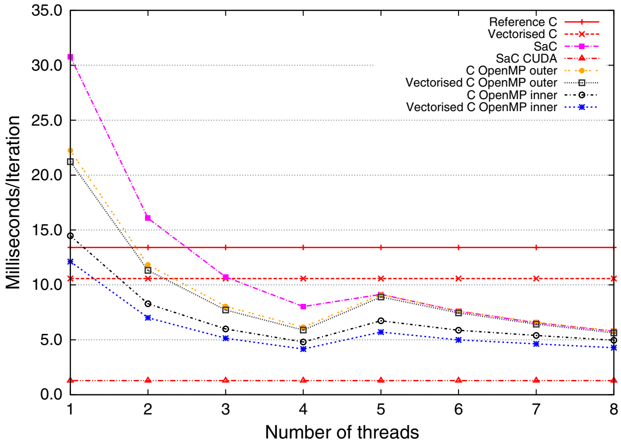

Figure 11. Average wall-clock execution times per N-body simulation step on a quad-core hyper-threaded Core-i7 processor with and without acceleration by an NVidia GTX-480 GPU using single-precision floating point arithmetic.

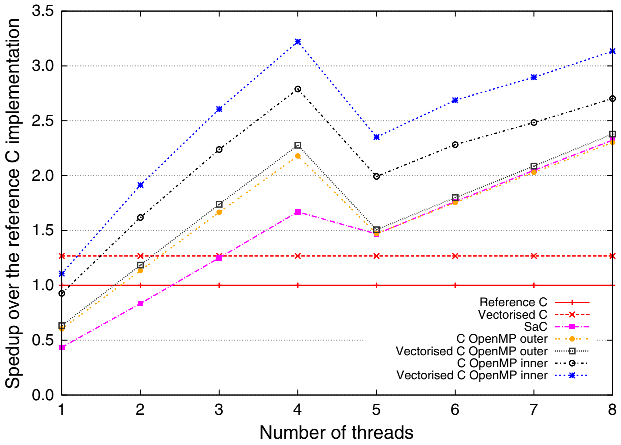

Figure 12. Speedups of average wall-clock execution times per N-body simulation step on a quad-core hyper-threaded Core-i7 processor relative to the (sequential) C reference implementation using single-precision floating point arithmetic.

We can further observe that OpenMP, regardless of parallelisation approach and vectorisation effort, does not benefit from hyper-threading, achieving roughly identical runtimes with four and with eight threads. In contrast, SAC effectively reduces the runtime from 8 ms using four threads to 5.8 ms using eight threads.

We summarise the absolute runtimes of our various implementations in Figure 13. With all cores used SAC outperforms the C reference implementation by a factor of 2.3. We also quantify the impact of floating point precision on overall performance in Figure 13. Similar to the Dell PowerEdge 2950, we observe little performance differences between single precision and double precision arithmetic on the Core-i7 processor.

Lastly, we investigate the impact of the spatial problem size on performance and show the results in Figure 14. Unlike in the case of the Dell PowerEdge 2950, where larger problem sizes led to marginally increased parallel performance, we cannot observe this effect here and rather see a constant ratio between parallel and sequential performance.

5.4. SAC *on Intel Core-i7 + NVidia GTX-480 graphics accelerator*

Our Intel Core-i7 system is also equipped with an NVidia GTX-480 graphics accelerator. Although the technical specifications of the GPU promise performance levels far beyond those of the

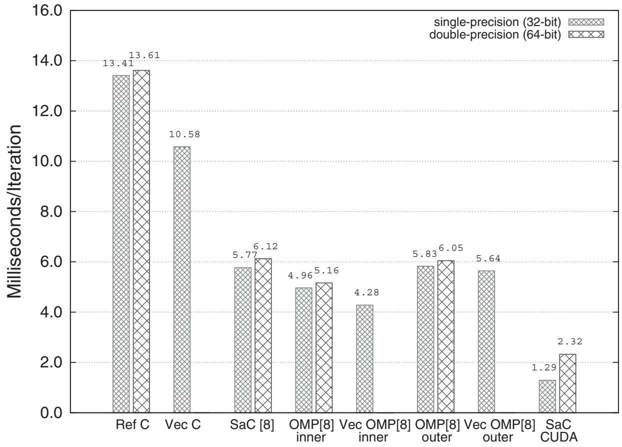

Figure 13. Performance impact of floating point arithmetic precision on a quad-core hyper-threaded Core-i7 processor and a GTX-480 GPU.

