Automatic Locality-Friendly Interface Extension of Numerical Functions

Benjamin Hess Thomas Gross Markus Püschel

Department of Computer Science
ETH Zurich, Switzerland
hessbe@student.ethz.ch, {trg, pueschel}@inf.ethz.ch

Abstract

Raising the level of abstraction is a key concern of software engineering, and libraries (either used directly or as a target of a program generation system) are a successful technique to raise programmer productivity and to improve software quality. Unfortunately successful libraries may contain functions that may not be general enough. E.g., many numeric performance libraries contain functions that work on one- or higher-dimensional arrays. A problem arises if a program wants to invoke such a function on a non-contiguous subarray (e.g., in C the column of a matrix or a subarray of an image). If the library developer did not foresee this scenario, the client program must include explicit copy steps before and after the library function call, incurring a possibly high performance penalty. A better solution would be an enhanced library function that allows for the desired access pattern. Exposing the access pattern allows the compiler to optimize for the intended usage scenario(s). As we do not want the library developer to generate all interesting versions manually, we present a tool that takes a library function written in C and generates such a customized function for typical accesses. We describe the approach, discuss limitations, and report on the performance. As example access patterns we consider those most common in numerical applications: permutations, striding and block striding, as well as scaling. We evaluate the tool on various library functions including filters, scans, reductions, sorting, FFTs and linear algebra operations. The automatically generated custom version is in most cases significantly faster than using individual steps, offering gains that are typically in the range of 20–80%.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – Code Generation, Compilers, Optimization; D.2.2 [*Design Tools and Techniques*]: Software libraries, User interfaces

Keywords Libraries, components, performance, interface extension, preprocessors, programming language features interaction, software product lines

1. Introduction

Raising the level of abstraction is a key concern of software engineering. Components and libraries offer a successful example: a library hides low-level details and offers an abstraction to its users. A library or component raises programmer productivity (the user can focus on the level of abstraction defined by the library) and improves software quality (assuming the interface is designed and correctly implemented by experts). A good example are numerical performance libraries that can greatly improve the productivity of application programmers. If most of the application runtime is spent in functions provided by the library, a high level of performance may be achieved without optimization effort.

Unfortunately, even successful libraries often contain functions that are not general enough. Libraries are pre-implemented and therefore they make assumptions on the data type, the data layout, and the exact computation that is performed. If the application requires a variant that is not supported, some driver code is needed, which may decrease or completely annihilate the performance gains offered by the library.

One typical example of this situation is a numerical library function that assumes that the in- and output arrays are contiguous in memory. Consider the simple finite impulse response (FIR) filter in Fig. 1. If the client program needs to apply this filter to all the even-indexed elements of an array `x` to produce the output array `y`, an explicit copy operation is needed to pack the input data as shown.

```
fir(double *in, double *out, int len)
   for(int i = 0; i < len-1; i++)
      out[i] = (in[i+1] + in[i])/2;

for(int i = 0; i < len; i++)
  in1[i] = in[2*i];
fir(in1, out, len)
```

Figure 1: Simple FIR filter and its use when applied to strided input data (here: stride = 2).

The copy code that must be included in the program has many undesirable consequences. First, there is extra work that must be done. Second, the program suffers from poor locality and increased cache misses (in the present case the performance may decrease by 2–3x). Depending on the details of the context of the call to a library function, such copy code may be required for different call sites, increasing the size of the program.

If the library developer had foreseen this use, it would have been possible to provide this support with a more general function, e.g., by allowing for strided access as shown in Fig. 2. Note that

`f_prime` does not just move the copy into the function, but removes it altogether and instead modifies the index expressions and hence the array accesses. Now, the client program can perform the filter with one call to `fir_prime`. Allowing in addition for strided output would require yet another code modification and extension of the interface.

```
fir_prime(double *in, double *out,
            int len, int s)
   for(int i = 0; i < len-1; i++)
      out[i] = (in[(i+1)*s] + in[i*s])/2;

fir_prime(in, out, len, 2)
```

Figure 2: Simple FIR filter with support for strided access and its use on strided data.

