# Towards Truly Boolean Arrays in Data-Parallel Array Processing

Clemens GRELCK [a], Hraban LUYAT [a]
[a] *Computer Systems Architecture Group*
*University of Amsterdam*
*Amsterdam, Netherlands*

**Abstract.** Booleans are the most basic values in computing. Machines, however, store Booleans in larger compounds such as bytes or integers due to limitations in addressing memory locations. For individual values the relative waste of memory capacity is huge, but the absolute waste is negligible. The latter radically changes if large numbers of Boolean values are processed in (multidimensional) arrays. Most programming languages, however, only provide sparse implementations of Boolean arrays, thus wasting large quantities of memory and potentially making poor use of cache hierarchies.

In the context of the functional data-parallel array programming language SAC we investigate dense implementations of Boolean arrays and compare their performance with traditional sparse implementations. A particular challenge arises in data-parallel execution on today's shared memory multi-core architectures: scheduling of loops over Boolean arrays is unaware of the non-standard addressing of dense Boolean arrays. We discuss our proposed solution and report on experiments analysing the impact of the runtime representation of Boolean arrays both on sequential performance as well as on scalability using up to 32 cores of a large ccNUMA multi-core system.

**Keywords.** Boolean arrays, data parallelism, ccNUMA architectures, functional array programming, automatic parallelisation

## 1. Introduction

Booleans are the most basic values in computing, and as such in one way or another supported by any programming language. Surprisingly, at least from a conceptual point of view, this does not hold for arrays of Boolean values. In almost all programming languages Booleans effectively require more than one bit for representation. While this is largely irrelevant for individual values, it does make a crucial difference for large arrays of Boolean values if the required storage grows by a factor of 8 or 32. For example, in the latest C standard the (new) built-in type `_Bool` requires 8 bits storage; a single bit in a bit structure requires 32 bit storage.

Inefficient storage of Boolean arrays has a secondary adverse effect on runtime performance: relevant information is sparsely scattered across memory. This results in low data locality, inefficient utilisation of cache hierarchies, and thus poor performance on modern ccNUMA systems. The situation is, of course, not without reason. Since the early days of computing memory cannot be addressed on the level of bits, but at best at

the level of bytes. Actual memory transfers are usually organised in even much coarser grained units, so-called cache lines.

Despite the lack of support in many modern programming languages, Boolean arrays do have countless applications in areas such as image and signal processing, text and image compression/decompression, etc.

In this paper we propose a dense memory representation of Boolean arrays. We circumvent the limitations of computer hardware, as described above, by a double-layered indexing scheme when reading elements from or writing elements to a Boolean array. First, we extract the corresponding byte or double word (configurable) from memory, then we read or set the individual bit by means of bit-wise logic operations, and if needed write the entire byte or double word back to memory. This implementation makes a trade-off between the negative performance effect of double-layered indexing and the positive effect of improved cache utilisation. In the presence of an ever-growing disparity between memory speed and processor speed we speculate on a net gain in many scenarios.

We investigate the effect of dense vs sparse representation of Boolean arrays in the context of the functional data-parallel array language SAC (Single Assignment C, [6,3]). SAC combines a purely functional, state-free semantics with a C-like syntax and high-level support for multi-dimensional stateless arrays; it aims at reconciling high programming productivity with competitive performance in the age of multi- and many-core computing through fully compiler-directed parallelisation [2,7,1]. A number of case studies demonstrate the effectiveness of the approach [8,4,9].

The need for parallel execution on any modern computing platform substantially complicates the efficient implementation of densely stored Boolean arrays. In SAC, just as in any other shared memory data parallel approach (e.g. OpenMP), a loop scheduler in one way or another distributes iterations over compute cores. It does so regardless of the base type, and thus it becomes likely that the boundary between loop indices computed by one core and those computed by another core falls within a byte or double word used for compact storage of Boolean element values. We investigate two different approaches to deal with this situation.

