Shonan Challenge for Generative Programming

Short Position Paper

Baris Aktemur
Ozyegin University, Turkey
baris.aktemur@ozyegin.edu.tr

Yukiyoshi Kameyama
University of Tsukuba, Japan
kameyama@acm.org

Oleg Kiselyov
oleg@okmij.org

Chung-chieh Shan
ccshan@post.harvard.edu

Abstract

The appeal of generative programming is "abstraction without guilt"[1]: eliminating the vexing trade-off between writing high-level code and highly-performant code. Generative programming also promises to formally capture the domain-specific knowledge and heuristics used by high-performance computing (HPC) experts. How far along are we in fulfilling these promises? To gauge our progress, a recent Shonan Meeting on "bridging the theory of staged programming languages and the practice of high-performance computing" proposed to use a set of benchmarks, dubbed "Shonan Challenge".

Shonan Challenge is a collection of crisp problems posed by HPC and domain experts, for which efficient implementations are known but were tedious to write and modify. The challenge is to generate a similar efficient implementation from the high-level specification of a problem, performing the same optimizations, but automatically. It should be easy to adjust optimizations and the specification, maintaining confidence in the generated code.

We describe our initial set of benchmarks and provide three solutions to two of the problems. We hope that the Shonan Challenge will clarify the state of the art and stimulate the theory and technology of staging just as the POPLmark challenge did for meta-theory mechanization. Since each Shonan Challenge problem is a kernel of a significant HPC application, each solution has an immediate practical application.

1. Introduction

Both the programming language (PL) and high-performance computing (HPC) communities have come to realize the importance of code generation, in particular, staging. Whereas PL theorists widely regard staging as the leading approach to making modular software and expressive languages run fast, HPC practitioners widely regard staging as the leading approach to making high-performance software reusable and maintainable ([3] and references therein). The recent NII Shonan Meeting [7] "Bridging the theory of staged programming languages with the practice of high-performance computing" gave a rare chance for PL researchers and the potential consumers of their work to meet each other.

[1] This slogan was coined by Ken Kennedy.

A particular area of current interest shared by PL and HPC researchers is how to use *domain-specific languages* (DSLs) to capture and automate patterns and techniques of code generation, transformation, and optimization that recur in an application domain [8, 11]. For example, HPC has created and benefited from expressive DSLs such as OpenMP directives, SPIRAL's signal processing language [8], and specialized languages for stencil computations and domain decomposition. Moreover, staging helps to build efficient and expressive DSLs because it assures that the generated code is correct in the form of precise static guarantees.

Alas, the communication between PL researchers working on staging and HPC practitioners could be better. On one hand, HPC practitioners often do not know what PL research offers. On the other hand, PL researchers often do not know how much HPC practitioners who write code generators value this or that theoretical advance or pragmatic benefit – in other words, how the HPC wish list is ranked by importance. Therefore, at the Shonan Meeting, we sought real-world applications of assured code generation in HPC that would drive PL research in meta-programming.

Specifically, we decided to collect examples of what staging *should* express:

- a high-level specification that a high-performance programmer *should* be able to write easily and
- a low-level efficient implementation that is known today but typically optimized by hand tediously.

In response to each such challenge statement, staging researchers can show how to carry out the same optimizations and derive a similar efficient implementation automatically from a similar simple specification. The benefits of such an interaction are bidirectional:

- HPC practitioners would obtain the efficient implementation without writing tedious code by hand.
- PL researchers would obtain real-world applications to enhance the theory and technology of staging.

§2 below details the selection criteria for the challenge problems and the evaluation criteria for their solutions. §3 introduces the initial set of challenges collected by Kenichi Asai during the meeting. To date, several solutions have already been submitted for several challenges. §4 and §5 discuss solutions for the stencil and hidden Markov model benchmarks, respectively. The submitted solutions also show the still open problems in the theory of staged computation, in particular, ensuring that the generated code is well-typed and well-scoped. Bugs do occur that cause the generated code to contain unbound variables, and the culprit is not trivial to find.

A public mailing list `http://groups.google.com/group/stagedhpc` and a GitHub repository `https://github.com/StagedHPC/shonan-challenge` have been set up for the Shonan Challenge. Please submit new problems and new solutions.

2. Design of the benchmark

In this section we motivate the selection of the Shonan Challenge problems and the criteria to evaluate solutions. In particular, we describe how the Shonan Challenge differs from existing HPC benchmarks.

