Abstract Expressionism for Parallel Performance

(This paper is an expanded version of that presented at ARRAY'15.)

Robert Bernecky

Snake Island Research Inc, Canada
bernecky@snakeisland.com

Sven-Bodo Scholz

Heriot-Watt University, UK
S.Scholz@hw.ac.uk

Abstract

Programming with abstract, mathematical expressions offers benefits including terser programs, easier communication of algorithms, ability to prove theorems about algorithms, increased parallelism, and improved programming productivity. Common belief is that higher levels of abstraction imply a larger semantic gap between the user and computer and, therefore, typically slower execution, whether sequential or parallel. In recent years, domain-specific languages have been shown to close this gap through sophisticated optimizations benefitting from domain-specific knowledge.

In this paper, we demonstrate that the semantic gap can also be closed for non-domain-specific functional array languages, without requiring embedding of language-specific semantic knowledge into the compiler tool chain. We present a simple example of APL-style SaC programs, compiled into C-code that outperform equivalent C programs in both sequential and parallel (OpenMP) environments.

We offer insights into *abstract expressionist* programming, by comparing the characteristics and performance of a numerical relaxation benchmark written in C99, C99 with OpenMP directives, scheduling code, and pragmas, and in SaC, a functional array language. We compare three algorithmic styles: if/then/else, hand-optimized loop splitting, and an abstract, functional style whose roots lie in APL.

We show that the SaC algorithms match or outperform serial C, and that the hand-optimized and abstract SaC styles generate identical code, and so have identical performance. Furthermore, parallel SaC variants also outperform the best OpenMP C variant by up to a third, with *no* SaC source code modifications. Preserving an algorithm's abstract expression during optimization opens the door to generation of radically different code for different architectures.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

Keywords parallelism, readability, expressiveness, HPC, algorithms, APL, SAC, functional array languages

1. Introduction

Functional array languages, such as APL, SISAL, J, and SaC, being designed for the computer between our ears, offer considerable ex-

pressive power, compared to scalar-oriented, imperative languages, such as C and Java. [10] Indeed, Turing Award winner Kenneth E. Iverson, the inventor of APL, was adamant that programming is about communication of ideas, saying *what* is to be done, rather than *how* to do it.[15, 17] He told a parable of a parent asking a child for assistance. The array language approach is: "Please bring me a good apple from the basket." The imperative language approach is: "Pick an apple from the basket. If it is a good apple, bring it to me. Otherwise, see if there is another apple in the basket, and if so, repeat these steps until you get a good apple or run out of apples." [1] He demonstrated the power of applying natural language concepts to computing by summing the elements of a vector. In APL or J, the sum is merely $+/v$, in which an adverb, *reduce* ($/$), accepts a verb, *add* ($+$), as a left argument, creating a derived verb (sum), as its result. When sum is applied to a vector, it inserts the verb among the elements of the vector, then evaluates the resulting expression. Contrast this with its long-winded C equivalent:

```
int i;
double z = 0.0;
for( i=0; i<N; i++) z = z + v[i];
```

APL's terseness renders its meaning immediately obvious, whereas the C code requires careful reading just to understand the intent of the algorithm, and very meticulous reading to ensure that it does not contain any subtle errors, *e.g.*, the array having N+1, rather than N elements.

In APL, application across the entire vector is implicit: there is no need for an auxiliary variable, N, to be dragged along, to specify the shape of v, nor do we have to wonder whether N is the shape of v, or if N bears *any* relationship to v.

Generating efficient code for such expressions is challenging. [1, 3–6, 8, 26] Many existing approaches leverage semantic knowledge of primitive verbs, adverbs, and conjunctions. Generated code may contain alternative implementations, which are then dynamically dispatched.

Things are more challenging when generating parallel code, because array operation fusion is crucial, to prevent memory accesses from becoming a bottleneck. Since it is not feasible to support *ad hoc* fusion techniques for arbitrary combinations of many primitive verb, SaC takes a more generic approach, and does not provide *any* of the APL verbs, adverbs, or conjunctions as built-in constructs. Instead, it has one parallel array skeleton – the *with loop* – which suffices to define all APL constructs, as well as any desired new ones, enabling generic compiler technology to optimize arbitrary combinations of array constructs.