Some library functions provide extended interfaces, in particular those whose applications are well established. For example, dgemm in BLAS [6] provides three additional parameter that allow the multiplication of matrices within larger matrices, and FFTW [8] provides an interface to support arbitrary block-strided data layout for in- and output.

Contributions. In this paper we present a source-to-source translation tool that can take a C function (such as Fig. 1) as input and create variants (such as Fig. 2) automatically. The variants have additional input parameters and can be viewed as generalizations of the original function. Specifically, the tool has the following abilities:

- It modifies functions (meaning interface and body) to provide them with support for four common access patterns for arrays: strided access (as above), block-strided access (e.g., subimages in images), permuted accesses, and array scaling.
- The support for these patterns can be on the input arrays, the output array, or both.
- The support for these patterns can be combined (e.g., strided access plus scaling).
- If the C function includes vector code (i.e., intrinsics for SSE instructions to access multiple data items with one instruction) then this property is preserved and the function variants also use intrinsics.

This tool can help a library designer to provide generalized versions of a function that otherwise would just exist in one version that would need to cover all cases, and as a consequence either would require glue code at the call site or not exploit the obvious optimization opportunities that are implied by different access patterns. We concentrate on the handling of different access patterns as those have a significant influence on the performance of numeric libraries. We explain the design and implementation of the tool and discuss limitations. Finally, we show the effectiveness of the tool on a collection of numeric functions. Specifically, we compare the performance of the modified function to an equivalent implementation using the original function with copies. In most cases the speed-up is significant.

2. Supported access patterns

In this section we present the access patterns supported by our tool. Each of these patterns occurs frequently in numerical applications operating on arrays. Without special support by a chosen library function, each pattern requires an explicit copy step or pass through the data. Our tool removes this step by fusing it with the function body, thus reducing work and potentially improving performance. Note that this does not mean that the copy is performed inside the functkion as explicit step, rather the first (or last) accesses to the data arrays are modified (as for example in Fig. 2).

```
for(int i = 0; i < size; i++) {
        strided[i] = in[i*s];
}
f(strided, out, size);
```

Figure 3: Strided access of an array.

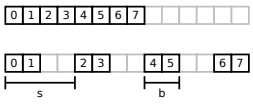

Figure 4: Blockstride access pattern with $b = 2$ and $s = 4$.

```
for(int i = 0; i < size; i++) {
   int bnum = i/bsize;
   bstrided[i] = in[bnum*s + i%bsize];
}
f(bstrided, out, size);
```

Figure 5: Block strided access of an array.

The four basic patterns are as follows. The term ”array” refers to an input or output array of a library function.

- *Strided access* of an array (as in the introduction before);
- *Block strided access* of an array;
- *Permuted access* of an array;
- *Scaling* of all elements of an array either with a single value or with an array of scaling values.

We briefly discuss these access patterns assuming as example a function `f(in, out, size)` that computes the array `out` from the array `in`, both of length `size`. Then we discuss the requirements that must be satisfied by a library function so that it can be augmented to include the access patterns.

Strided access. As explained in the introduction, this pattern occurs when in- or output data are not contiguous but spread with a constant stride `s`. The logic of this pattern is shown in Fig. 3.

Examples include applying a function to the column of a matrix in C (the stride will be the number of columns) or applying a filter to the red channel in an RGB image (the stride will be three). Our tool modifies the function to include the stride as parameter and removes the need to copy by remapping the array accesses.

Block strided access. This access pattern generalizes the strided access in that it allows the array to be spread as *blocks* with constant block size b and stride s. An example is visualized in Fig 4. For given b and s, $b < s$, the block number is given by $\lfloor i/b \rfloor$ and the position within the block by $i \bmod b$. The logic of this pattern is shown in Fig. 5.

The most important usage scenario for this pattern is applying a function to matrices within matrices, e.g., subimages of images.