## 2. Introducing SAC

Core SAC is a functional, side-effect free variant of C: we interpret assignment sequences as nested let-expressions, branches as conditional expressions and loops as syntactic sugar for tail-end recursive functions. Despite the radically different underlying execution model (context-free substitution of expressions vs. step-wise manipulation of global state), all language constructs adopted from C show exactly the same operational behaviour as expected by imperative programmers. Our choice of syntax is meant to facilitate adoption of SAC by programmers with an imperative background while the SAC compiler can exploit the benefits of a side-effect free semantics for advanced optimisation and automatic parallelisation.

On top of this language kernel SAC provides genuine support for truly multidimensional arrays promoting a shape-generic style of programming. Arrays are truly stateless/functional and as such array values may be passed as arguments or results of functions without restrictions. SAC only provides a small set of built-in array operations. Essentially, these are primitives to retrieve data pertaining to the structure and contents of

arrays, e.g. an array's rank (`dim(`*array*`)`) or its shape (`shape(`*array*`)`). Any multidimensional array is characterised by its rank scalar, that determines the array's number of dimensions (or axes) and the shape vector, which determines the extent of the arry along each dimension (or axis). Thus, the rank of an array defines the length of its shape vector, and the product of all elements of the shape vector yields the array's number of element. A selection facility provides access to individual elements or entire subarrays using a familiar square bracket notation: *array*`[`*idxvec*`]`. All aggregate array operations are specified using WITH-loop expressions, a SAC-specific array comprehension:

```
with {
   ( lower_bound <= idxvec < upper_bound ) : expr;
    ...
   ( lower_bound <= idxvec < upper_bound ) : expr;
}: genarray( shape, default )
```

The key word `genarray` makes this with-loop define an array whose shape is given by the *shape* expression and whose elements default to the value of the *default* expression. The body consists of multiple (disjoint) *partitions*. Here, *lower_bound* and *upper_bound* denote expressions that must evaluate to integer vectors of equal length. They define a rectangular (generally multidimensional) index set. The identifier *idxvec* represents elements of this set, similar to induction variables in for-loops. We call the specification of such an index set a *generator* and associate it with some potentially complex SAC expression. Thus, we define a mapping between index vectors and values, in other words an array. We deliberately use index *sets*, which make any with-loop expose fine-grained concurrency and thus the ideal basis for parallelisation.

There are several variants of with-loops, among others for the specification of reduction operations. Furthermore, generators are not confined to dense rectangular spaces, but are generalised to cover various forms of periodic patterns. As a consequence, a single with-loop generally defines a fairly complex aggregate multi-dimensional array operation. Aggressive optimisation with the aim to condense many light-weight with-loops into few heavy-weight ones typically lead to non-trivial iteration spaces [5].

## 3. Implementation

For the implementation of dense Boolean arrays we can focus ourselves on three array operations: allocation of memory suitable to hold a new array value, reading an element of an array at a given index and writing a new given value at a given index of some array. These three operations are well encapsulated in the code generator of the SAC compiler limiting the necessary implementation effort.

```
contype *ba_malloc( shp)
{
  return (con_type *) malloc( ⌈ (∏_{i=0}^{|shp|-1} shp_i) / (|con_type| * 8) ⌉ );
}
```

**Figure 1.** Allocation procedure for dense bit arrays

Fig. 1 shows a pseudo C implementation of the array allocation operation. As only argument `ba_malloc` receives the shape of the new array value; it yields a pointer to a sufficiently sized chunk of contiguous address space cast to the chosen *container type* that hosts a number of Boolean values. For the purpose of experimentation we abstract from the concrete container type; any (unsigned) integer type can be used here. Of course, the C language requires us to choose a concrete container type at compile time.

As pointed out in Section 2, the number of elements of a multidimensional array equals the product of the elements of the shape vector. Since every container element holds a number of Boolean values, we divide the number of array elements by the size of the container type in bits. We use bars to denote the C `sizeof` operator, that gives the size of some type in bytes. As we use bits, we further multiply the size of the container type by 8.