The HPC community has developed many benchmark suites, such as

NAS parallel benchmarks[2] a set of pervasively common computational codes, designed to evaluate the performance of parallel supercomputers;

Sandia Mantevo suite[3] kernels and parameterizable applications to mimic the performance of real production codes;

LLNL Sequoia suite[4] simplified versions of real-life HPC applications, selected as tests of parallel efficiency, single-processor performance, performance of various aspects of Message Passing Interface (MPI), as well as of quality of compiler optimizations;

DARPA HPC Challenge[5] synthetic benchmarks to measure floating point performance, memory bandwidth, latency and other aspects of supercomputer performance.

Most of these benchmarks are intended for measuring hardware performance or the quality of MPI implementations and of compiler optimizations. Since the benchmarks contain common HPC algorithms, some of them could also be used for evaluating new languages or programming models. They have not been specifically designed for testing how easy it is to write or generate these codes or adjust them for a new platform. For example, micro-benchmarks contain fixed low-level code not intended for optimization. Some other benchmarks are scaled-down versions of real-world applications like molecular dynamics simulations, whose specifications require extensive background to understand and whose implementations are significant research problems.

2.1 Problem selection criteria

For the Shonan Challenge, we want problems that are easy to describe at a high level, so that many PL researchers can contribute their solutions. At the same time, to keep the problems relevant to HPC applications, we want the problems to pose a variety of commonly encountered challenges for code generation: generating code with arbitrarily many new bindings, with simple and complex loop transformations (splitting, merging, unrolling), with the elimination of temporary arrays.

We have selected the following format for the Challenge problems:

1. A brief description of the problem;
2. A high-level specification of what to compute;
3. A high-level description of the desired optimizations;
4. Sample optimized code (typically hand-written) along with sample unit tests;
5. Variations of the problem: what other optimizations are worth trying, which parameters could be changed (e.g., unrolling factors), what related problems can the same technique apply to.

We use this format to describe the submitted problems in §3 and we request new submissions in this format.

[2] `http://www.nas.nasa.gov/publications/npb.html`

[3] `https://software.sandia.gov/mantevo/`

[4] `https://asc.llnl.gov/sequoia/benchmarks/`

[5] `http://icl.cs.utk.edu/hpcc/`

2.2 Solution evaluation criteria

A natural evaluation criterion for an HPC program is performance. After all, HPC practitioners are often viewed by outsiders as (ab)using assembly and low-level tricks to squeeze out the last bit of performance. However, time and again observations of real-life HPC projects contradict that naive view: "Since one of the project goals is to develop algorithms that will last across many machine lifetimes, it's not seen as productive to try to maximize the performance on any particular platform. Instead, code changes are made that will improve the performance on a wide range of platforms" [5]. A desire for *portable performance* was likewise expressed by the participants of the Shonan Meeting.

Therefore, respondents to the Shonan Challenge are not required to benchmark their code on a supercomputer. Likewise, the responses are not required to match the hand-written code *verbatim*. Of course, the responses to the challenge must compute the correct results, in particular, pass the unit tests. Any response should implement all the optimizations desired in the challenge. The solutions will be further evaluated by:

- how well they support the extensions, variations and generalizations;
- which correctness assurance of the generated code they provide;
- how easy are they to use by domain or HPC experts:

  the cost of entry how easy it is to obtain the programming system that implements the optimizations on a variety of platforms and how easy it is to learn to use the system or extend the set of optimizations;

  the ongoing overhead how easy it is to apply the automatic optimizations compared to writing the code by hand.

3. Shonan Challenge problems

This section lists sample challenges.

3.1 Complex number representation

Reiji Suda, an HPC expert, posed a challenge of accommodating changes to data layout. Transforming array-of-structures to structure-of-arrays is one of the most profitable optimizations for massively parallel systems such as GPU [13].

High-level specification Compute the product of two complex vectors b_j and c_j of length N element-wise:

$$a_j = b_j \star c_j \quad j = 0..N-1$$

where the complex multiplication $x \star y$ yielding z is defined as

$$\Re z = \Re x \, \Re y - \Im x \, \Im y$$
$$\Im z = \Re x \, \Im y + \Im x \, \Re y$$

and $\Re x$ and $\Im x$ are, respectively, the real and the imaginary parts of the complex number x.

Desired optimizations A complex number is commonly represented as a structure (in C) with two fields for the real and imaginary parts of a number:

```
typedef struct { double r,i; } complex;
```

A complex vector then is an array-of-structures. The function to compute the complex product of two vectors `b` and `c` with the result in `a` has the signature:

```
void aos_cmul(int n, complex a[],
                     const complex b[], const complex c[])
```

(see the Github repository for the code.)