We now show that generic array programming has reached the point where it combines the expressiveness of natural language and

---

[1] Some of Ken's insights, often both educational and humorous, can be seen at: http://keiapl.org/anec/.

mathematics with higher performance than that of hand-optimized imperative languages. Moreover, that same expressiveness facilitates efficient parallel execution, with no tuning effort on the part of the user, as we shall now see.

2. Relaxation

When generating code for GPUs, even rather well-known codes benefit from unusual algorithmic specifications. One such example is *relaxation*, a common algorithmic pattern used in image processing and in numerical codes for approximating partial differential equations. For the sake of simplicity, we look at one-dimensional relaxation, in which a vector is recomputed by replacing a non-boundary element by the arithmetic mean of its left and right neighbor elements, while boundary elements remain unchanged. A straightforward formulation of the kernel code in C is `ifc.c`:

```
for( j=0; j<N; j++) {
    if(0==j) {
        res[j] = v[j];
    } else if((N-1)==j) {
        res[j] = v[j];
    } else {
        res[j] = (v[j-1] + v[j+1])/2.0;
    }
}
```

This kernel can be expressed in a more functional, but no more efficient, manner, as `condc.c`:

```
for( j=0; j<N; j++) {
    res[j] = (0==j)       ? v[j] :
             ((N-1)==j) ? v[j] :
               (v[j-1] + v[j+1])/2.0;
}
```

In SaC, we can express the computation as three data-parallel partitions, this way, as `hands.sac`:

```
res = with {
        ([0]    <= [j] < [1]) : v[j];
        ([1]    <= [j] < [N-1]) :
          (v[j-1] + v[j+1])/2.0;
        ([N-1] <= [j] < [N]) : v[j];
    } : modarray( v);
```

The result, `res`, is computed from the vector `v` by the boundary elements, (at index positions `[0]` and `[N-1]`), and by the arithmetic mean of the neighboring elements `v[j-1]` and `v[j+1]` elsewhere.

This formulation produces excellent performance on a traditional shared-memory system, but when executed on a GPU, its performance is suboptimal.[11] This is because GPUs are inherently tuned to a Single-Instruction-Multiple-Threads (SIMT) style. The following algorithm, `conds.sac`, similar to `condc.c` above, turns out to perform much better on a GPU:

```
res = with {
        ([0] <= [j] < [N]) :
          (0==j)       ? v[j] :
          ((N-1)==j) ? v[j] :
            (v[j-1] + v[j+1])/2.0;
    } : modarray( v);
```

Instead of performing two distinct computations, this algorithm applies one computation to all elements, at the price of a two-fold nest of conditionals. This is beneficial on a GPU, but on a shared-memory system, this trade-off comes at a high price, so we decided to investigate architecture-independent algorithms.

```
TD←{(1↑ω),(((2↓ω)+¯2↓ω)÷2.0),¯1↑ω}
ROT←{N←ρω
     m←(0=ιN)∨(N-1)=ιN
     (m×ω)+(~m)×((1⌽ω)+¯1⌽ω)÷2.0}
```

Figure 1. Two Relaxation Algorithms in APL

3. Abstract Expressionism

We chose to investigate an APL programming style. In APL and J, a common approach in algorithm design is to compose abstract expressions from array-based verbs (*e.g.*, addition and multiplication), adverbs (*e.g.*, reduction), and conjunctions, (*e.g.*, inner product). [16] This eliminates the need for many variables and lends itself to being highly readable, compared to equivalent programs written in an imperative style. As we shall see, that *abstract expressionist* style lends itself to creating SIMD parallel code, usually with little or no effort on the part of the user.

It is usual, in APL-style programming, to replace control flow by data flow, frequently through the use of Boolean arrays. For example, if every employee in a company who earned less than $10,000 a year was to be given a $500 raise, that would typically be expressed in an imperative language using control flow. In APL, Booleans are merely the subset $[0, 1]$ of the integers, so the algorithm could be written as:

```
pay ← pay + (pay<10000)×500
```