Since the concrete container type is fixed at compile time, its size is available to the compiler. Assuming that the size of any suitable container type is a power of two bytes, any optimising compiler should manage to replace the costly division operation by a corresponding cheap bit shift operation. For the sake of readability we refrain from manually applying such optimisations to this and the following pseudo codes.

```
bool *ba_read( idx, arr)
{
  ravel_idx = Σ_{i=0}^{|idx|-1} (idx_i * Π_{j=i+1}^{|idx|-1} shape(arr)_j);
  outer_idx = ravel_idx  (|con_type| * 8);
  inner_idx = ravel_idx % (|con_type| * 8);
  return !! (arr[outer_idx] & (1 << inner_idx));
}
```

**Figure 2.** Read operation for dense bit arrays

Fig. 2 shows our implementation of the dense bit array read operation in pseudo C code. The `ba_read` operation receives an index vector and a multidimensional array and it yields a Boolean value, more precisely zero or one. First, we determine the *ravel index*, i.e. the scalar index into flat contiguous address space that is equivalent to the index vector into the given multidimensional array, following the well known Horner scheme. This ravel index is then divided into an *outer index* specifying the index of the corresponding container element and the *inner index* that defines the bit address within the container element. Next, we compute a suitable mask by shifting the number 1 as many bit positions left as prescribed by the inner index. Then, we compute the bit-wise conjunction of the container element and the mask and, thus, extract the Individual Boolean value. At last, the double exclamation mark operator normalises the value to 0 (false) or 1 (true).

The third and last bit array operation, bit write, is shown in Fig. 3. We follow the example of `ba_read` and first compute ravel, outer and inner index in the same way as before. The further procedure, however, depends on the given Boolean value. If the value is `true`, we compute the same bit mask as in the read operation, but this time we set the container element to the bit-wise disjunction of the previous element and the mask. This sets the indexed bit value to `true` and leaves all other bits in the container unchanged. If the given value is `false`, we invert the mask and then compute set the container to the bit-wise conjunction of the previous value and the inverted mask. This sets the indexed bit to `false` and leaves all other bits in the container unchanged.

```
void ba_write( idx, arr, val)
{
  ravel_idx = sum_{i=0}^{|idx|-1} (idx_i * prod_{j=i+1}^{|idx|-1} shape(arr)_j);
  outer_idx = ravel_idx  (|con_type| * 8);
  inner_idx = ravel_idx % (|con_type| * 8);
  if (val) {
   arr[outer_idx] |= 1 << inner_idx);
  } else {
   arr[outer_idx] &= ~(1 << inner_idx);
  }
}
```

**Figure 3.** Write operation for dense bit arrays

Array programming naturally lends itself to multi-core and many-core architectures. Consequently, SAC is first and foremost a data-parallel language. The obvious basis for (data-)parallel program execution are the with-loops introduced in Section 2. Usually the number of concurrently computable indices in a multidimensional index space by several orders of magnitude exceeds the available number of cores. Thus, an *index scheduler* (often called *loop scheduler*) assigns indices to cores (more precisely kernel threads) for computation taking various aspects into account such as load balancing, data locality in deep cache hierarchies as well as overhead. Neither the problem nor the solutions are specific to SAC, but can in variations be found in any data-parallel shared memory approach, e.g. OpenMP.

```
void ba_write_mt( idx, arr, val)
{
  ravel_idx = sum_{i=0}^{|idx|-1} (idx_i * prod_{j=i+1}^{|idx|-1} shape(arr)_j);
  outer_idx = ravel_idx  (|con_type| * 8);
  inner_idx = ravel_idx % (|con_type| * 8);
  if (val) {
   do {
    old = arr[outer_idx];
    new = old | (1 << inner_idx);
   } while (! _cas( &arr[outer_idx], old, new));
  } else {
   do {
    old = arr[outer_idx];
    new = old & (1 << inner_idx);
   } while (! _cas( &arr[outer_idx], old, new));
  }
}
```

**Figure 4.** Thread-safe write operation for dense bit arrays

Since the index scheduler is not aware of the array's base type and in particular not of the bit-wise representation that we are interested in here, different threads on different cores may concurrently write Boolean values that coincidentally map to the same con-

tainer element. The `ba_write` operation in Fig. 3 is a combination of three consecutive more basic operations: loading a container element from memory into a register, updating the register with the corresponding bit manipulation and writing the resulting value back to memory. If independent threads perform this operation concurrently on different bits of the same container element, we are likely to see interleavings that overwrite bits with false values. To avoid this and to ensure the deterministic behaviour of programs we must embed the above three steps into a critical region that guarantees their single-threaded execution.