Permuted access. Such an access is not directly to an array but to a permuted version of it. The permutation is specified by an integer array of the same length. The logic of this access pattern, when applied to our running example `f(in, out, size)` is shown in Fig. 6.

An example where this might occur is a Fourier transform function that gets the input in a form that is bit-reversed or underwent

```
for(int i = 0; i < size; i++) {
   permuted[i] = in[permutation[i]];
}
f(permuted, out, size);
```

Figure 6: Permuted access to an array.

```
for(int i = 0; i < size; i++) {
   in[i] *= scale;
}
f(in, out, size);
```

Figure 7: Scaling of an array with a fixed constant.

```
for(int i = 0; i < size; i++) {
   in[i] *= scale[i];
}
f(in, out, size);
```

Figure 8: Scaling of an array with another array.

a perfect shuffle. Our tool will translate such a function into one where the permutation array is passed as additional parameter and transforms the code to avoid the explicit permutation in Fig. 6.

Scaling. Scaling adjusts the range of the input or output array of a function. It is conceptually different from the previous patterns in that it transforms the data and not the access pattern. Prescaling scales the input before the function is applied. Postscaling adjusts the output of the function. The scaling factors can be constant or given by an array of the same length (vector scaling). The logic of the two types of scalings is shown in Figs. 7 and 8 for the case of prescaling.

As an example, scaling is often used in signal processing to ensure that an array has norm (= energy) 1. If a library function is applied to such an array, it makes sense to create a variant that gets the scaling factors (a number or array) as additional parameters and performs the scaling as part of the function.

Vector code. Optimized library functions often explicitly vectorize the code using intrinsics (e.g., for Intel's SSE considered in this paper). To preserve the performance benefits obtained this way, it is important that the use of intrinsics is preserved when integrating the access pattern into a library function.

Limitations to ensure correctness. To allow the automatic generation of variants of a function `f` that fuse the access pattern with the body of `f`, several constraints must be fulfilled. We list the most important constraints that are needed for the analysis. Later, we discuss additional restrictions that are due to the implementation of the tool.

Assignments with a pointer on the left hand side must be of the form `p = q + index` where `p` and `q` are pointers of the same type and `index` is an arbitrary expression with an integer type as a result. For an occurring `p` such an assignment must occur exactly once (SSA format) unless `q = p` in which cases the assignment is allowed multiple times.

For the scaling transformation, the underlying memory block of a pointer is either read or written, never both within a function. Without this limitation, the tool cannot determine if an access is already scaled because the accessed value could have been written by a scaled value beforehand.

If recursion is used, the allocated memory blocks referred to by parameters must stay the same throughout the recursion.

Each of these conditions can be detected by our tool for a suitable error message in case of failure.

3. Generalizing library functions

Given a library function `f` and a specific access pattern (e.g., a vector scaling or a strided access) for an input or output array, our tool generates a generalized variant `f_prime`. This variant has a widened interface (so a scaling vector or stride can be specified) and an accordingly modified function body. To do this, our tool must understand how `f` operates on input and output parameters. This information is kept in a dependence graph that shows how source and destination operands are modified. We first explain this graph and then show how it is used to perform the various source code transformations. We will focus on the striding access and discuss the others more briefly as they are mostly analogous.

3.1 Dependence graph

The transformations need the total offset and the origin of access to the array affected by the access pattern. To provide this information, a dependence graph for the input function `f` is created. The nodes of the graph represent the pointers and integer variables used in the function. The edges between the nodes represent statements that create a dependence between these. Fig. 9 shows the graph for a simple function `f`.

```
void f(double* src, double* dst, int size) {
    double* p = src + size;
    int half = size / 2;
    dst[half] = p[size/2];
}
```

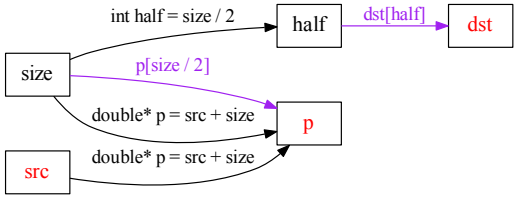

Figure 9: Dependence graph.

