Functionally Redundant Declarations for Improved Performance Portability

— Extended abstract —

Artjoms Šinkarovs
School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh
a.sinkarovs@macs.hw.ac.uk

Sven-Bodo Scholz
School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh
s.scholz@hw.ac.uk

1. Introduction

Most of the programmers who got used writing their codes in imperative style find it very difficult to switch to a more declarative style. One of the difficulties they often have is that a declarative style of programming in most cases makes it very difficult to develop a cost intuition. Even worse, the declarative style typically prevents programmers from enforcing low-level choices which on some architectures are essential for achieving acceptable performance. Consequently, programmers in the declarative domain have to rely on the abilities of the compiler to get these choices right. If the compiler fails to deliver the desired performance, typically, there is very little that can be done to achieve substantial performance improvements.

One way out of this dilemma is to provide declarative encapsulations of low-level mechanisms and, thus, to shift back the control to the programmer. Examples for that approach are explicit parallelism control instructions such as `par` and `seq` or destructive updates for statefull arrays. Being properly integrated into the declarative world, these can subsequently be abstracted away by applying the full arsenal of declarative weaponry. Strategies as described in [2] as well as the parallel arrays described in [1] demonstrate the benefits of this approach very nicely. Even though one might argue that this approach actually puts back the obligation to specify how things are to be computed into the hands of the programmer and, thus, is counter the basic credo of declarative programming, the elegant abstractions on top of the low level primitives create a rather declarative eco-system.

The downside of this approach is that it is at odds with the current shift towards a rather diverse landscape of computing platforms. Different hardware architectures may, and increasingly do, require different solutions not only on the lowest level but also in the way the code is mapped to hardware and in the way code is optimised. A good example for this are the differences in code optimisation for GPUs and for multi-core CPUs. Compilation for FPGAs has yet other requirements. When targeting heterogeneous systems, even mixtures of different codes need to be employed. This diversity and heterogeneity of modern hardware architectures renders old assumptions inapplicable and, as a result, the software written with old assumptions in mind increasingly fails to deliver excellent performance. To make matters worse, re-writing the performance critical sections from sketch for every new hardware architecture is only viable in very limited application scenarios.

This observation seems to demand a renaissance of the purely declarative approach and, with it, the challenge of being stuck with whatever performance a compiler can deliver for a given specification.

This paper proposes a new and, as we will argue, more declarative approach towards helping the compiler to find a good performance across a wide range of platforms. The key idea is to enable programmers to specify code equalities of various kinds. These equalities could be different algorithms that specify the same computation such as box standard matrix multiplication and Strassen's algorithm, or they could be some domain-specific knowledge such as a formulation of some basic laws on certain function compositions, or they could be different low-level versions that aim at several different hardware-specific effects. Based on these equalities, a compiler then can choose whatever combination of code snippets will deliver a good performance on a given target hardware. The extent to which such equalities are provided can be tuned to the capabilities of any given compiler. Consequently, any given set of equalities can be extended to widen the capabilities of a compiler. The important aspect here is that all the specifications create options without forcing the compiler to take a specific route.

In order to support this statement with an example, we are presenting a study of different implementations of the Cholesky Decomposition. The benchmark has a number of properties that are very common to several algorithms from linear algebra, so the results can be transferable to the whole class of applications.

Most of the algorithms of linear algebra, like for example Cholesky Decomposition, can be expressed on scalar elements or on matrix blocks. Blocking is very handy if it can increase the level of concurrency for a given sub-operation, which it normally does. However, one has to choose the granularity right, one has to consider a new memory access and one has to decide how to process an individual block: dividing it further, or switching to the algorithm on scalars.

In this work we are studying several options for Cholesky decomposition on shared memory machines equipped with CUDA graphic cards and capable of doing SIMD operations. First of all we are going to demonstrate effects of choosing one or the other implementation. Secondly we consider kinds of optimisations a compiler should be capable of in order to generate efficient code. Finally we are going to demonstrate how the sizes of input data and properties

of an architecture influence choices of algorithm implementation. We also discuss which properties can be obtained automatically and which should come from a programmer.