An alternative representation for a complex vector is a structure of two real-valued vectors, collecting the real and the imaginary parts of the complex vector and in the separate arrays:

```
typedef struct {double *rv; double *iv} complex_vector;
```

This structure-of-arrays representation is especially suitable for vector computers and GPU. The complex vector multiplication will have the signature

```
void soa_cmul(int n, complex_vector a,
                     complex_vector b, complex_vector c)
```

Challenges

1. Generate code for aos_cmul and soa_cmul.
2. Generate data layout transformation functions

   ```
   void soa_to_aos(int n, complex a[], const complex_vector b)
   void aos_to_soa(int n, complex_vector a, const complex b[])
   ```

3. Suppose we are given the source code for aos_cmul but our complex vectors are represented as structures-of-arrays. We can still use aos_cmul if we apply soa_to_aos to the input vectors and aos_to_soa to the output vector. Incorporate the layout transformation functions into the given code for aos_cmul so to eliminate intermediary arrays. Preferably, the resulting code should match soa_cmul written by hand.

Variations

1. Accommodate other representations of a complex array:
   - double a[2*N] with the first half of the array a containing the real part of the vector, and the last half containing the imaginary part;
   - A complex vector of length N as a $2 \times N$ or $N \times 2$ real matrix

   and generate the code for the cmul functions and for the data layout transformations.
2. Generate code for other complex vector operations (for example, computing $d \star b_j + c_j$ where d is a scalar). In short, write a DSL for vector operations, supporting all mentioned data layout formats and a number of vector operations.

3.2 Sparse vector representation

Vectors can be represented using various formats to save memory space depending on their sparsity level. A vector operation has different implementations and optimization opportunities based on the representation. It is typical to switch among different sparse formats for different program runs [13]; the flexibility to quickly adjust the code assuring correctness and performance is important. Reiji Suda submitted a challenge[6] that requires automatic optimization of code specified using high-level operations. Although formulated for vectors, the challenge extends to sparse matrices as well.

High-level specification A standard dense vector, assuming double-precision values, is represented by an array (double*) to store the elements and an integer for the length. BLAS libraries provide a function named daxpy, with the signature below, that computes the generalized vector-vector addition $z \leftarrow a * x + y$ where a is a scalar. The function can be implemented using a straightforward for-loop.

```
void daxpy(int n, double *z, double a, double *x, double *y)
```

[6] https://github.com/StagedHPC/shonan-challenge/tree/master/problems/spvec

A sparse vector is represented by the following structure:

```
typedef struct {
  int n;          // the length of the vector
  int nnz;        // the number of the non-zero elements
  int *idx;       // the indices of the non-zero elements
  double *val;    // the values of the non-zero elements
} spv;
```

and can be converted to a dense vector by the following function (assuming no two elements of v.idx are the same):

```
void spv2vec(double *u, spv v) {
  for (int i=0; i< v.n; i++)
    u[i] = 0.0;

  for (int i=0; i< v.nnz; i++)
    u[v.idx[i]] = v.val[i];
}
```

Using the conversion function above, we can define the variants of daxpy where one or two input vectors are sparse.

```
void daxpspy(int n, double *z, double a, double *x, spv y) {
  double u[y.n];
  spv2vec(u, y);
  daxpy(n, z, a, x, u);
}

void daspxpy(int n, double *z, double a, spv x, double *y)
  // similarly

void daspxpspy(int n, double *z, double a, spv x, spv y) {
  double u[x.n], v[y.n];
  spv2vec(u, x);
  spv2vec(v, y);
  daxpy(n, z, a, u, v);
}
```

HPC programmers prefer specifying data format adjustments by inserting transformation functions as above. The programmers then proceed to hand-optimize the code, to eliminate extra function calls and temporary arrays and to fuse loops.

Desired optimizations The implementations of daxpspy, daspxpy and daspxpspy use temporary arrays to compose spv2vec and daxpy. The optimizations must remove these intermediary arrays.

The function daspxpy is subject to a second optimization: detect when z and y are the same array and generate code that updates only x.nnz elements of z (x.nnz is the number of non-zero elements of x).

There are many other sparse representations possible. For example, if the vector's non-zero elements appear in (sufficiently large) groups of contiguous elements, the format below can be used to further reduce the required memory. The function to convert to dense format is also given below.

```
typedef struct {
  int n;          // length
  int ng;         // number of groups
  int *size_g ;   // sizes of groups
  int *iidx_g :   // initial index of groups
  double **val;   // values
} spv_g;

void spv_g2vec(double *z, spv_g x) {
  for (int i=0; i< x.n; i++)
    z[i] = 0;

  for (int i=0; i< x.ng; i++)
    for (int j=0; j< x.size_g[i]; j++)
      z[j + x.iidx_g[i]] = x.val[i][j];
}
```

Even more space can be saved by removing the group size array if the group sizes are (multiples) of the constant BLOCKSIZE:

```
typedef struct {
  int n;            // length
  int ng;           // number of groups
  int *iidx_g :     // initial   index of groups
  double **val;     // values
} spv_b;

void spv_b2vec(double *z, spv_b x) {
  for (int i=0; i< x.n; i++)
    z[i] = 0;

  for (int i=0; i< x.ng; i++)
    for (int j=0; j< BLOCKSIZE; j++)
      z[j + x.iidx_g [i]] = x.val[i][j];
}
```

A separate daxpy function can be written for when x and/or y is in one of the formats above. Again, temporary array allocations must be optimized away. Furthermore, the inner loop in spv_b2vec must be fully unrolled to eliminate loop-nesting.