We expressed the relaxation algorithm using the same technique, replacing control flow by data flow, ending up with two distinct APL algorithms, shown in Figure 1. `TD` uses structural array operations, *take* (↑) and *drop* (↓) to decompose the vector and operate on its parts, then uses *catenate* (,) to reassemble the results. `ROT` generates a Boolean mask, m, of the same shape as the vector, with ones at both ends and zeros elsewhere. We then perform both computations and combine them under control of the mask:[2] We also wrote a SaC-based variant that uses array shifting, shown here:

```
m = (0 == iota(N)) | ((N-1) == iota(N));
res = (tod(m) * v) + tod(!m) *
 ((shift([1],v) + shift([-1],v)))/2.0;
```

Operation of array `shift` is best shown by example. `shift([2], [ 0,1,2,3,4])` produces `[0,0,0,1,2]`. The `tod()` verb coerces the Boolean array to double-precision. Note that the algorithm has no loops or conditionals, in the imperative sense, and memory management code is absent.

Abstract programming styles are easy to teach to domain specialists, children, artists, and others who have no background in traditional scalar-oriented languages, and likely have little or no interest in learning them. This is because they reflect natural language. Furthermore, common coding errors, such as off-by-one loop counts or array bounds violations, which might occur in the C code shown in Section 1, and so forth, are less likely to happen. Finally, because code is functional and mathematically expressive, an algorithm can be communicated to others easily and quickly.

4. Corresponding C Implementations

In order to put the performance of the algorithm into an absolute context, we now look at `condc.c`, a direct transliteration of the SaC code into C.

```
#pragma omp parallel for
```

---

[2] Iverson knew the power of such operations; his 1962 book includes a *mask* verb that does exactly what we want, with less work. [15]

```
for( j=0; j<N; j++) {
  res[j] = (0==j)     ? v[j] :
           ((N-1)==j) ? v[j] :
             (v[j-1] + v[j+1])/2.0;
}
```

An openMP pragma enables parallel execution of this embarrassingly parallel loop; an alternative version, `handc.c`, is simpler:

```
#pragma omp parallel for

for( j=1; j<N-1; j++) {
  res[j] = (v[j-1] + v[j+1])/2.0;
}

res[0]   = v[0];
res[N-1] = v[N-1];
```

In this version, boundary elements are initialized when vectors `v` and `res` are allocated. While this could be considered a mildly unfair comparison favoring the C implementation, it is, in fact, not much of an issue, as our benchmark iteratively applies the relaxation steps, thereby minimizing any difference.

5. Experimental Setup

We conducted performance tests on a multi-core (8 cores, 4 FPUs) AMD FX-8350 (Piledriver), with 32GB of DRAM, running at 4013MHz, under Ubuntu 14.04LTS, Build #18605 of the `sac2c` compiler, and gcc (Ubuntu 4.8.2-19ubuntu1). We measured performance using Ubuntu's `/usr/bin/time` command, with thread counts varying from 1–8.

5.1 Theoretical Peak Performance Rate

Our AMD CPU, running at 4GHz, can issue one 4-wide double-precision floating-point instruction on each clock cycle, giving a theoretical peak performance rate of just over 16GFLOP/s for a single core, and about 64GFLOP/s over all four FPUs. However, Agner says "...Piledriver ...has a throughput of one 256-bit store per 17–20 clock cycles".[9] That is 4–5 clock cycles per array element, so we end up with system-wide peak performance of about 12–16GFLOPs, due to memory subsystem bottlenecks.

6. Single-thread Performance

The relaxation benchmark performance rates for SaC and C are shown in Figure 2, with curve names as given in Section 2. The 1-thread column of that Figure gives single-thread performance. We were surprised to see that SaC single-thread performance is usually somewhat faster than that of equivalent native C algorithms, despite the overhead of SaC codes having been compiled in multi-thread mode.[3]

What we found more remarkable is that four SaC algorithms – `hands`, `rotates`, `shifts`, and `takedrops` – have essentially the same performance, despite very different stdlib coding styles used for them, as can be seen in Figure 6 and Figure 7. This performance level arises from array-oriented optimizations in the SaC compiler, generating inner loops for the latter three abstract benchmarks that are *identical* to the first, hand-optimized one. Generating high-performance code from abstract, mathematical compositions has been a major goal of the functional array language community since before the time of the SISAL project. [1, 4, 5, 20] The fact that we have achieved this identical performance with radically

different algorithmic styles is pleasing, even though our example is relatively simple.