The overall approach we take can be considered as a generalisation of DSL-based approaches, where all the optimisation potential stems from domain-specific knowledge within one particular application domain. In our setting we hope to achieve the same transformational potential by means of establishing algorithmic equalities between functions and function compositions. After this step is done, a compiler gets significantly larger set of options it can chose from. By applying the knowledge about underlying architecture and by being able to approximate execution times, it can do a better job in composing the parts delivering the best performance.

Our paper is structured as follows. The next section introduces our case study problem of Cholesky Decomposition. It discusses several possible algorithms for Cholesky Decomposition alongside with their C implementation and some rationale why the different versions are of practical interest. Section Three provides an extensive performance study of the considered versions. Section Four repeats the experiment using now the functional language SaC and the attending toolchain's ability to generate code for a range of platforms. Section Five discusses the findings and outlines our proposed solution to the problem. Section Six presents related work before we conclude in Section Seven.

2. Cholesky decomposition

In this section we present the Cholesky decomposition which we are going to use as a running example through the rest of the paper. For every Hermitian positively defined matrix A, a decomposition into lower triangular matrix L and its conjugate transpose is called a Cholesky decomposition.

$$A = LL^* \tag{1}$$

For our discussion we are going to consider the case when A is real, which simplifies the equation to:

$$A = LL^T \tag{2}$$

From this specification we can deduce a straight-forward computation of L which is called Cholesky–Banachiewicz algorithm:

$$\begin{aligned} A &= LL^T \\ &= \begin{pmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{31} & l_{33} \end{pmatrix} \begin{pmatrix} l_{11} & l_{21} & l_{31} \\ 0 & l_{22} & l_{31} \\ 0 & 0 & l_{33} \end{pmatrix} \\ &= \begin{pmatrix} l_{11}^2 & & \text{symmetric} \\ l_{21}l_{11} & l_{21}^2 + l_{22}^2 & \\ l_{31}l_{11} & l_{31}l_{21} + l_{32}l_{22} & l_{31}^2 + l_{32}^2 + l_{33}^2 \end{pmatrix} \end{aligned} \tag{3}$$

Which allows us obtaining an algorithm to compute L:

$$l_{ij} = \begin{cases} \sqrt{a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2} & i = j \\ \frac{1}{l_{jj}} \sqrt{a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk}} & i > j \end{cases} \tag{4}$$

By analysing the formula we can conclude that it would be possible to reuse the memory of the original matrix to compute L. Two straight-forward implementations can be deduced directly from the formula. One would update the diagonal element and update the matrix row by row or column by column. From the perspective of a row-major data representation, row-by-row update is a bit more favorable as the updated elements are going to be sequential in memory. The corresponding code in C looks as follows:

```
1  void cholesky_ref_inplace (real * _m, size_t n)
   {
3    real (*m)[n][n] = (real (*)[n][n])_m;

5    for (size_t i = 0; i < n; i++)
       for (size_t j = 0; j < (i+1); j++) {
7        real s = 0;

9        for (size_t k = 0; k < j; k++)
           s += (*m)[i][k] * (*m)[j][k];
11
         if (i == j)
13         (*m)[i][j] = sqrt ((*m)[i][i] - s);
         else
15         (*m)[i][j] = 1.0 / (*m)[j][j] * ((*m)[i][j] - s);
       }
17 }
```

As we use dynamically allocated two-dimensional array, we have to use `(*m)[i][j]` syntax to access the element at row i and column j. We use type alias `real` to alternate between float and double types.

Several interesting things to notice about the implementation. First of all, the loops in lines 5 and 6 can be interchanged, but not with a simple swap. It would be possible to express the same loop where j would iterate from 0 to n and i would iterate from j to n. This optimisation probably can be done by some of the compilers by using some of the dependency analysis techniques, but most likely would be rejected as useless, as the overall gain is not clear.

The loop at line 9 can can be rewritten using SIMD operations. The benefit of that is that we do V multiplications at the same time and do the reduction of V elements. V in this case is a length of a vector register on a given architecture. Technically, this optimisation is often considered unsafe, as it changes the canonical order of the reduction, which might result in a different value due to rounding errors. Modern compilers like GCC can identify this loop as vectorisable, but due to order violation it would skip it as unsafe. A user can override this by passing `--unsafe-math` flag to the compiler.