More formally, the graph is defined as follows:

1. There is a node for every pointer, formal parameter, or integer variable.
2. Pointer assignments of the form `p2 = p1 + index` create a dependence edge from `p1` to `p2`.
3. Assignments of integer variables of the form `i = <expr>` create a dependence edge from every integer variable in `<expr>` to `i`; accordingly `i+= <expr>` creates in addition a dependence of `i` on itself as does `i++`.
4. Usages of integer variables or formal parameters in array accesses of the the form `a[<expr>]` create a dependence edge from every variable in `<expr>` to `a`.

The edges are annotated by the expression(s) that tie the variables (pointers, formals, arrays) together. Special nodes represent loops. The dependence graph for `f` captures just enough information for the tool to operate (e.g., floating variables are ignored) and poses implicit limitation on the type of code that can be processed (e.g., casting a float to an integer, which is then used in an array access, is not allowed).

3.2 Transforming the body

Given a function `f` and its dependence graph, the body of the function is transformed according to the kind of access code that is to be included. We explain these transformations for each access pattern.

3.2.1 Striding

The striding transformation moves the relevant elements by a striding factor apart. The striding transformation does not modify any data, but only indices; namely it multiplies the total offset of an access by the striding factor. The detailed steps are:

1. The first step is checking the dependences. A dependence of arrays arises from an assignment `p2 = p + offset`, from which can be concluded that the offset of `p2` must be `offset` bigger than the offset of `p`. This offset must then be multiplied with the striding factor, resulting in `p2 = p + offset * stride`.
2. After the dependences are handled, every relevant array subscript (or dereference) is modified. The index of an access is again multiplied with the striding factor, e.g., `p2[index]` is replaced with `p2[index * stride]`.
3. Our tool can also transform C that was vectorized with intrinsics. Our implementation considers the SSE instruction set with either 2-way doubles or 4-way floats. Vectorized code poses an additional challenge as the most commonly used load/store intrinsics load multiple elements that are consecutive in memory. Thus the striding of the elements forces a switch to single slot load/store intrinsics:
   (a) The index of an access is again multiplied with the striding factor. The resulting address points to the first element to load.
   (b) If packed load/store intrinsics are used, they are split up into either 2 (double) or 4 (float) single slot loads. These use the address from the previous step. For each single slot load-/store intrinsic, the address is adjusted by the striding factor to get the subsequent element. For example the intrinsic load `x1 = _mm_load_ps(src + x + y * width)` has an offset of `x + y*width`. This call is transformed into four single slot loads with shuffling, as shown in Fig. 7. Intrinsic stores are handled analogously.
   (c) After loading the single elements, shuffle intrinsics combine the single elements into one SSE register.

Pseudo code for the entire striding transformation (showing for SSE 2-way double precision only) is shown in Fig. 11.

3.2.2 Blockstriding

The blockstriding transformation is a more general case of the striding transformation and can handle arbitrary block sizes (see Fig. 4). For a given striding factor s and a block size $b < s$, the position of a specific element in the blockstrided data structure can be calculated from the given position of the element in the consecutive data layout. If the original index of an element is i, the blockstrided position i_2 of the same element can be calculated as

$$i_2 = \underbrace{\lfloor \frac{i}{s} \rfloor}_{\text{block number}} * b + \underbrace{i \bmod b}_{\text{position inside block}} \quad (1)$$

The block size and the striding factor are added to the parameter list to use in the formula (1).