7. Multi-thread Performance

Turning to multi-thread performance, we see that `hands`, `rotates`, `shifts`, and `takedrops` all scale decently, with performance approaching the theoretical platform maximum of 16GFLOPs. The best-performing C code is `handc`, which starts off about the same as SaC , but then scales poorly, ending up at about 11GFLOPs.

Conditional-based codes – `condc`, `conds`, `ifc`, and `ifs` – all exhibit lackluster performance, peaking at just over 7GFLOPs, for both SAC and C, suggesting pipeline problems. The fact that the compilers map IF-statements into conditional expressions is evident from the matching performance of `conds` and `ifs`, and of `condc` and `ifc`.

8. Code Changes for Parallel Execution

One major difference between the imperative (C, JAVA) and functional array language communities (APEX, J, SaC , SISAL) is the approach taken to achieve parallel execution. In C, at best, pragmas must be sprinkled throughout the program, and prolog code, such as that in Figure 4 and Figure 3, included in the application. In the functional array language community, parallelism is achieved with *no* source code changes.

That a SaC application need not contain target-system-specific tuning code has several benefits: first, code generation for different architectures is left to the compiler tool-chain, thereby avoiding any source code changes, possible code review, and recertification, in order to operate on, say, a GPU, rather than on a multi-core system. Second, the mathematical clarity of algorithms is preserved, with no clutter to mask the intent of the computation or, worse yet, to introduce faults into an otherwise correct algorithm. Finally, parallel performance problems such as the failure to privatize a single variable, thereby causing "parallel slowdown", can not occur. SaC makes no attempt to preserve ordering of operations on real numbers, so non-associativity problems can arise, as they do in any parallelization scheme.

9. Code Volume

Another benefit of programming with abstract expressions is that algorithms are shorter and, therefore, more readable. The APL community has known this for more than fifty years, yet is regularly derided for it, despite the fact that effective communication among humans must be performed with as little verbiage as possible.

One of us (Bernecky) regularly asserts that code that is not there can not break. Assuming that this is true, and that code fault rates are proportional to code volume, what does our benchmark tell us about this? Well, admittedly, nothing, but it does suggest that you might prefer to ride in a train whose controlling software is written in a functional array language, rather than an imperative language.

10. Flexibility

Unlike languages in which parallelism is achieved through hand-tuned libraries (*e.g.*, BLAS), or through built-in language constructs (*e.g.*, APL, J), SaC offers great flexibility, in that libraries are made from SaC code: new libraries can be created on demand and, because they are exposed to the compiler's optimizations, perform as well as if they had been written as inline code.[2, 13]

11. Communicating Algorithms

Functional array languages lend themselves to formal theorem proving, as a number of people have shown. [14, 16, 17, 22] In
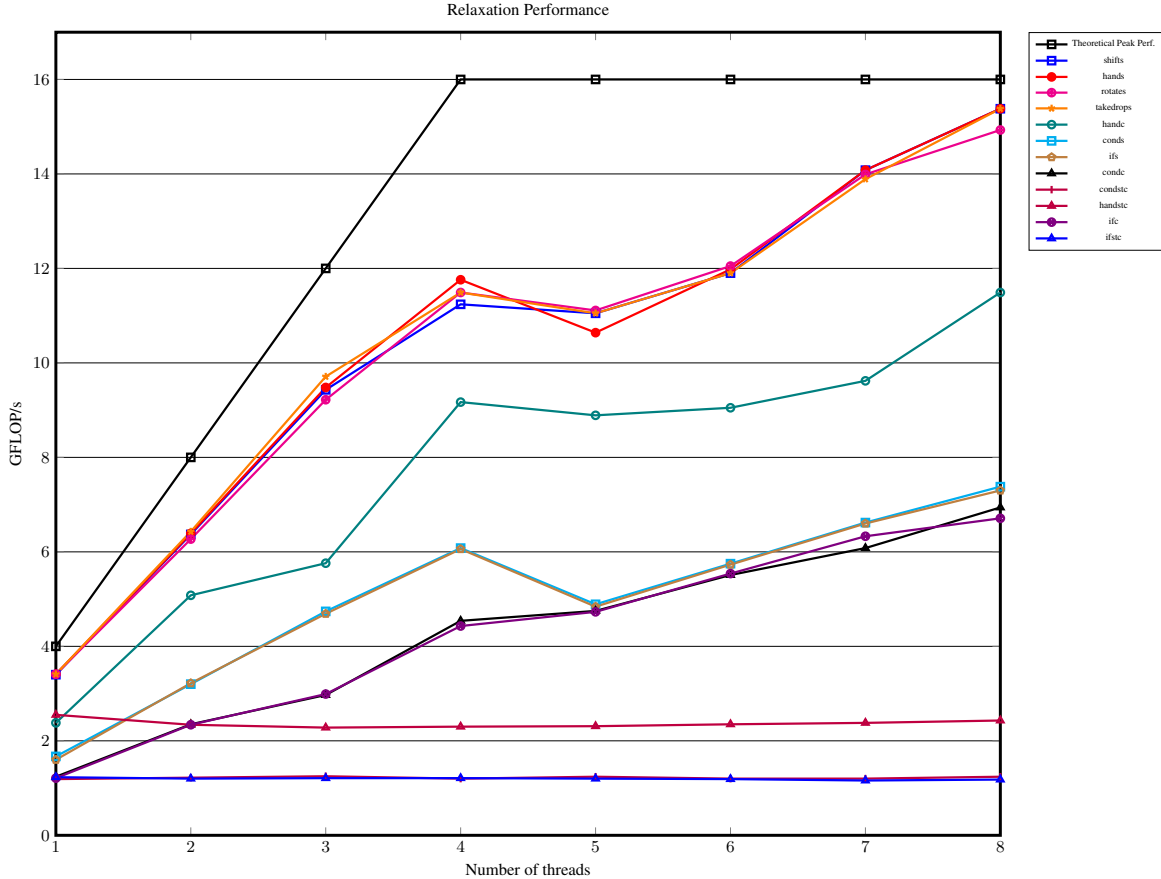
---