As for concurrency, both row-updating and column-updating algorithms grant a chance to run the inner loop (line 6 in case of row-updating algorithm), for that a slight code modification is required. In the row-updating algorithm the last element has to be computed after all the elements in line j are already pre-computed. In case of colum-updating algorithm, the diagonal element has to be precomputed first before the rest of the column can be updated.

Finally we can observe that both algorithms would be less suitable for computing L^T, assuming that row major order is used, as that would require traversing along the columns and would also prevent from vectorising a loop at line 9. Obviously in the column-major data storage, the computation of L^T would make more sense. In order to perform transition from computing L^T instead of L, a compiler would need to know that, first of all, the matrix is symmetric, and secondly, that the elements in the upper triangular are not going to be used later in the program. To our knowledge none of the existing compilers could possibly do such an optimisation. Some compilers could decide to copy the data to have a better memory access, but that requires quite some sophisticated analysis to figure out that it actually is not harmful. A general rule of thumb is to avoid as much copying as possible.

Another version of the algorithm that we are going to study is based on the observation that we can try to get rid of the loop at line 9, by updating all the elements on the previous iteration. In other words, we can avoid reads from the columns less than i (in case we are computing L) if on the previous iterations all the columns greater than i would have been updated. Mathematically it can be expressed as follows:

$$\begin{pmatrix} \alpha_{11} & a_{21}^T \\ a_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} \lambda_{11} & 0 \\ l_{21} & L_{22} \end{pmatrix} \begin{pmatrix} \lambda_{11} & l_{21}^T \\ 0 & L_{22}^T \end{pmatrix} = \begin{pmatrix} \lambda_{11}^2 & 0 \\ \lambda_{11} l_{21} & l_{21} l_{21}^T + L_{22} L_{22}^T \end{pmatrix} \quad (5)$$

In this notation α_{11} and λ_{11} are scalars, a and l are column vectors and A_{22} and L_{22} are matrices of size $n-1 \times n-1$. Now, $L_{22}L_{22}^T$ term suggests that L_{22} can be found if we apply cholesky to the whole term. Hence the algorithm decomposes into the following steps:

$$\begin{pmatrix} \lambda_{11} = \sqrt{\alpha_{11}} & \lambda_{11} l_{21}^T \\ l_{21} = \frac{a_{21}}{\lambda_{11}} & L_{22} = \text{cholesky}(A_{22} - l_{21} l_{21}^T) \end{pmatrix} \quad (6)$$

That gives us the following straight-forward implementation in C:

```
void cholesky_rec (real * _m, size_t n)
{
  real (*m)[n][n] = (real (*)[n][n]) _m;

  for (size_t i = 0; i < n; i++) {
    (*m)[i][i] = sqrt ((*m)[i][i]);

    for (size_t j = i+1; j < n; j++)
      (*mat)[j][i] /= (*m)[i][i];

    for (size_t j = i+1; j < n; j++)
      for (size_t k = i+1; k <= j; k++)
        (*m)[j][k] -= (*m)[j][i] * (*m)[k][i];
  }
}
```

In order to automatically deduce this implementation from the reference one, a compiler has to have a powerful dependency-analysis mechanism which would allow to reschedule the iterations avoiding reads from the columns greater than i. Probably some of the polyhedra-based compilers can figure this out, however, such an optimisation, intuitively at least, brings a slowdown as it shifts reduction into memory updates. The number of updates increases from $O(n^2)$ to $O(n^3)$, so it is highly unlikely that any of the compiler would actually choose such a code transformation.