The straightforward approaches to implementing critical regions are mutex locks or semaphores. However, in our fine-grained scenario of bit-wise operations it must be expected that these techniques incur prohibitive overhead that would invalidat our entire approach. As shown in Fig. 4, we rather follow a so-called optimistic approach. Apart from rewriting the bit-wise operation such that they expose both the old and the new value of a container element we use a compare-and-swap operation (`cas`). This operation is supported in hardware by all modern instruction set architectures. If the value at the address given as first argument equals that of the second argument, the operation atomically writes the third value to the given address. Otherwise, it does nothing. The result signals success or failure. The assumption is that in most cases the atomic write succeeds. Only in case of statistically unlikely conflicts the `cas` operation fails and the whole operation is repeated.

## 4. Experimental evaluation

Our experimental system is a 48-core ccNUMA SMP machine with 4 AMD Opteron 6172 "Magny-Cours" processors running at 2.1 GHz eqipped with 128 GB of DRAM. Each processor core has 64 KB of L1 instruction cache, 64 KB of L1 data cache and 512 KB unified L2 cache. Each group of 6 cores shares one L3 cache of 6 MB. The system runs Linux kernel 2.6.18 with Glibc 2.5.

Our first benchmark, Gauss, stresses memory read performance. A sliding window of eleven elements is moved over a vector of Booleans, and a simple computation on these eleven bits yields the result value that is written back to another vector of Booleans. We investigate and compare six different implementations of Boolean arrays: a traditional sparse implementation, a thread-unsafe dense implementation and a thread-safe dense implementation, each based on either 8-bit or 32-bit memory addressing (i.e. using either `unsigned char` or `unsigned int` as implementation types).

Fig. 5 shows the results obtained with the Gauss benchmark using up to 32 cores. Performance-wise we can easily identify that the six implementation variants form two groups with almost identical runtime behaviour. The dense implementations of Boolean arrays clearly outperform the traditional sparse implementations in this scenario. Neither the question of implementation type (8-bit or 32-bit) nor making the write operation thread-safe shows any measurable impact on performance. In essence, we can observe that improved data locality in the cache hierarchy more than outweighs the overhead inflicted by two-level indexing into memory as outlined in the previous section. It is noteworthy, however, that this benefit only arises in terms of scalability in parallel execution. Sequential execution on a single core, albeit outside the range of Fig. 5 yields the same runtime for all six variants.