Finally, a vector v can be decomposed into the sum of multiple other vectors to allow combination of various representations, e.g. $v = v_b + v_s$ where v_b is in spv_b and v_s is in spv format. The same optimization requirements apply when a composed representation is used.

Challenge The main challenge is to automatically remove the temporal array definitions as explained above. The challenge can be extended to include additional operations. For instance, given a function vec2spv that converts dense vector to sparse format, the daxpy of sparse vectors resulting in a sparse vector is

```
spv spdaxpy(double a, spv x, spv y) {
  double xx[x.n], yy[y.n], zz[x.n];
  spv2vec(xx, x);
  spv2vec(yy, y);
  daxpy(zz, a, xx, yy);
  return vec2spv(x.n, zz);
}
```

Again, temporal array allocations should be removed if possible. Depending on whether the input vectors are nearly dense or very sparse, a different algorithm to construct the output vector can be used. It can be assumed that the indices of sparse vectors are sorted, because the challenge becomes more complicated without this assumption.

Inner product and element-wise multiplication of sparse vectors can be defined similarly by composing conversion and dense-vector operations. The challenge can be extended to include these operations as well.

3.3 Stencil

The main HPC challenge today is to overcome the memory bottleneck, reducing the amount of time the CPU (local node) spends waiting for data. The speed disparity between the CPU and the memory subsystem is often characterized by the *byte/flop* (B/F) ratio: the number of bytes moved from or to the main memory during a computation divided by the number of floating-point operations. The challenge of the HPC code optimization is to reduce the B/F ratio *and* the amount of needed local memory. Takayuki Muranushi, an astrophysicist whose work depends on solving partial differential equations (PDEs) on huge meshes, submitted a challenge regarding typical PDE solvers and finite-element algorithms.

High-level specification Signal processing or finite-element algorithms can often be stated as element-wise computations on shifted arrays. The following is the running example in the challenge. It is depicted in Figure 1(a).

$$\vec{w} = \vec{a} \times \mathcal{S}^1 \vec{a}$$
$$\vec{b} = \vec{a} - \vec{w} + \mathcal{S}^{-1} \vec{w}$$

Here $\vec{a}$ is a global input array, $\vec{b}$ is a global output array, and $\vec{w}$ is a working array. The operation $\mathcal{S}^n$ shifts the argument array by n (to the left, if n is positive). All arithmetic operations on arrays (addition, subtraction, even multiplication) are element-wise. Global arrays are shared or distributed throughout a supercomputer; reading or writing them requires accessing off-the-chip memory or inter-node communication.

Desired optimizations The naive implementation, neglecting for now edge elements, can be written in C as

```
for (i=0; i < N-1; i++)
  w[i] = a[i] * a[i+1];
for (i=1; i < N-1; i++)
  b[i] = a[i] - w[i] + w[i-1];
```

Assuming w is a local array, the first loop reads 2(N−1) elements from the global array a and does N−1 floating-point multiplications.[7] The second loop reads N−2 elements from a, writes N−2 elements to b and performs 2(N−2) floating-point operations. With 8-byte floating-point values, the B/F ratio is 32 to 3. Current supercomputers can sustain a B/F ratio of about 1 to 2. Therefore, the naive implementation will run an order of magnitude slower than the peak performance, because global memory cannot keep up with the CPU. The slow-down will likely be bigger since for large N the array w will not fit into the local memory.