Nevertheless, there are two important facts to be noted here. First of all, if we would compute L^T instead of L the memory access at line 8 becomes sequential, and all the inner loops become vectorisable by means of SIMD instructions. Unfortunately C compiler won't do it automatically, as vector loads and stores should be aligned to be effective, otherwise memory operations might outweight vectorisation benefits. Now, here we can use the knowledge that the compiler doesn't have – we don't care about the elements below (in case of L^T) the main diagonal and we can use the fact that $a \equiv \frac{a}{\sqrt{a}}$, which might or might be known to a compiler. So we can perform aligned vector stores overwriting the elements that are irrelevant to our computation. Finally, as operations in the nested loop at line 11 can be executed in arbitrary order, we can interchange loops at lines 11 and 12 in order to get sequential memory access. This can be expressed in C[1] as follows:

```
1  #define sz(V) vector_size (V * sizeof (real))
   typedef real __attribute__ ((sz (V))) vecreal;
3  void cholesky_rec_up_vec (real * _m, size_t n)
   {
5    real (*m)[n][n] = (real (*)[n][n]) _m;

7    for (size_t i = 0; i < n; i++) {
       real mii = sqrt ((*m)[i][i]);
9      vecreal *  rowi = (vecreal *) (*m)[i];

11     for (size_t iv = i / V; iv < n / V; iv++)
         rowi[iv] /= mii;
13
       for (size_t j = i+1; j < n; j++) {
15       vecreal *  rowj = (vecreal *) (*m)[j];
         for (size_t kv = j / V; kv < n / V; kv++)
17         rowj[kv] -= (rowi[kv] * (*m)[i][j]);
       }
19   }
   }
```

[1] C with GCC-specific extensions which allow portable vector programming avoiding architecture-specific assembly.

Finally we consider the blocked version of Cholesky decomposition. The overall idea is the same as with the recursive algorithm, but we split the original matrix in four parts where the top left submatrix is of size $k \times k$. Then after applying some of linear algebraic operations we get the following:

$$\begin{pmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} L_{11}^T & L_{21}^T \\ 0 & L_{22}^T \end{pmatrix} = \begin{pmatrix} L_{11} L_{11}^T & L_{11} L_{21}^T \\ L_{21} L_{11}^T & L_{21} L_{21}^T + L_{22} L_{22}^T \end{pmatrix} \quad (7)$$

Now from the following equation we can deduce the final blocked algorithm:

$$\begin{pmatrix} L_{11} = \text{cholesky}(A_{11}) & 0 \\ L_{21} = A_{21} L_{11}^{-T} & L_{22} = \text{cholesky}(A_{22} - L_{21} L_{22}^T) \end{pmatrix} \quad (8)$$

As we learned previously, we would prefer to compute L^T instead of L as it is more beneficial for the row-major order. So the code for the blocked L^T looks as follows:

```
void cholesky_top_upper (real * _block, real * _diag,
                         size_t bs)
{
  real (*block)[bs][bs] = (real (*)[bs][bs]) _block;
  real (*diag)[bs][bs] = (real (*)[bs][bs]) _diag;

  for (size_t i = 0; i < bs; i++) {
    for (size_t j = 0; j < bs; j++)
      (*block)[i][j] /= (*diag)[i][i];

    for (size_t k = i+1; k < bs; k++)
      for (size_t j = 0; j < bs; j++)
        (*block)[k][j] -= (*diag)[i][k] * (*block)[i][j];
  }
}

void cholesky_bot_upper (real * _block, real * _b1,
                         real * _b2, bool diagonal_p,
                         size_t bs)
{
  real (*block)[bs][bs] = (real (*)[bs][bs]) _block;
  real (*b1)[bs][bs] = (real (*)[bs][bs]) _b1;
  real (*b2)[bs][bs] = (real (*)[bs][bs]) _b2;

  for (size_t j = 0; j < bs; j++)
    for (size_t i = 0; i < bs; i++) {
      real t = -(*b2)[j][i];
      size_t k = diagonal_p ? i : 0;

      for ( ; k < bs; k++)
        (*block)[i][k] += t * (*b1)[j][k];
   }
}

void cholesky_blocked_upper (real *  _blocks, size_t bn,
                             size_t bs)
{
  real * (*blocks)[bn][bn];

  blocks = (real * (*)[bn][bn]) _blocks;
  for (size_t i = 0; i < bn - 1; i++) {
    size_t k, j;

    cholesky_diagonal ((*blocks)[i][i], bs);

    for (size_t j = i+1; j < bn; j++)
      cholesky_top_upper ((*blocks)[i][j],
                          (*blocks)[i][i], bs);
```

```
50      for (k = i+1; k < bn; k++) {
          for (j = k; j < bn; j++)
52          cholesky_bot_upper ((*blocks)[k][j],
                                (*blocks)[i][j],
54                              (*blocks)[i][k],
                                j == k, bs);
56      }
    }
58
    cholesky_diagonal ((*blocks)[bn-1][bn-1], bs);
60 }
```

Please note that `cholesky_diagonal` call in the blocked function can be substituted with any available implementation of the Cholesky decomposition. The choice of the implementation depends on the problem size and underlying architecture. Update of the bottom triangular (`cholesky_bot_upper`) is actually a matrix multiplication where the first matrix is transposed. Again, that might be arbitrary difficult for a compiler to spot this, especially as the implementation includes a special case of a upper-triangular matrix. The current implementation of the blocked algorithm gets a matrix which is already split into blocks. The splitting routines are not included here.