Handling of vectorized code is more complicated than in the striding transformation. In the striding transformation, every packed load or store must be replaced with its single slot counterpart, because only one element at the time is loaded. The blockstriding transformation can differentiate three different cases instead of always switching to single slot intrinsics (Table 1).

```
a_0 = _mm_load_ss(src + stride *
                  (x + y * width + 0));
a_1 = _mm_load_ss(src + stride *
                  (x + y * width + 1));
a_2 = _mm_load_ss(src + stride *
                  (x + y * width + 2));
a_3 = _mm_load_ss(src + stride *
                  (x + y * width + 3));

a_0 = _mm_shuffle_ps(a_0, a_1, 0);
   // shuffle a_0 and a_1 together
a_2 = _mm_shuffle_ps(a_2, a_3, 0);
   // shuffle a_2 and a_3 together
a_0 = _mm_shuffle_ps(a_0, a_2, 136);
   // shuffle a_0 and a_2 together
x1 = a_0;
```

Figure 10: Example of striding transformation of `x1 = _mm_load_ps(src + x + y * width);`.

These cases depend on the alignment of the parameters, the block size and the striding factor. The parameters are aligned if the address is divisible by 16, block size and striding factor are aligned if they are divisible by either 2 or 4, depending on the used data type. If the parameters, block size and striding factor are aligned, packed aligned load/store intrinsics can be used. In the second case, the block size is aligned but not the start of a block and therefore unaligned packed load/store intrinsics must be used. If the block size is unaligned, unaligned packet load/store intrinsics are used until there are not enough elements left in the current block. The remaining elements and elements from the next block are loaded with single slot intrinsics. These cases can only be checked at runtime and therefore the transformation inserts a check at the beginning of the function and branches to the corresponding code block. This branching enables the use of higher performing code if the alignment conditions are fulfilled.

The most important application of the block striding transformation is the access of subimages in images. If the larger image (of size `height` by `width`) is represented as array `a`, the function may use throughout an indexing style like `a[i+width*j]`. In that case, the subimage can be accessed in the same style as `a[i+b*j]` with a suitable adjustment of the ranges of `i` and `j`. We refer to this special case as a 2d optimized blockstriding transformation and the tool attempts to use induction variables (i.e., the value is computed by a simple affine expression of the loop counters) to step through both the image and the subimage. More details can found in the report [12].

3.2.3 Permuting

The steps of the permuting transformation are similar to the steps of the striding transformation. Instead of multiplying the index with the striding factor, the index is used to access the permutation array `permuted`. To access this array, the total offset is needed; therefore additional variables are added to track the offset of every variable: For every assignment of a pointer (`p1 = p + offset`) an additional offset variable `p1_offset` is inserted. To calculate the offset of `p1`, the tool calculates `int p1_offset = p_offset + offset`. With those additional variables, the tool can easily determine the total offset of an access of any pointer and use it to get the permuted index.

An access to `p1` has an implicit offset included from its origin. As the permutation vector stores the indices of the elements

```
// insert offset variables
for assignment (p2 = p1 + offset) in all assignments:
   if p1 is parameter and function is not recursive:
      insert  p2_offset = offset  before assignment
   else:
      insert  p2_offset = p1_offset + offset  before assignment

// adjust recursive calls
if function has recursive calls:
   add offset parameters for all affected vectors
   for function call  f(..., p + offset, ...)  in all recursive function calls:
      if p in affected_parameters:
         // pass offset through newly added offset parameter of vector parameter p
         change call to  f(..., origin(p), ..., p_offset + offset) ;

// replace all scalar accesses
for access (p[i]) in all read accesses:
   if origin(p) in affected_parameters:
      replace  p[i]  with  origin(p)[ (p_offset + i) * stride ]

// replace sse double precision packed loads
for function call  sse_load(p + offset)  in all sse double precision packed loads:
   if p in affected_parameters:
      insert  int tmp_index = p_offset + offset;  before load statement
      insert  a_0 = _mm_loadh_pd(a_0, origin(p) + tmp_index + (stride * 1));  before load statement
      insert  a_0 = _mm_loadl_pd(a_0, origin(p) + tmp_index + (stride * 0));  before load statement
      replace  sse_load(p + offset)  with  a_0

// replace sse double precision packed stores
for function call  sse_store(p + offset, sse_value)  in all sse double precision packed stores:
   if p in affected_parameters:
      insert  _mm128d tmp_value = sse_value;  before store statement
      insert  int tmp_index = (p_offset + offset) * stride;  before store statement
      for n = 0 to 1:
         insert  _mm_store_sd(origin(p) + tmp_index + (stride * n), tmp_value);  before store statement
         insert  tmp_value = _mm_shuffle_pd(tmp_value, tmp_value, 1);  before store statement
      delete store statement
```