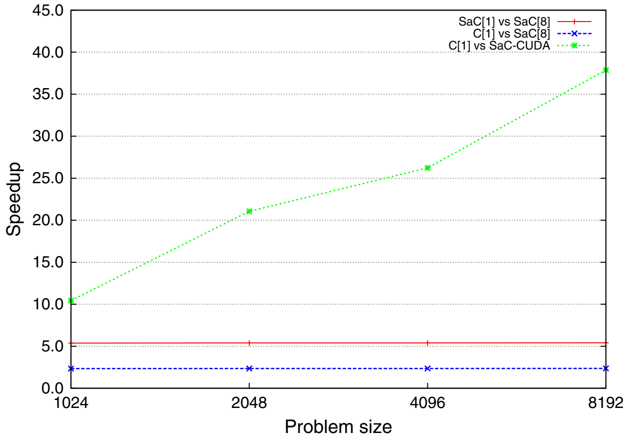

Figure 14. Performance impact of spatial problem size (number of bodies) on a quad-core hyper-threaded Core-i7 processor and a GTX-480 GPU.

Core-i7 processor (or even the Dell PowerEdge server used before), effectively harnessing this potential compute power is seriously difficult and time-consuming. The C reference implementation as such does not run on the GPU at all. Making it run based on the NVidia CUDA programming environment requires a complete non-trivial rewrite of the application into *CUDA kernels* and explicit organisation of memory transfers from host memory to GPU memory and vice versa, to name just a few issues.

One of the compelling features of the SAC compiler sac2c is its ability to fully automatically generate code suitable to run on NVidia GPUs [6]. This simply requires instruction through a command line option. As a consequence, the exact same SAC source code can be used to run executable programs on a variety of architectures.

For convenient comparison of the SAC runtime performance on the GTX-480 GPU with plain CPU performance of the Core-i7 processor, we include SAC CUDA times in Figures 11 and 13. As the figures show, the purely functional SAC implementation, when compiled for GPU execution, runs one N-body iteration in 1.3 ms. This is 3.3 times faster than the hand-vectorised, hand-parallelised C code on the CPU, 4.5 times faster than SAC compiled for CPU execution and 10.4 times faster than the original C reference implementation on the Core-i7 processor. We deliberately omit the SAC CUDA speedup in Figure 12 because the speedup of 10.4 is so much higher than any other measured speedup that the required y-axis scaling in Figure 12 would be detrimental to the readability of all other results.

Looking at the performance impact of floating point arithmetic precision in Figure 13, we observe that on the GTX-480 double precision code runs about 80% slower than single precision code. This clearly sets the GPU architecture of the GTX-480 apart from the CPU architectures we have looked at so far. As a consequence, this is still 2.1 times faster than the best performing hand-parallelised C code, 2.6 times faster than SAC compiled for CPU execution and about 5.9 times faster than the sequential C reference implementation.

Figure 14 demonstrates the impact of the number of bodies on performance also for using the GTX-480 graphics accelerator. In sharp contrast to using the CPU, we observe substantial improvements for the CUDA version as the problem size grows. Our speedup over the reference C version increases from 10.4 for 1024 bodies to almost 38 for 8192 bodies. The reason for this improvement lies in the fact that our runtimes do include the memory transfers to and from the graphics card as well as the overhead due to kernel invocations. With increasing body numbers, these overheads are much better amortised by the actual computations on the graphics card.