To our knowledge none of the existing compilers can automatically deduce a blocked version from any of the non-blocked ones. Some of the polyhedra-based frameworks can perform tiling of the loop, but that is not enough. The current blocked version abstracts away operations on individual blocks and applies those abstractions on the groups of blocks. In order to have a transition from tiling a compiler would have to prove that all the operations on all the blocks can be abstracted away into finite number of operations and then to find a scheduling for the block operations. It is arbitrary hard to solve this in general.

3. Experimental Evaluation

In this section we are going to demonstrate the runtime of different implementations we have discussed earlier. First of all we notice that the reference implementation reveals a clear cubical behaviour while running on a single core. The runtime can be approximated as $T(n) = c\frac{n^3}{2}$ with a precision of 10%. That allows us to use the reference runtime as a basis.

All our measurements currently presented in this paper were taken on Intel i3-2310M CPU with AVX SIMD instructions. The length of the SIMD register is 256 bits which allows to perform arithmetic operation on 4 doubles or on 8 floats. We used GNU GCC compiler version 4.7.3 p1.0 with the following compilation flags enabled: `-march=native -mtune=native -Ofast -funroll-all-loops`. The runtime figures were obtained by measuring pure function execution time avoiding initialisation and input-output operations. The resolution of the timer is 10^{-9} sec. The runtime was taken as a median out of 5 consequent runs.

First of all the vectorisation of the reference implementation (line 9 in `cholesky_ref` results in 1.5 times difference for all the problem sizes. The way to check that was to compile the benchmark with `-fno-tree-vectorize` flag. Further down all the measurements of the reference version are performed on a vectorised code.

As we can see on Fig. 1 the `cholesky_rec_up_vec` version of the algorithm clearly outperforms the reference one on the problem sizes with N being less than 300. After such a size the reference version becomes faster. This effect can be clearly seen on bigger problem sizes (Fig. 2).

The behaviour of the blocked version (we use the constantly sized blocks 64×64 elements) can be seen at Fig. 2 which demonstrates performance benefits for N being greater or equal to 1024 elements.

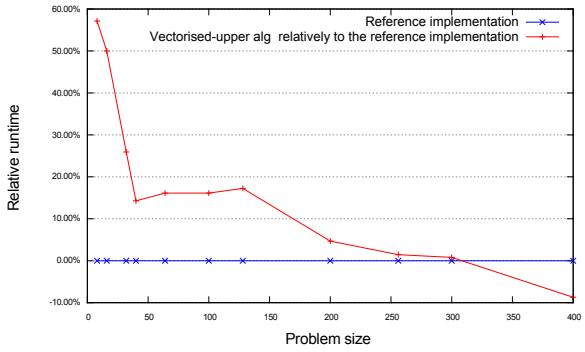

Figure 1. Runtime of the Cholesky decomposition on small matrices using vectorised algorithm (`cholesky_rec_up_vec`) and the reference implementation.

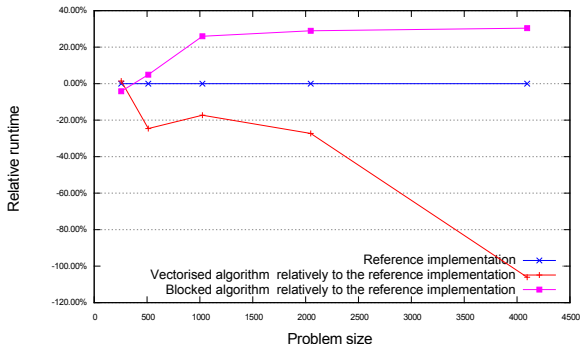

Figure 2. Runtime of the Cholesky decomposition on big matrices using vectorised algorithm (`cholesky_rec_up_vec`), blocked algorithm (`cholesky_blocked_upper`) and the reference implementation.