[3] In an effort to determine why this is so, we timed the SaC algorithms with the SaC private heap manager disabled. This increased execution times by about 12%, but that is still more than 10% faster than the equivalent single-thread C code.

Figure 2. Relaxation Benchmark Performance

this sense, we assert, without proof, that those languages are superior to imperative languages, because they facilitate formal proofs and communication of an algorithm's underlying ideas. Iverson called APL an *executable analytical notation*, reflecting its utility as both a mathematical notation, and as a programming language.

Communication of algorithms is eased by the fact that functional array language programs are generally much shorter than those written in imperative languages. Furthermore, if the number of faults in a program is proportional to its size, then terse programs have another edge over long ones.

Functional array languages are popular in financial markets, because people dealing in extremely large amounts of money have this idea that people writing programs that will become *You Bet Your Company* applications should be able to convince their colleagues that the applications will work properly when released. The rapid time-to-solution provided by functional array languages gives those companies a leg up over their competition. [1]

12. Related Work

We note three areas of related work: optimization/code generation from DSLs, code generation for generic array programs, and sophisticated optimization/code generation from imperative languages, such as C or Fortran.

Since each of these areas is vast, and we have limited space, we focus on a few representative works that demonstrate how these three areas are steering towards an increasingly common ground. In DSL approaches, Spiral [23] demonstrates nicely how special-

ized knowledge of signal processing can be leveraged to generate excellent runtime performance from very high-level specifications. Recently, Spiral is attempting to generalize the approach towards more generic linear algebra expressions [25], *i.e.*, exactly the subclass of array programs of which this paper is concerned. A lot of effort has been put into compilation of high-level array languages, such as compilation of APL[6, 12] and Matlab[19]. Most of these approaches are, in principle, very similar to the work presented here, but the generality of the source language often inhibits a similar effectiveness. Recent work on the identification of semantical difficulties[7] indicates that, ultimately, similar performance levels might be achievable.

In classical high-performance computing, advances over the last few years have tried to bridge the gap between human-readable specifications and highly-performing codes. A lot of work has been done in the context of the polyhedral model [18, 24], where sophisticated code manipulations can transform naive codes into highly sophisticated ones. Another example is the build-to-order BLAS project [21], where the authors demonstrate how elaborate code generation techniques can outperform highly tuned library codes. While these works are not primarily concerned with high levels of abstraction, they implicitly get closer to a point where more abstract program specifications no longer impede overall performance.