After the array computation finishes, the output array b becomes the input array for the next 'time step', and the computation continues. To reduce the B/F ratio, we can unroll this outer loop and compute two time steps at once:

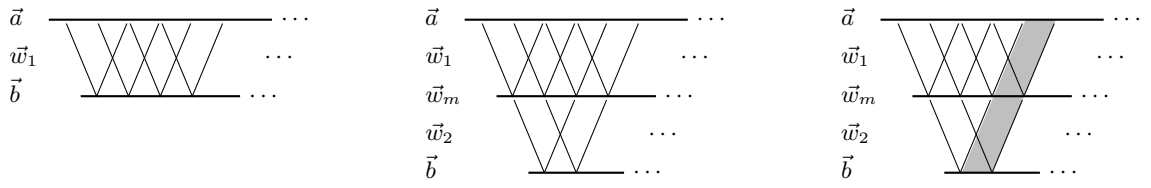
$$\vec{w}_1 = \vec{a} \times \mathcal{S}^1 \vec{a}$$
$$\vec{w}_m = \vec{a} - \vec{w}_1 + \mathcal{S}^{-1} \vec{w}_1$$
$$\vec{w}_2 = \vec{w}_m \times \mathcal{S}^1 \vec{w}_m$$
$$\vec{b} = \vec{w}_m - \vec{w}_2 + \mathcal{S}^{-1} \vec{w}_2$$

This unrolling is depicted in Figure 1(b). The naive implementation of the twice-unrolled loop body has a B/F ratio of 32 to 6 – a two-fold improvement over the ratio before of 32 to 3. However, the implementation requires three times more local memory.

It is easy to see that computing one value b[i] requires only the values within a small window around a[i] – the *stencil*. The intermediate array w can be avoided then. To compute the next element of the output array b[i+1], we 'slide' the stencil over a, reusing all but one of the elements from the previous stencil. Building the shifted stencil requires then a single reading from the global array a. This computation is depicted in Figure 1(c). Applying these optimizations manually to the two-time-step algorithm gives the following code:

```
for (i = 2; i < N-2; ++i) {
   a_1 = a_2; a_2 = a[i+2];
   w1_0 = w1_1; w1_1 = a_1 * a_2;
   wm_0 = wm_1; wm_1 = a_1 - w1_1 + w1_0;
   w2_m1 = w2_0; w2_0 = wm_0 * wm_1;
   b[i] = wm_0 - w2_0 + w2_m1;
}
```

[7] We assume that the elements of a are not cached, as is the case for GPUs or Cray XMT. If the read elements of the array are cached, the first loop accesses the global array N times. Since the array is presumed long, when the second loop begins the needed elements will already be evicted from the cache.



(a) high-level specification (b) twice-unrolled loop (c) sliding stencil

Figure 1. The running example for the stencil problem

We assume that the stencil, formed by local variables a_1, a_2, etc., is appropriately initialized before the beginning of the loop. Each iteration of the loop reads one floating-point value, writes one floating-point value, and executes 6 floating-point operations. So the B/F ratio is 16 to 6 – without using any local arrays. The code is expected to run closer to the peak speed of the supercomputer.

Challenge and variations The basic challenge problem is to automatically apply the stencil optimization and generate local-array-free code. The result does not have to look exactly like the hand written code above, but it has to give the same outputs given the same inputs, use no local arrays, perform only a single reading and single writing to global arrays and 6 floating-point operations per iteration (the B/F ratio must be 16 to 6). To demonstrate flexibility of the generator, the first variation asks to generate the optimal code for the thrice-unrolled algorithm (in other words, compute three time steps at once to further reduce the B/F ratio). The second variation is to generate optimal code for a different computation

$$\vec{w}_2 = -\mathcal{S}^{-1}\vec{a} + 2\vec{a} - \mathcal{S}^1\vec{a}$$
$$\vec{w}_1 = -\mathcal{S}^{-1}\vec{a} + \mathcal{S}^1\vec{a}$$
$$\vec{b} = \vec{a} + c_1\vec{w}_1 + c_2\vec{w}_2$$

where c_1 and c_2 are scalar constants.

3.4 Hidden Markov model

Alexander Schliep, a bioinformatics expert, posed a problem about Hidden Markov Models (HMM). HMMs are statistical models widely used in machine learning and bioinformatics.

High-level specification Finding the probability of observing a particular state as the next state in the HMM involves a matrix-vector multiplication, where the matrix states the transition probabilities between states. Depending on the sparsity level of this matrix, some particular loops must be unrolled automatically (fully or partially), and the transformations $0 * x = 0$ and $0 + x = x$ need to be performed.

Desired optimizations and Challenge Kenichi Asai posted a simplified version of the HMM problem[8]. Given a general matrix-vector multiplication program

```
int * f(int n, int **a, int *v) {
  int *w = (int*)calloc (n, sizeof (int));
  for (int i=0; i < n; i++)
    for (int j=0; j < n; j++)
      w[i] += a[i][j] * v[j];
  return w;
}
```

that is used to compute the next state in HMM, a desired optimization is to unfold the loop for each row of a particular adjacency matrix if the number of elements of the row is below a certain threshold value. For example, for the matrix given below on the left and the threshold value 3, the desired output is on the right:

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

```
int * f(int *v) {
  int *w = (int*)calloc (5, sizeof (int));
  w[0] = v[0] + v[3];
  w[1] = v[2];
  w[2] = v[1];
  for (int j=0; j < 5; j++)
    w[3] += a[3][j] * v[j];
  w[4] = v[2] + v[4];
  return w;
}
```

The challenge is to perform the unfoldings and to obtain the desired output.