The reference version reveals some potential for parallelism, but as we can only do one row or column concurrently, which makes granularity of work very small an also it introduces a necessity to synchronise on every row/column. That results in relatively poor scaling. Using OpenMP annotations, the maximum performance increase was 20% for 4 threads.

The blocked and recursive versions grant more concurrency as the update of a triangular (line 11 in recursive algorithm and line 50 in blocked) can be done in parallel. The only problem with that is non-rectangular shape of the data structure which has an impact on the workload of each individual thread. The naive parallelisation of the outer loop allows to get two times speed-up in case of blocked version.

4. Cholesky Decomposition in SaC

In this section we relate C implementation and its behavior with corresponding SaC implementations. We would start with a reference version which can be literally translated from C into SaC. It looks as follows:

```
double[.,.] cholesky_ref (double[.,.] A)
2 {
  n = shape (A)[0];
4
  for (i = 0; i < n; i++)
6   for (j = 0; j < i + 1; j++) {
      s = 0d;
8     for (k = 0; k < j; k++)
        s += A[i,k] * A[j,k];
10
      if (i == j)
```

```
12          A[i,j] = sqrt (A[i,i] - s);
          else
14          A[i,j] = 1.0d / A[j,j] * (A[i,j] - s);
      }
16    return A;
  }
```

The recursive version (`cholesky_rec`) is implemented in SaC using data parallel constructs to update the bottom triangular of L^T, however, the region of update increases from triangular to rectangular. By doing this, a scheduling within the multi-threaded execution becomes simpler, and the overhead can be amortized starting if the number of threads is large enough.

```
1 double[.,.] cholesky_rectangular (double[.,.] A)
  {
3   n = shape(A)[0];
    for (i = 0; i < n; i++) {
5     A[i,i] = sqrt (A[i,i]);
      A = with {
7           ([i,i+1] <= iv < [i+1, n])
              : A[iv] / A[i,i];
9       }: modarray (A);
      A = with {
11          ([i+1,i+1] <= [j,k] < [n,n] )
              : A[j,k] - A[i,j] * A[i,k];
13      } : modarray (A);
    }
15  return A;
  }
```

Finally the blocked version uses rectangular algorithm to compute Cholesky decomposition on the blocks from the main diagonal and the `cholesky_bot_upper` is implemented as a call to matrix multiplication.

```
  double[.,.] trisolve (double[.,.] L, double[.,.] A)
2 {
    bs = shape (L)[0];
4   for (i=0; i< bs; i++) {
      A[i] = A[i] / L[i,i];
6     A = with {
          ([i+1] <= [ip] < [bs]): A[ip] - L[i,ip] * A[i];
8     } : modarray (A);
    }
10  return A;
  }
12
  double[.,.] cholesky_blocked (double[N,N] AA)
14{
   A = block (AA, 2^bse);
16 bn = shape (A)[0];

18 for (bi = 0; bi < bn; bi++) {
     A[bi,bi] = cholesky_rectangular (A[bi,bi]);
20   A = with {
          ([bi,bi+1] <= iv < [bi+1, bn])
22          : trisolve (A[bi,bi], A[iv]);
      } : modarray (A);
24  A = with {
          ([bi+1,bi+1] <= [bj,bk] < [bn,bn])
26          : A[bj,bk] - matmmul (transpose (A[bi,bj]),
                                  A[bi,bk]);
28    } : modarray (A);

30 }
   return unblock (A);
32 }
```

In the full paper we are going to include more details regarding each variant of the algorithm as well as performance figures on a range of architectures including GPUs.

5. Specifying Code Equalities

As we can see from the previous section, the performance obtained, even in a highly optimising declarative setting, depends very much on a combination of three aspects:

- the algorithm including potential parameterisations such as the blocking size,
- the problem size, and
- the architecture the program is run on.

In particular the dependency between the algorithm used and the performance obtained shows that a pure compiler-optimisation-based approach in infeasible in general.

Another important observation from our experiment is that it is not the implementation of several alternative algorithms that is tedious. In contrast, it is the runtime experiments for finding out which one to use which takes most of the effort.

The key idea of this paper is to create a mechanism that allows the programmer to do the former and enables the compiler to do the latter.