5.5. SAC *vs C on the Oracle Sparc SuperCluster T4-4*

Lastly, we ran an experiment on the T4-4 server to investigate how SAC scales on a highly parallel 256-fold hardware-threaded system. Figure 15 compares our SAC implementation with the (sequential) reference C implementation by using 32-bit floating point numbers. Our experiments confirm our previous experience with the machine that using all 256 hardware threads is extremely prone to coincidental operating system activity. Sometimes the combined performance of 256 threads is not better than what is achieved by only two threads. Therefore, we only present figures for using up to 255 threads in the following.

In fact, SAC scales rather well. With 16 threads, we achieve an almost linear speedup (precisely 15.87) over the sequential SAC-version and 8.8 over the reference C implementation. Going from 16 to 32 threads incurs a small performance penalty. We attribute this to memory bandwidth limitation and the lack of data sharing between threads in shared caches. Note that with 32 threads, exactly one thread runs on each physical core of the T4-4 system. So latency hiding through hardware multithreading is ineffective. Using multiple hardware threads per core performance increases again up to a 30.4-fold speedup over sequential SAC and 17.0 over the reference C implementation.

As expected, the (inner loop) OpenMP-based implementation again does not scale as good as the SAC code. Although it starts out on par with the C reference implementation, OpenMP achieves only about 15% speedup when using two threads instead of one. With 16 threads, it requires more than twice the time of SAC per iteration. Lastly, using more than 64 threads results in a performance decrease, whereas SAC makes effective use of up to 255 threads.

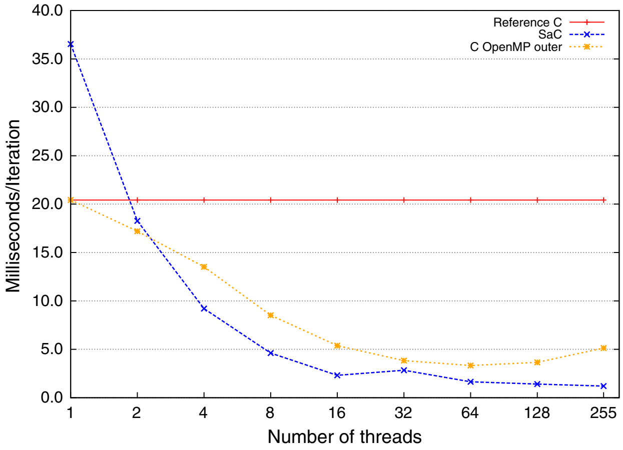

Figure 15. SAC versus C on Sparc T4-4 server using single precision.

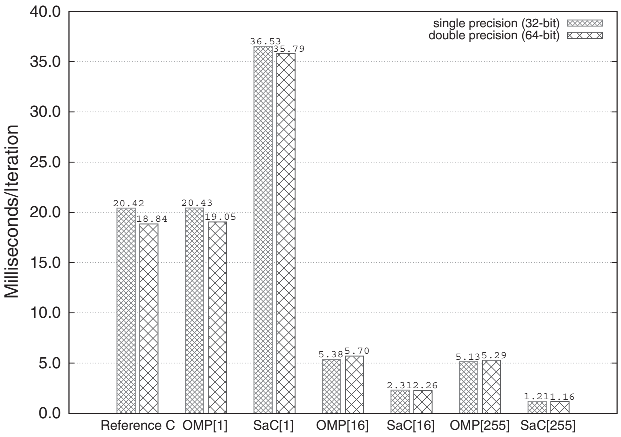

Figure 16. Performance impact of floating point arithmetic precision on Oracle Sparc SuperCluster T4-4.