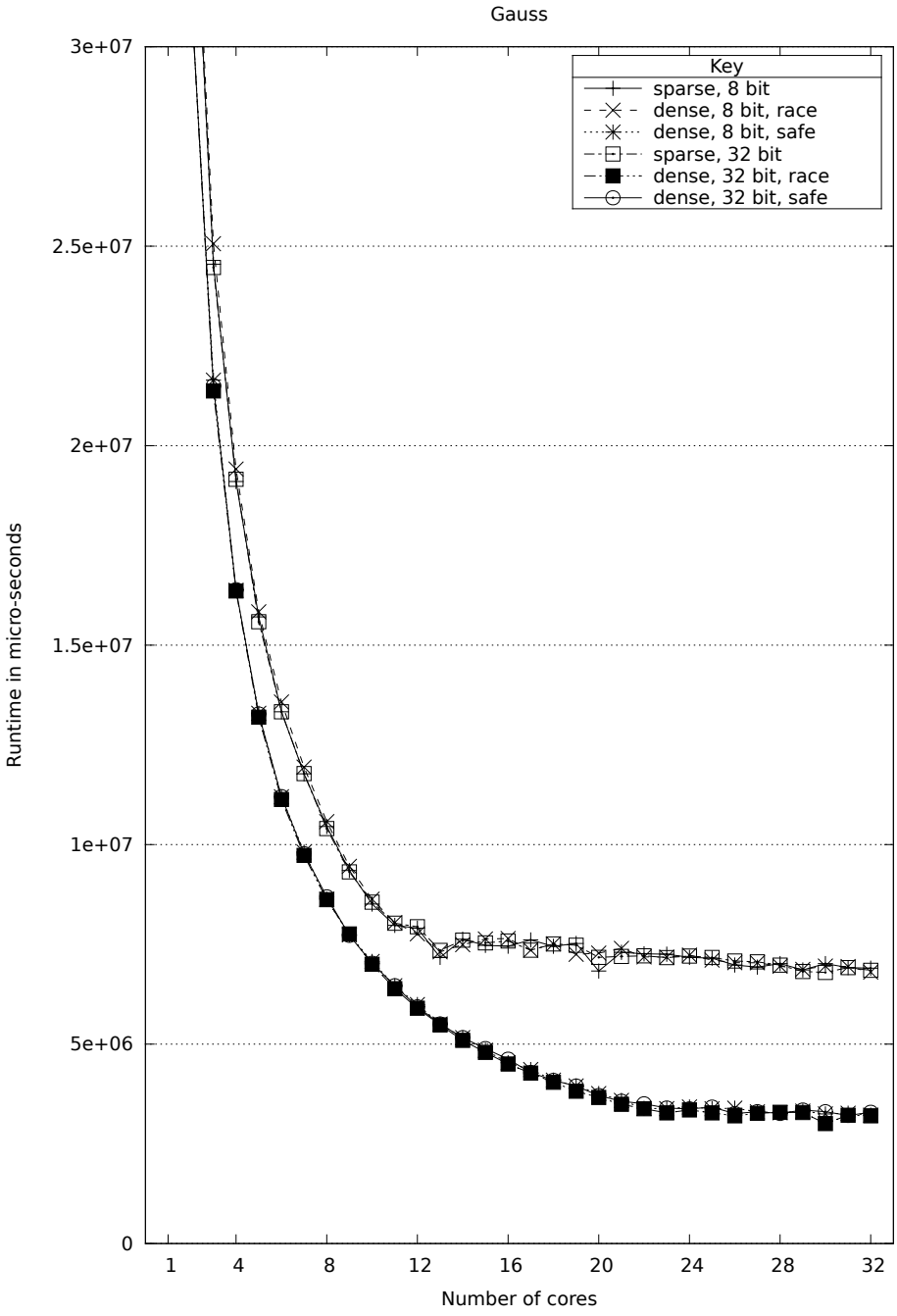


**Figure 5.** Dense vs sparse implementations of Boolean arrays: Gauss benchmark

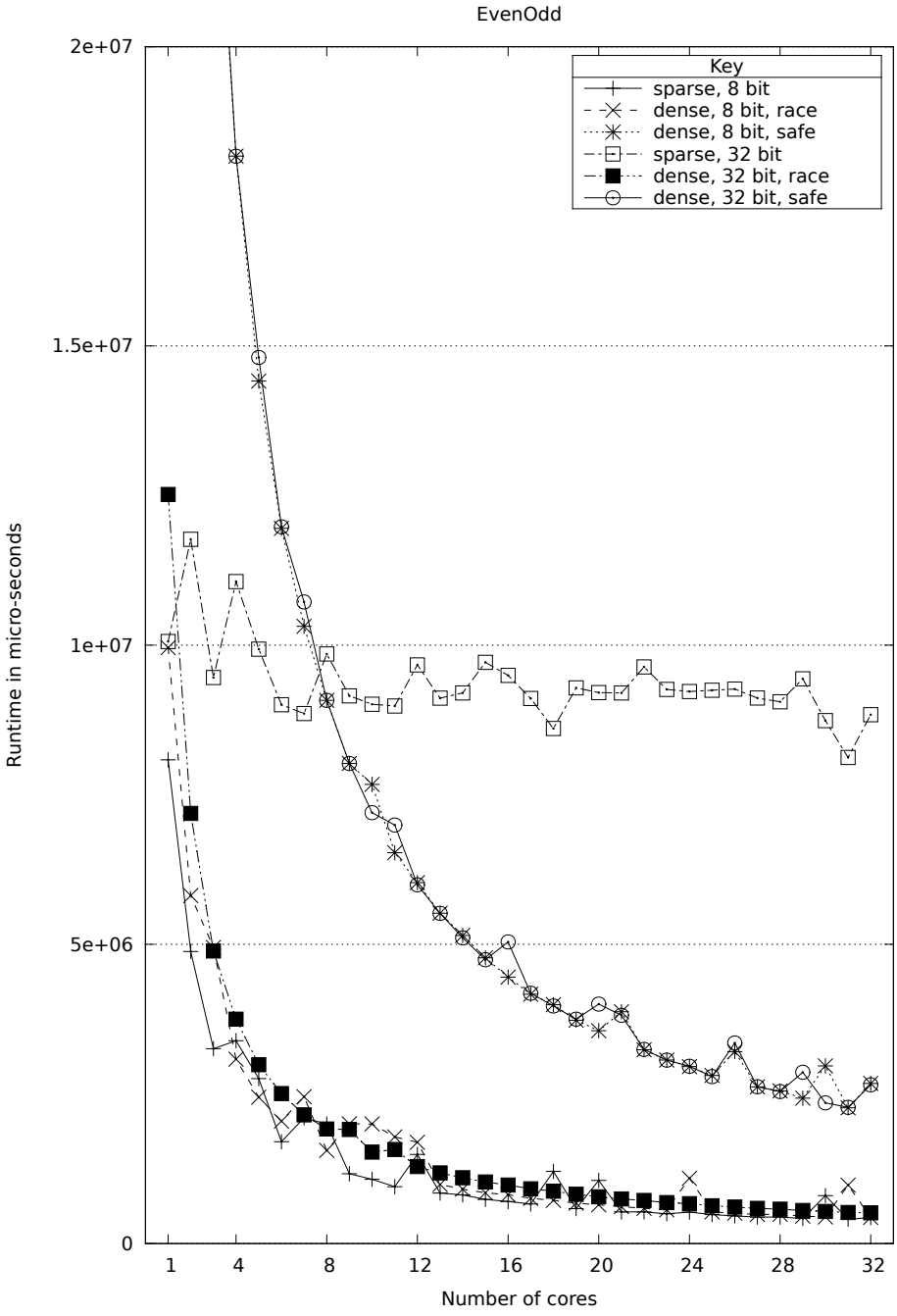


**Figure 6.** Dense vs sparse implementations of Boolean arrays: EvenOdd benchmark

Our second benchmark, EvenOdd, stresses write performance: it alternatingly writes `true` and `false` to a vector of Booleans without any read instructions whatsoever. For this kind of operation we would expect much less favourable results, and indeed Fig. 6 confirms our epectations. The thread-safe dense implementations scale well, but start out from a 5–6 times higher execution time than their unsafe counterparts. The 32-bit sparse implementation, in contrast, does not scale at all. Apparently, the benchmark immediately becomes memory-bound. The best performance is indeed achieved by the 8-bit sparse implementation, but the thread-unsafe dense implementations are more or less on par for larger core counts. Given that this benchmark represents the worst case for our approach this is encouraging. It shows, however, that synchronising the loop scheduler with the storage format to avoid the costly compare-and-swap operations may be worthwhile as future work.

## 5. Related Work

Despite the fact that Boolean values are the most fundamental unit of information in computing, arrays of Boolean values have largely been neglected in the design and implementation of programming languages. In early versions of C, for instance, no basic type for Booleans existed. Only with the C99 standard C adopted the type `_Bool` from C++ as an elementary data type. While C and C++ traditionally do not fix the representation size of elementary data types, arrays of Booleans use the smallest addressable unit (i.e. a whole byte) to represent a single value. This is equivalent to the 8-bit sparse format that we investigated in Section 4. Similarly bit structures in C++ and C99 do not lead to dense representations of Boolean arrays either.

The same holds in one way or another for most popular imperative languages. For example, Java, C#, Fortran and Pascal unanimously define the elementary type Boolean to have one byte size. Consequently, arrays of Booleans inevitably lead to the same 8-bit sparse representation as above. The situation is not very different in general-purpose functional languages such as ML, OCaml or Haskell. Here, the situation is aggravated by the fact that continuously store arrays are not at the heart of language design.