Figure 11: Pseudocode for the striding transformation including the handling of 2-way double SSE vectorized code.

Alignment			
Parameters	Block Size	Striding Factor	
✓	✓	✓	packed aligned intrinsics are *preserved*
✓/ -	✓	-	packed aligned intrinsics are converted to *packed unaligned intrinsics*
-	-	-	packed intrinsics are converted to unaligned intrinsics or split up into *single slot intrinsics* for the remaining elements in a block

Table 1: Different cases for SSE intrinsics handling in the blockstriding transformation.

in the parameter, which has a zero offset, the implicit offset of `p1` must be removed to access the correct permuted element. Therefore, the pointer `p1` is replaced with its origin resulting in `origin[permuted[p1_offset + value]]`.

3.2.4 Scaling

There are two types of scaling: prescaling and postscaling. Prescaling applies when data is read from an array, postscaling when data gets written to an array. The steps for a prescaling transformation are:

1. First, the tool creates a list of affected arrays using the dependence graph to check the dependences of the to-scale parameters and collects the dependent pointers. In addition, new variables are inserted to track the total offset when a pointer is assigned.
2. With the list of affected arrays, the tool first modifies the normal array subscripts (pointer dereferences are equivalent to array subscripts).
   - For a *scalar scaling* an access `p[i]` is replaced with `p[i] * scaling_factor`
   - For a *vector scaling* the tool needs to calculate the total offset of the access and use it as an index for the scaling vector: an access `p[index]` is replaced with `p[index] * scale[index + offset_of_p]`. The index of the scale vector `index + offset_of_p` is the total offset of the access to `p` and is calculated with the additional offset variables.
3. After all array subscripts are transformed, the tool checks if the function uses any SSE intrinsics. The changes to the vectorized

code are more complicated as there are several different ways to load an SSE register.

(a) The first step is to isolate the load intrinsics and assign the result to a temporary variable `tmp`.

(b) For vector scaling, two or four new scaling factors are loaded into an SSE register before the scaling operation can be executed. The index used to access the scaling vector must be the total offset used in the load intrinsic.

(c) Then, the multiplication of the temporary variable `tmp` with the scaling factor is done.

(d) If the load intrinsic does not load a full packed SSE register, the transformation must preserve the unloaded values in the registers to stay the same as before the multiplication. Therefore, shuffling intrinsics are needed to get the unscaled original values back into the final SSE register.

(e) Lastly, the scaled SSE variable is inserted where the original load intrinsic was placed.

4. Finally, any recursive call is modified to add the additional scaling parameter.

The postscaling transformation is similar. Instead of modifying read accesses, write accesses are modified. An assignment `p[i] = value` is transformed into `p[i] = value * scale` for a scalar scaling or `p[i] = value * scale[i]` for vector scaling. Vectorized code is handled the same way.

3.3 Widening the interface

The dependence graph provides the original formal parameters. Depending on the access transformation, additional parameters are needed (the scalar or a vector for scaling, striding and block information for (block)strided accesses, and a permutation vector for permutations). These parameters are added to the list of formal parameters. A user of the library variant must then provide these parameters at call sites to employ the generalized library function.

3.4 Combinations

Most transformations can be applied in any sequence to a given function that obeys the limitations in Section 2. The only limitation relates to the 2d optimized blockstriding transformation as it requires that index computations have the format `i + j*width`. The permuting transformation destroys this indexing style by replacing the index with an access to the permutation array and therefore a 2d optimized blockstriding cannot be applied after a permuting transformation has been applied.