13. Summary

We claim that algorithms written in functional array languages are shorter, faster in both serial and parallel modes, and are easier to

read and to communicate than when written in C. We did not prove that communication is easier, but we provided some evidence to support that assertion. We showed that the relaxation benchmark is shorter in SaC than in C, even without any OpenMP constructs, and that top-performing abstract expressionist algorithms can be written in a purely functional way.

We also showed that *all* SaC-based variants outperform the C-based equivalents, and that the functional variants – `hands`, `rotates`, `shifts`, `takedrops` – share top honors for top performance because, despite their radically different coding styles, they all generate identical inner loops.

We assert that abstract expressionist programming is likely to result in fewer bugs than other styles, because less code implies less chance of errors. The utter absence of *any* explicit memory allocation and deallocation, combined with a call-by-value semantics on arrays, also reduce the frequency of program faults. Similarly, the absence of all explicit parallel constructs, such as pragmas, eliminates another possible source of code faults and poor performance, particularly when cross-architecture support is required.

In our experience, domain specialists prefer functional array languages for these reasons and others. Such languages, used in conjunction with an abstract expressionist programming style, comprise a rising tide that will raise all ships.

Acknowledgments

References

[1] R. Bernecky. APEX: The APL Parallel Executor. Master's thesis, University of Toronto, 1997.

[2] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, June 2002. ISSN 0098-3500. URL http://doi.acm.org/10.1145/567806.567807.

[3] T. Budd. *An APL Compiler*. Springer, 1988.

[4] D. Cann. Retire Fortran? A Debate Rekindled. *Communications of the ACM*, 35(8):81–89, 1992.

[5] D. Cann. *The Optimizing SISAL Compiler: Version 12.0*. Lawrence Livermore National Laboratory, LLNL, Livermore California, 1993. Part of the SISAL distribution.

[6] W.-M. Ching. An APL/370 compiler and some performance comparisons with APL interpreter and FORTRAN. *ACM SIGAPL Quote Quad*, 16(4):143–147, July 1986.

[7] A. W. Dubrau and L. J. Hendren. Taming matlab. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 503–522, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6. . URL http://doi.acm.org/10.1145/2384616.2384653.

[8] J. Feo. *Arrays in Sisal*, chapter 5, pages 93–106. Arrays, Functional Languages, and Parallel Systems. Kluwer Academic Publishers, 1991.

[9] A. Fog. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*, 2010.

[10] 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. .

[11] J. Guo, J. Thiyagalingam, and S.-B. Scholz. Breaking the gpu programming barrier with the auto-parallelising SAC compiler. In *6th Workshop on Declarative Aspects of Multicore Programming (DAMP'11), Austin, USA*, pages 15–24. ACM Press, 2011.

[12] A. W. Hsu. Co-dfns: Ancient language, modern compiler. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, pages 62:62–62:67, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2937-8. . URL http://doi.acm.org/10.1145/2627373.2627384.

[13] *International Standard for Programming Language APL*. International Standards Organization, ISO N8485 edition, 1984.

[14] K. Iverson. *Algebra: an Algorithmic Treatment*. APL Press, 1976.

[15] K. E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., 1962.

[16] K. E. Iverson. Programming style in APL. In *APL Users Meeting 1978*. I.P. Sharp Associates Limited, 1978.

[17] K. E. Iverson. Notation as a tool of thought. *Commun. ACM*, 23(8), Aug. 1979.

[18] M. Kong, A. Pop, L.-N. Pouchet, R. Govindarajan, A. Cohen, and P. Sadayappan. Compiler/runtime framework for dynamic dataflow parallelization of tiled programs. *ACM Trans. Archit. Code Optim.*, 11(4):61:1–61:30, Jan. 2015. ISSN 1544-3566. . URL http://doi.acm.org/10.1145/2687652.

[19] X. Li. Mc2for: A tool for automatically translating MATLAB to FORTRAN 95. pages 234–243. IEEE, 2014. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6747175.

[20] L. M. R. Mullin. *A Mathematics of Arrays*. PhD thesis, Syracuse University, 1988.

[21] T. Nelson, G. Belter, J. G. Siek, E. Jessup, and B. Norris. Reliable generation of high-performance matrix algebra. *ACM Transactions on Mathematical Software*, 41(3).

[22] D. Orth. *Calculus in a New Key*. APL Press, 1976.