4. Sample solutions to the stencil problem

To answer the stencil challenge and its variations, this section describes a domain-specific language for element-wise computation on shifted arrays and its two implementations, using MetaOCaml[9] and Lightweight Modular Staging – the Scala framework for building DSLs [10]. Both implementations let the programmer automatically apply the stencil optimization and attain the code that is equivalent to the hand-optimized code shown in §3.3.

4.1 MetaOCaml solution

The key observation is that any element-wise computation on shifted arrays can be implemented as a single loop, with all array references within its body being of the form a[i+n] where i is the loop variable and n is a small and statically known constant. A stencil for an array a may be then thought of as a cache – a memo table keyed by the offset n. When the element a[i+n] is needed, we check the cache for the entry with the key n, loading it from the array if it was missing. At the end of the iteration, we re-key the cache: the element at the key, say, 0, will now have the key -1. Temporary arrays like $\vec{w} = \vec{a} * \mathcal{S}^1\vec{a}$ are likewise cached: if an entry, say w[i+1], is missing, it will be computed as a[i+1] * a[i+2] rather than loaded from memory. Temporary arrays therefore become ephemeral: they never have to be allocated.

This memoization of array access is straightforwardly staged. Since all memo table indices are statically known, the lookup can be performed at code generation time and the memo table implemented as a sequence of local variables. Such staged memoization has been described in detail by Kameyama et al. [6]. The outcome is reminiscent of the common scalar-promotion optimization.

The final piece of the solution is the handling of edge elements. Near the edge, computations on shifted arrays may generate references to non-existing elements (e.g., at the index -1). Such references return 'halo' values set by the programmer. To eliminate the index bounds check in the main loop, the loop over i=0..N−1 is split into three loops, over i=0..n−1, i=n..N−m−1, and i=N−m..N−1 where n and m are small constants chosen such that

[8] https://github.com/StagedHPC/shonan-challenge/tree/master/problems/hmm

[9] http://www.metaocaml.org

all references in the middle, main loop, are definitely within the array bounds. The first and the third loops are fully unrolled.

The code[10] develops the DSL for stencil computation in detail and has many samples of generated code and numerical tests. For illustration, here is the naive generator of the double-time-step computation:

```
let t21 max_bounded a b =
  forlooph 2 min_bounded max_bounded (
    let a = gref0 a in
    let w1 = a *@ ashift 1 a in
    let wm = a -@ w1 +@ ashift (-1) w1 in
    let w2 = wm *@ ashift 1 wm in
    b <-@ wm -@ w2 +@ ashift (-1) w2)
```

where $+@$, $*@$ etc. are element-wise addition, multiplication and other operations on abstract arrays. An abstract array, unifying in-memory, ephemeral and cached arrays, is a function from an index to the code computing the value at that index. The operation $\mathcal{S}^n$, or ashift n in the code, offsets the index. The loop body is a function from the index (loop variable) to the code for the computation at that index; forlooph 2 generates the overall loop from min_bounded, currently 0, to max_bounded, the statically unknown upper bound for array indices. The generator unrolls the first two and the last two iterations. For the optimal code, we merely stencil-memoize all abstract array computations:

```
let t25 max_bounded a b =
  let p = new_prompt () and q = new_prompt () in
  push_prompt q (fun () ->
  forlooph 2 min_bounded max_bounded (with_body_prompt p (
    let a = stencil_memo q p (gref0 a) in
    let w1 = stencil_memo q p (a *@ ashift 1 a) in
    let wm = stencil_memo q p (a -@ w1 +@ ashift (-1) w1) in
    let w2 = stencil_memo q p (wm *@ ashift 1 wm) in
    b <-@ wm -@ w2 +@ ashift (-1) w2)))
```

The memoizer needs to know the loop scope (to generate loop-local bindings) and the scope outside the loop (to generate variables that cache data across loop iterations: the stencil itself). These scopes are denoted by the so-called prompts p and q. The generated code, shown in the comments in stencil.ml, reads and writes a global array once per iteration and does six floating-point operations, with B/F of 16/6.

Evaluation The MetaOCaml EDSL answers the challenge: it lets the programmer write element-wise computations on shifted arrays at a high level and apply the stencil optimization. The generated code has essentially the same loop body and the same B/F as the hand-written code in §3.3. Incidentally, the first version of the hand-written code left the elements b[0], b[1], and b[N−1] of the output array uninitialized and read past the end of the input array. Getting edge cases right is really difficult. The generated code has very long and boring pieces of code before and after the main loop, to properly compute the edge elements. Writing such tedious code by hand is excruciatingly boring. Code generation truly helps.