At the same time, usually non-standardised libraries exist in most of the above mentioned programming languages that implement densely stored bit arrays using similar low-level bit-wise operations as we do in Section 3. We are not aware of any such implementation, however, that takes parallel execution in shared memory environments into account whatsoever.

More related work can be found in the area of interpreted array languages such as APL and J. With the clear focus on arrays, also arrays of Boolean values have been more in the focus of design and implementation than in most other programming languages, and, consequently, dense representations of Boolean arrays are fairly common in array languages. Implementation-wise they benefit from the fact that programs are predominantly written by composition of built-in operations on whole arrays rather than by some form of loop that iterates over individual array elements. This makes it much easier to map bit-wise operations back to pseudo-vectorised operations on bytes, words or double words, and, thus, to harness substantial improvements in runtime performance. To the present day, these interpreted array languages are run sequentially.

## 6. Conclusion and Future Work

We propose a dense, bit-wise implementation for Boolean arrays in the functional, data-parallel array language SAC, Single Assignment C. We further compare the relative performance of dense and sparse memory representations of Boolean arrays in the context of shared memory data-parallel execution. While we made this work in the context of SAC, our findings are relevant across all languages that deal with data-parallel arrays, e.g. OpenMP extensions of C or Fortran. They demonstrate that even in the absence of (pseudo-) vectorisation of bit-wise array operations dense storage formats can outperform that straightforward sparse array representations due to improved data locality when memory bandwidth is a scarce resource as is common in larger multi-core systems.

In the future we aim at conducting further experiments to validate our results on further architectures and explore opportunities for compiler-directed pseudo vectorisation, i.e. replacing inner loops over Boolean arrays by appropriate bit-wise operations on whole character or integer values.

## References


[1] M. Diogo and C. Grelck. Heterogenous computing without heterogeneous programming. In K. Hammond and H. Loidl, editors, *Trends in Functional Programming, 13th Symposium, TFP 2012, St.Andrews, UK*, volume 7829 of *Lecture Notes in Computer Science*. Springer, 2013. to appear.

[2] C. Grelck. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming*, 15(3):353–401, 2005.

[3] C. Grelck. Single Assignment C (SAC): high productivity meets high performance. In V. Zsók, Z. Horváth, and R. Plasmeijer, editors, *4th Central European Functional Programming Summer School (CEFP'11), Budapest, Hungary*, volume 7241 of *Lecture Notes in Computer Science*, pages 207–278. Springer, 2012.

[4] C. Grelck and R. Douma. SAC on a Niagara T3-4 Server: Lessons and Experiences. In K. de Bosschere, E. D'Hollander, G. Joubert, D. Padua, F. Peters, and M. Sawyer, editors, *Applications, Tools and Techniques on the Road to Exascale Computing*, volume 22 of *Advances in Parallel Computing*, pages 289–296. IOS Press, Amsterdam, 2012.

[5] C. Grelck and S.-B. Scholz. Merging compositions of array skeletons in SAC. *Journal of Parallel Computing*, 32(7+8):507–522, 2006.

[6] C. Grelck and S.-B. Scholz. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

[7] J. Guo, J. Thiyagalingam, and S.-B. Scholz. Towards Compiling SaC to CUDA. In Z. Horváth and Viktória Zsók, editors, *10th Symposium on Trends in Functional Programming (TFP'09)*, pages 33–49. Intellect, 2009.

[8] A. Kudryavtsev, D. Rolls, S.-B. Scholz, and A. Shafarenko. Numerical simulations of unsteady shock wave interactions using SAC and Fortran-90. In *10th International Conference on Parallel Computing Technologies (PaCT'09)*, volume 5083 of *Lecture Notes in Computer Science*, pages 445–456. Springer, 2009.

[9] A. Šinkarovs, S. Scholz, R. Bernecky, R. Douma, and C. Grelck. SAC/C formulations of the all-pairs N-body problem and their performance on SMPs and GPGPUs. *Concurrency and Computation: Practice and Experience*, 2013. to appear.