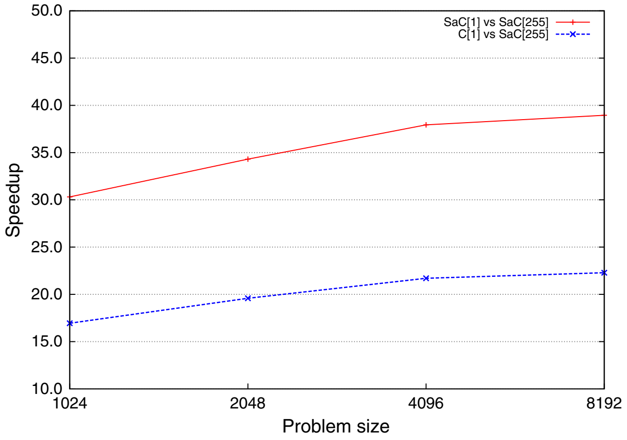

Figure 17. Performance impact of spatial problem size (number of bodies)on Oracle Sparc SuperCluster T4-4.

We were not able to repeat the vectorisation experiments on the Sparc T4-4 system as we did not manage to make gcc 4.7 run properly on this machine.

Figure 16 summarises our findings on the Sparc T4-4 system and shows the performance effect of floating point precision. The interesting insight is that on the T4 architecture, double precision arithmetic is actually marginally faster than single precision arithmetic. This is in contrast to the two ×86 architectures investigated before where double precision floating point performance was marginally lower than single precision and the GTX-480 graphics accelerator where double precision performance was considerably lower than single precision performance.

Comparing the execution times in Figure 16 with those in Figure 13 and in Figure 9, we see that with 1.21 ms for single-precision and 1.16 ms for double precision floating point arithmetic, the Sparc T4-4 turns out to deliver the best performance across the range of architectures we investigated. From a computer architecture perspective, this demonstrates that highly parallel general-purpose many-core systems such as the T4-4 can still compete well with more specialised graphics accelerators even for applications that suit the latter well. It must be noted, however, that the cost/performance ratio is nonetheless very much in favour of the GPU. Although a state-of-the-art graphics accelerator hardly costs more than around 1K EUR, the price tag of four T4 processors alone is around 45K EUR, not to mention a complete system as the T4-4 server we tested.

Lastly, we are interested on the impact of the spatial problem size on performance scalability; results are shown in Figure 17, in which we can observe the impact of latency hiding through hardware multithreading. With increased problem sizes from 1024 to 8192 bodies, we observe increased speedups from 30 to 38 for the intra-SAC comparison and from 17 to 22 for the comparison against the C reference code. Increased granularity does a better job of amortising overheads, and the increased demands on the memory subsystem are dealt with very effectively by hardware multithreading.

6. DISCUSSION

Our experiments show that the SAC implementation successfully combines a high level of abstraction, very close to the mathematical specification, with very competitive runtime performance. Without any user annotations or program manipulations of any kind, the automatically parallelising SAC compiler sac2c generates code that matches or even outperforms hand-vectorised and hand-parallelised C code on all three machines we used. The fact that this is achieved with no vectorisation support inside sac2c even leaves a considerable performance potential for future improvements.

Moreover, we demonstrate that the SAC compiler is able to transform, entirely automatically, the very same SAC source code into CUDA-enabled binary code that exploits the performance potential of a state-of-the-art graphics accelerator. At the same time, the SAC approach does not require the domain specialist to make particular programming efforts or to have any specialised knowledge or skills in parallel programming.

We observe that the array programming style advocated by SAC may seduce programmers into specifying algorithms in a seemingly sub-optimal style: the smart reuse of already computed gravitational effects, as it is carried out in the reference implementation in C, does not fit the data-parallel setting of SAC well. As a consequence, sequential SAC runtimes in the concrete example of the N-body simulation are nearly twice as long as those of the reference C implementation. In this light, it is highly interesting to observe that in the end, that is when we employ sufficiently many computational resources, both data-parallel specifications, the SAC and the OpenMP outer loop variant, actually outperform seemingly smarter implementations that avoid the redundant computations. This observation shows once more that in a parallel setting, redundant computations can be a tolerable or even a desirable trade-off if they help avoid conflicting memory accesses and reduce the number of synchronisation barriers. In our running example of all-pairs N-body simulation, only eight cores of a ×86-based SMP system suffice to amortise the redundant computations; on the Sparc T4-4 server even on four cores superior performance is achieved without the seemingly smart trick. The data-parallel approach scales considerably better and, thus, proves to be crucial for efficient multi-core and many-core performance.