[23] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[24] J. Shirako, L. Pouchet, and V. Sarkar. Oil and water can mix: An integration of polyhedral and ast-based transformations. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, pages 287–298, 2014. . URL http://dx.doi.org/10.1109/SC.2014.29.

[25] D. G. Spampinato and M. Püschel. A basic linear algebra compiler. In *International Symposium on Code Generation and Optimization (CGO)*, pages 23–32, 2014.

[26] J. Weigang. An Introduction to STSC's APL compiler. In *APL89 Conference Proceedings*, pages 231–238. ACM SIGAPL Quote Quad, volume 15, 1989.

```c
if (argc<2) {
  printf( "provide num_threads!\n");
  exit;
} else {
  num_threads = atoi(argv[1]);
}
printf("num Threads: %d\n", num_threads);
omp_set_num_threads( num_threads);
kind = omp_sched_auto;
mod = 0;
omp_set_schedule( kind, mod);
```

Figure 3. OpenMP C prolog

```c
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
#define ITER 100000
int main(int argc, char *argv[]) {
  int i,j,N,mod,num_threads;
  double sum,el;
#ifdef OPENMP
  omp_sched_t kind;
#endif // OPENMP
  N = 100001;
  double *v = malloc(sizeof(double)*(N));
  double *res = malloc(sizeof(double)*(N));
  double *tmp;
  for( i=0; i < N; i++) {
    v[i] = 2.0;
    res[i] = 2.0;
  }
  v[N/2] = 500.0;
  res[N/2] = 500.0;
#ifdef OPENMP
#include "prolog.h"
#endif // OPENMP
  for( i=0; i<ITER; i++) {
#ifdef OPENMP
#pragma omp parallel for
#endif // OPENMP
#include "kernel.c"
    tmp = res;
    res = v;
    v = tmp;
  }
  sum = 0.0;
  for( j=0; j<N; j++) { sum += res[j]; }
  printf("C result is %f\n", sum);
  return( 0);
}
```

Figure 4. C benchmark code

```
use Array: all;
use StdIO: all;
#define ITER 100000
int main() {
  N = 100001;
  v = genarray( [N], 2d);
  v[N/2] = 500d;
  res = v;
  for( i=0; i<ITER; i++) {
#include "kernel.sac"
    v = res;
  }
  show( String::tochar("SaC result is:"));
  show( sum(res));
  return( 0);
}
```

Figure 5. SaC benchmark code

```
inline
double[*] take( int[.] v, double[*] array)
{
  shpa = _shape_A_( array);
  zr = _mul_VxS_( shpa, 0);
  offset = zr;
  vext = shpa;
  i = 0;
  while (_lt_SxS_( i, _sel_VxA_( [ 0 ],
                    _shape_A_( v))))
  {
    el = _sel_VxA_( [ i ], v);
    shpel = _sel_VxA_( [ i ], shpa);
    vext = _idx_modarray_AxSxS_( vext, i, el);
    val = _mask_SxSxS_( _lt_SxS_( el, 0),
                        _add_SxS_( shpel, el), 0);
    offset = _idx_modarray_AxSxS_( offset,
                i, val);
    i = _add_SxS_( i, 1);
  }
  shpz = _abs_V_( vext);
  res = with {
        (. <= iv <= .)
        {
        } : _sel_VxA_( _add_VxV_( offset, iv),
                array) ;
       } : genarray( shpz, zero( array) );
  return( res);
}
```

Figure 6. SaC stdlib take

```
inline
double[+] rotate( int dimension,
                  int count, double[+] A)
{
  max_rotate = _sel_VxA_( [ dimension ],
                          _shape_A_( A));
  count = NormalizeRotateCount( count,
                                max_rotate) ;
  offset = _modarray_AxVxS_( _mul_SxV_( 0,
              _shape_A_( A)),
              [ dimension ], count);
  slice_shp = _modarray_AxVxS_( _shape_A_( A),
                [ dimension ], count);
  result = with {
        (offset <= iv <= .)
        {
        } : _sel_VxA_( ( iv - offset) , A) ;
         } :
      modarray( A);
  result = with {
        (. <= iv < slice_shp)
        {
        } : _sel_VxA_((( _shape_A_( A)
                      - slice_shp) + iv) , A) ;
         } :
      modarray( result);
  return( result);
}
```

Figure 7. SaC stdlib rotate