All that is needed to achieve that goal is to extend the overloading mechanism of the language and to implement a mechanism that chooses between the variants provided.

We propose two extensions over the existing language: First, we allow function overloadings with identical name and parameter types. Second, we introduce the notion of non-functional parameters. These are parameters that are not provided by applications of the function but are chosen by the compiler. In conjunction with these non-functional parameters, we also allow for constraint expressions in order to restrict the sset of possible values.

$$
\begin{array}{lcl}
\mathit{signature} & \Rightarrow & \mathit{rettypes}\ \mathit{funname}\ \left\lceil \mathit{nf\_args} \right\rceil \mathit{f\_args} \\
\mathit{nf\_args} & \Rightarrow & \texttt{[}\ \mathit{args}\ \left\lceil \texttt{|}\ \mathit{expr} \right\rceil \texttt{]} \\
\mathit{f\_args} & \Rightarrow & \texttt{(}\ \mathit{args}\ \texttt{)} \\
\mathit{rettypes} & \Rightarrow & \mathit{type}\ \left\lceil \texttt{,}\ \mathit{type} \right\rceil^{*}
\end{array}
$$

Figure 3. SaC syntax extensions to allow specification of equal algorithms.

Our syntax for function headers can be found in Fig. 3, and for the blocked Cholesky example we could specify:

```
  double[.,.] cholesky(double[.,.] A)
2 {...}

4 double[.,.] cholesky(double[.,.] A)
  {...}
6
  double[.,.] cholesky[int bse | bse > 3 ](double[.,.] AA)
8 {
    A = block (AA, 2^bse);
10  bn = shape (A)[0];

12  for (bi = 0; bi < bn; bi++) {
      A[bi,bi] = cholesky (A[bi,bi]);
14    A = with {
          ([bi,bi+1] <= iv < [bi+1, bn])
16          : trisolve (A[bi,bi], A[iv]);
      } : modarray (A);
18  A = with {
          ([bi+1,bi+1] <= [bj,bk] < [bn,bn])
20          : A[bj,bk] - matmmul (transpose (A[bi,bj]),
                                  A[bi,bk]);
22    } : modarray (A);

24  }
    return unblock (A);
26 }
```

Note here, that we have three different implementations of cholesky. Only the blocked implementation has one non-functional parameter named `bse` which is of type `int` and is constrained to be larger than 3. This non-functional parameter then is referred to in the body to compute the block size as `2 ^ bse` which serves as a functional parameter of the function `block`. Another interesting aspect is that any application of `cholesky` only receives the functional parameters. In the example, this can be observed for the recursive call in line 13.

We define the semantics of these extensions as a non-deterministic choice between the matching overloading and a non-deterministic choice of the non-functional parameters. In terms of implementation, we envision a profiling-based choice between the specified alternatives.

6. Related Work

...will be in the full paper only...

7. Conclusions

This paper looks at the problem of obtaining high levels of performance from declarative specifications. At the example of Cholesky Decomposition, we show that a single implementation does not suffice to get good performance on a range of architectures, even when building on the very high transformational potential that results from the side-effect free setting. We analyse various different implementations in an imperative as well as in the declarative setting. These experiments show clearly that even in the declarative setting there is a need to provide implementation alternatives and that the choice between the alternatives requires experimental evaluation.

Consequently, we propose a language extension that enables the programmer to specify implementation alternatives but leaves the tedious, execution platform specific choice between these alternatives to the compiler and the runtime system.

This approach has several benefits. It opens up a wide range of compilation options. The choice between variants can either be based on code analyses or on profiling or auto-tuning techniques, or on combinations thereof. This flexibility also allows the compiler to enforce choices depending on the context in which such code alternatives are executed. As a consequence, this mechanism could be used to generally inform the compiler about domain specific equalities and, thus, enable meta-level optimisations typically found in the context of DSLs. If time to solution is the dominating criterion and overall resource consumption only plays a minor role, different alternatives can be executed in parallel and the first result be chosen.

References

[1] G. Keller, M. M. T. Chakravarty, R. Leshchinskiy, S. L. P. Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In P. Hudak and S. Weirich, editors, *ICFP*, pages 261–272. ACM, 2010. ISBN 978-1-60558-794-3.

[2] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, Jan. 1998.