Our measurements also show that the auto-vectorising capabilities of the C compilers used, at least for the plain C codes tested, do not succeed without the programmer's help. This does not come at a big surprise because a change in data layout seems to be essential. Although this cannot easily be automated in the context of C, it is possible in a SAC setting because the entire memory management in SAC is under the control of the compiler and its runtime system. Future work in this direction may yield further runtime improvements.

Finally, the fact that SAC shows consistently high performance on a variety of architectures, despite zero changes to the application source code, demonstrates the benefits that SAC provides to application programmers in terms of code portability and long-term maintainability in the presence of evolving machine architectures. This is in stark contrast to the observations we make with the reference C implementation. For that code, non-trivial rewrites are required to achieve similar performance. Furthermore, none of these rewrites enables us to utilise the graphics card. This would require yet another, even more complex, code rewrite based on low-level APIs such as CUDA or OpenCL.

We conclude that, for the running example of all-pairs N-body simulation, SAC and its tool chain do exhibit the desired combination of high software engineering productivity and high execution performance. These findings are in line with several previous application studies [26–28].

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their helpful comments and Oracle for temporarily providing us with a Sparc SuperCluster T4-4 in the context of their beta test programme.

DOI: 10.1002/cpe

REFERENCES

1. SICSA MultiCore Challenge, Phase II, 2012. (Available from: http://www.macs.hw.ac.uk/sicsawiki/index.php/Challenge-PhaseII [accessed on 31 July 2013]).
2. Barnes J, Hut P. A hierarchical $O(N \log N)$ force calculation algorithm. *Nature* 1986; **324**:446–449.
3. Grelck C, Scholz SB. SAC: a functional array language for efficient multithreaded execution. *International Journal of Parallel Programming* 2006; **34**(4):383–427.
4. SAC. (Available from: http://www.sac-home.org/ [accessed on 31 July 2013]).
5. Grelck C. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming* 2005; **15**(3):353–401.
6. Guo J, Thiyagalingam J, Scholz SB. Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In *6th Workshop on Declarative Aspects of Multicore Programming (DAMP'11)*. ACM Press: Austin, USA, 2011; 15–24.
7. OpenMP Architecture Review Board. OpenMP Application Program Interface, Version 3.1, July 2011. (Available from: http://www.openmp.org/mp-documents/OpenMP3.1.pdf [accessed on 31 July 2013]).
8. OpenMP. (Available from: http://www.openmp.org/ [accessed on 31 July 2013]).
9. NVidia. NVIDIA CUDA C Programming Guide 4.0. *Technical Report*, NVidia, 2011.
10. Khronos OpenCL Working Group. The OpenCL Specification, version 1.2, 15 November 2011. (Available from: http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf [accessed on 31 July 2013]).
11. OpenCL. (Available from: http://www.khronos.org/opencl/ [accessed on 31 July 2013]).
12. Scholz SB. With-loop-folding in SAC — condensing consecutive array operations. In *Implementation of Functional Languages, 9th International Workshop (IFL'97), St. Andrews, UK, Selected Papers*, Vol. 1467, Clack C, Davie T, Hammond K (eds), Lecture Notes in Computer Science. Springer: Berlin Heidelberg, 1998; 72–92.