Our DSL lets us program shifted array computations in the form close to the mathematical notation. Generating the optimal three-time-step code is as straightforward as transcribing the mathematical specification. The variations of the challenge are hence easily satisfiable.

Alas, the correctness of the DSL (hence the correctness properties of the generated code) are not formally stated let alone independently verified (e.g., by the type system or a theorem prover). We just have to trust the DSL author. Although the type system of (pure) MetaOCaml does assure that the generated code is always well-typed, the stencil challenge solution had to use effects in code generation – in particular, effects that insert let-bindings across other let-bindings. Such effects void MetaOCaml guarantees and may lead to unbound variables in the generated code – so-called scope extrusion. During the development of the DSL, scope extrusion has indeed happened. Such problems are sometimes discounted: after all, the code with unbound variables will not compile, so the problem clearly manifests itself well before the run-time. The experience showed that looking through the generated code (which is typically rather messy with uninformative variable names) trying to determine the culprit within the generator has proved non-trivial. It remains an open problem, the subject of ongoing work, to develop a practical system that statically prevents scope extrusion.

4.2 Scala solution

The Scala solution to the stencil challenge, submitted by Tiark Rompf[11], relies on many of the same key ideas described in the previous section. It, too, regards a stencil as a sliding cache. It uses a different approach, so called LMS [10, 12]. MetaOCaml is purely generative: the generated code is treated as black-box, never to be transformed or even looked into. LMS, in contrast, produces the final code in a sequence of staging transformations. The intermediate representation produced by one stage can be examined and re-written by the next stage. Such an intensional code analysis lets us implement the stencil through the common subexpression elimination: the main driver generates the code for the loop body with the loop indices i and i+1 and detects the computations that can be shared.

The most visible difference between the two solutions comes from extensive meta-programming framework built around Scala [10]. The framework, which automatically performs many operations such as common-subexpression elimination, makes the development of new embedded DSL almost trivial. For example, here is the generator solving the baseline challenge:

```
def w1(j: Rep[Int]) = a(j) * a(j+1)
def wm(j: Rep[Int]) = a(j) - w1(j) + w1(j-1)
def w2(j: Rep[Int]) = wm(j) * wm(j+1)
def b(j: Rep[Int]) = wm(j) - w2(j) + w2(j-1)

for (i <- (2 until n-2).sliding ) {
        output(i ) = b(i)
}
```

The DSL code looks almost exactly as the mathematical notation. The stencil caching is activated by the .sliding method, which the DSL author has to implement. The generated code matches the hand-written one.

Evaluation Like the MetaOCaml solution, the Scala solution answers the challenge. The variations of the challenge are hence likewise easily satisfiable. The Scala solution also provides no independently verifiable correctness guarantees.

5. Sample solution to the HMM problem

Unfolding of the loops is not a brand new application of staging; examples exist for vector-vector multiplication [2, 4]. Kenichi Asai posted the first solution to the HMM problem, in MetaOCaml.[12]

[10] http://okmij.org/ftp/meta-programming/HPC.html#stencil

[11] https://github.com/TiarkRompf/virtualization-lms-core/blob/delite-develop2/test-src/epfl/test11-shonan/TestStencil.scala

[12] https://github.com/StagedHPC/shonan-challenge/blob/master/problems/hmm/specification.pdf?raw=true

Tiark Rompf submitted another solution, using the LMS framework in Scala.[13]. For the lack of space, we show only the latter:

```
def sparse_mv_prod(a: Array[Array[Int]], v: Rep[Array[Int]]) = {
  val v1 = NewArray[Int](n)
  for (i ← 0 until n: Range) {
    if ((a(i) filter (_ != 0)).length < 3) {
      for (j ← 0 until n: Range) {
        if (a(i)(j) != 0)
          v1(i) = v1(i) + a(i)(j) * v(j)}
    } else {
      for (j ← 0 until n: Rep[Range]) {
        v1(i) = v1(i) + (staticData(a(i)) apply j) * v(j) } }
  }
  v1 }
```

It is rather straightforward, with the first if-statement deciding if the i-th row of a has fewer than 3 non-zero elements. The types tell which iterations are done at compile-time and which are left for run-time. The input array `a : Array[Array[Int]]` and the global `n : Range` have the Rep-free types, the types of values available at code-generation time. Hence the loops `until n: Range` are done at generation time. In contrast, `v: Rep[Array[Int]]` is the "code" type, for an integer array. Therefore, `a(i)(j) * v(j)` generates *code* for multiplying the two values. Likewise, for `(j ← 0 until n:Rep[Range])` generates the code for the loop. Scala's implicit coercions transparently promote, or "lift", statically known values to the corresponding code, with one exception where the promotion has to be coded explicitly using the `staticData` operator.