13. Scholz SB. A case study: effects of with-loop folding on the NAS benchmark MG in SAC. In *Implementation of Functional Languages, 10th International Workshop (IFL'98), London, England, UK, Selected Papers*, Vol. 1595, Hammond K, Davie T, Clack C (eds), Lecture Notes in Computer Science. Springer: Berlin Heidelberg, 1999; 216–228.
14. Grelck C, Hinckfuß K, Scholz SB. With-loop fusion for data locality and parallelism. In *Implementation and Application of Functional Languages, 17th International Workshop (IFL'05), Dublin, Ireland, Revised Selected Papers*, Vol. 4015, Butterfield A (ed.), Lecture Notes in Computer Science. Springer: Berlin Heidelberg, 2006; 178–195.
15. Grelck C, Scholz SB, Trojahner K. With-loop scalarization: merging nested array operations. In *Implementation of Functional Languages, 15th International Workshop (IFL'03), Edinburgh, Scotland, UK, Revised Selected Papers*, Vol. 3145, Trinder P, Michaelson G (eds), Lecture Notes in Computer Science. Springer: Berlin Heidelberg, 2004; 118–134.
16. Scholz SB, Herhut S, Penczek F, Grelck C. SaC 1.0 – Single Assignment C – Tutorial. *Technical Report*, University of Hertfordshire, University of Amsterdam, 2010. (Available from: www.sac-home.org/publications/tutorial.pdf [accessed on 31 July 2013]).
17. Grelck C. Single Assignment C (SAC): high productivity meets high performance. In *4th Central European Functional Programming Summer School (CEFP'11), Budapest, Hungary*, Vol. 7241, Horváth Z, Zsók V (eds), Lecture Notes in Computer Science. Springer: Berlin Heidelberg, 2012; 207–278.
18. Nuzman D, Henderson R. Multi-platform auto-vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization, CGO '06*. IEEE Computer Society: Washington, DC, USA, 2006; 281–294, DOI: http://dx.doi.org/10.1109/CGO.2006.25. http://dx.doi.org/10.1109/CGO.2006.25.
19. Sreraman N, Govindarajan R. A vectorizing compiler for multimedia extensions. *International Journal of Parallel Programming* 2000; **28**:363–400.
20. Chen KH, Shen BY, Yang W. An automatic superword vectorization in LLVM. *16th Workshop on Compiler Techniques for High-Performance and Embedded Computing*, Taipei, 2010; 19–27.
21. Šinkarovs A, Scholz SB. Portable support for explicit vectorisation in C. *16th Workshop on Compilers for Parallel Computing (CPC'12)*, Padua, Italy, 2012. http://ashinkarov.github.io/publications/cpcgcc.pdf [accessed on 31 July 2013].
22. Chapman B, Jost G, van der Pas R. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press: Cambridge Massachusetts, 2008.
23. TOP500. (Available from: http://www.top500.org/ [accessed on 31 July 2013]).
24. SUN/Oracle. A technical overview of the Oracle Sparc Supercluster T4-4. *White paper*, SUN/Oracle, 2012.
25. Fulgham B. The Computer Language Benchmarks Game, 2012. (Available from: http://benchmarksgame.alioth.debian.org/ [accessed on 31 July 2013]).
26. Kudryavtsev A, Rolls D, Scholz SB, Shafarenko A. Numerical simulations of unsteady shock wave interactions using SAC and Fortran-90. In *10th International Conference on Parallel Computing Technologies (PaCT'09)*, Vol. 5083, Lecture Notes in Computer Science. Springer: Berlin Heidelberg, 2009; 445–456.
27. Grelck C, Douma R. SAC on a Niagara T3-4 Server: lessons and experiences. In *Applications, Tools and Techniques on the Road to Exascale Computing*, Vol. 22, de Bosschere K, D'Hollander E, Joubert G, Padua D, Peters F, Sawyer M (eds), Advances in Parallel Computing. IOS Press: Amsterdam, 2012; 289–296.
28. Wieser V, Grelck C, Haslinger P, Guo J, Korzeniowski F, Bernecky R, Moser B, Scholz S. Combining high productivity and high performance in image processing using Single Assignment C on multi-core CPUs and many-core GPUs. *Journal of Electronic Imaging* 2012; **21**(2):021116-1–021116-13.

DOI: 10.1002/cpe