Evaluation The solution completely answers the challenge. It can be written more concisely by abstracting the pattern of selected loop unrolling in a combinator `unrollIf` – see the code at GitHub.

6. Conclusions

There is an uncanny similarity between HPC and formal programming language research, in that the optimal HPC code, like a formal programming language proof, is very tedious, straightforward but with lots of details, with a small mistake invalidating large amount of work. Just as formal programming language research can benefit from proof assistants, HPC can benefit from meta-programming tools and research to build the tools. The similarity between the two areas is so strong that the motivation of the POPLmark challenge [1] applies to HPC modulo replacement of 'PL designers' with 'HPC practitioners' and 'developers of automated proof assistance' with 'staging programmers'.

It is fitting therefore to conclude by paraphrasing the abstract of the POPLmark challenge paper [1]:

> How close are we to a world where
>
> - every paper on high-performance computing is accompanied by an electronic appendix with machine program generators?
> - natural-science grad students no longer need to translate their high-level formulas into Fortran?
>
> We propose an initial set of benchmarks for measuring progress in this area. These benchmarks embody many aspects of HPC that require domain knowledge and difficult, for non-compiler experts, to perform with correctness guarantees: generating code with arbitrary number of new bindings, with complex loop transformations. We hope that these benchmarks will help clarify the current state of the art, provide a basis for comparing competing technologies, and motivate further research.

It remains to build the system that assures the generated code is well-typed (and free from array bound access problems, etc) and is also convenient to use. Proposed solutions have also been posted to HPC practitioners for their evaluation.

[13] `https://github.com/TiarkRompf/virtualization-lms-core/blob/delite-develop2/test-src/epfl/test11-shonan/TestHMM.scala`

Acknowledgments

We thank all participants of the Shonan Meeting for their contribution and discussion of the challenge problems. We thank Kenichi Asai for initiating the challenge and Walid Taha for coining the term "Shonan Challenge". Many helpful suggestions by Tiark Rompf, Sam Kamin and anonymous reviewers are gratefully acknowledged. Baris Aktemur received support from Tübitak grant 110E028.

References

[1] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. A. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLMARK challenge. In J. Hurd and T. F. Melham, editors, *TPHOLs 2005: Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics*, number 3603 in LNCS, pages 50–65. Springer-Verlag, 22–25 Aug. 2005.

[2] C. Chen and H. Xi. Meta-programming through typeful code representation. In *ICFP '03: Proc. of the 8th ACM SIGPLAN Int. Conference on Functional Programming*, pages 275–286, 2003.

[3] A. Cohen, S. Donadio, M. J. Garzarán, C. A. Herrmann, O. Kiselyov, and D. A. Padua. In search of a program generator to implement generic transformations for high-performance computing. *Science of Computer Programming*, 62(1):25–46, Sept. 2006.

[4] R. Davies and F. Pfenning. A modal analysis of staged computation. In *POPL '96*, pages 258–270. ACM Press, 1996.

[5] L. Hochstein and V. R. Basili. The ASC-alliance projects: a case study of large-scale parallel scientific code development. *IEEE Computer*, 41(3):50–58, Mar. 2008.

[6] Y. Kameyama, O. Kiselyov, and C.-c. Shan. Shifting the stage: Staging with delimited control. *Journal of Functional Programming*, 21(6): 617–662, 2011.

[7] O. Kiselyov, C.-c. Shan, and Y. Kameyama. Bridging the theory of staged programming languages and the practice of high-performance computing. Technical Report 2012-4, National Institute of Informatics, Japan, 2012. URL `http://www.nii.ac.jp/shonan/wp-content/uploads/2011/09/No.2012-4.pdf`.

[8] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[9] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *GPCE '10*, pages 127–136. ACM, 2010.

[10] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6):121–130, 2012.

[11] T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Odersky, and K. Olukotun. Building-blocks for performance oriented DSLs. In O. Danvy and C. chieh Shan, editors, *DSL*, volume 66 of *EPTCS*, pages 93–117, 2011.

[12] T. Rompf, A. K. Sujeeth, N. Amin, K. J. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *POPL*. ACM Press, 2013.

[13] J. A. Stratton, C. Rodrigues, I.-J. R. Sung, L.-W. Chang, N. Anssari, G. D. Liu, W.-m. W. Hwu, and N. Obeid. Algorithms and data

optimization techniques for scaling to massively threaded systems. *IEEE Computer*, 45(8):26–32, Aug. 2012.