The transformations do not commute, hence different orders of transformations produce different resulting functions. Some orderings influence the performance, others affect the added parameters of the following transformations. The most important implications are described next.

Scaling. The scaling transformation is the only transformation that does not affect any subsequent transformation. Scaling adds one floating point multiplication to every access and does not modify the index computations. However, the transformation is affected by previous transformations if vector scaling is used as it uses the index of a possibly modified access for accessing the scaling vector.

Striding. After a striding transformation, the indices are multiplied with the striding factor and therefore the size of a vector is also multiplied by the same factor. Subsequent transformations are affected as they are dependent on the used index. For example, a permutation vector must be strided if the permuting transformation is applied after the striding transformation. Furthermore, vector code may be split up into single slot accesses, resulting in an increase in the number of access operations. The subsequent transformations also operate on the increased access count and introduce a bigger overhead than would be necessary if these transformations were applied before the striding transformation.

Permuting. Similar to the striding transformation, subsequent transformations operate on the permuted index and therefore use the permutation too. After the transformation, the indexing format is `permuted[i]`, where `i` is the previously used index. This format is not compatible with the 2d blockstriding transformation, which therefore cannot occur after the permutation transformation.

Blockstriding. The blockstriding transformation is a more general case of the striding transformation. Therefore, all of the implications of striding apply also to the blockstriding transformation. The subsequently added parameters are blockstrided instead of strided. Further, the vectorized accesses are split into single slot accesses and the number of access operations is increased accordingly.

Reasonable combinations. If a specific sequence of transformations is useful or not depends on the requirements on the input or output data. However, two or more consecutive transformations of the same type can often be combined into one transformation resulting in a better performing function. Consecutive scaling, striding or permuting transformations can be easily combined into one transformation. If two consecutive 2D blockstriding transformations are applied, the first transformation has no effect on the function at all because the second transformation replaces the parameter that was added by the first transformation.

3.5 Implementation restrictions

The C language offers a variety of syntactic elements and control statements. As the tool is implemented in Python, and we do not know of an industrial-strength C frontend in Python, the tool handles only a (generous) subset of the C language. Note that these restrictions apply only to the library functions that are processed by the tool and do not apply to the application program that uses it. The allowed subset of C is sketched next.

Function declaration. The function that the transformation should be applied to must conform to the following:

- The return value can be `void`, `double` or `float`.
- Data types of arguments can be either a pointer or a scalar. Pointers represent input or output data and can have `double` or `float` as pointee type. Scalar arguments that store the length of a vector must be of `int` or `size_t` type; the remaining arguments can be of any scalar data type.

Global and local variables. The limitations on these the same as the limitations on the functions arguments. Additionally `__mm128` and `__mm128d` data types are allowed to enable vectorized code.

Function body. Allowed are only a few constructs: `for` loops with one iteration variable, `if/else` constructs, variable declarations and assignments. The latter are only restricted when the left hand side is any kind of pointer as is described in Section 2. `structs` and `switch` statements are not supported

Pointers. Due to the nature of the transformations, the most restrictions concern pointers:

- Pointer parameters do not alias;
- Pointers must only point to scalar data types (`double`, `float` or `int`) ;
- Pointers are not modified from outside the function;
- Pointers do not leave the function through a function call;
- Allowed operations on pointers are addition, subtraction, dereference and array subscript.

SSE. The usage of the streaming SIMD extensions are allowed by using intrinsic functions. Only the load/store intrinsics and the SIMD data types are restricted. The data types for packed float `_mm128` and packed double `_mm128d` are allowed. Furthermore, the most common load and store intrinsics are supported including packed and single slot, aligned and unaligned.

Macros/Defines. The usage of macros or defines are not allowed, as they are not fully expanded by the preprocessor of the parsing library. The only exception are the shuffle macros `_MM_SHUFFLE` and `_MM_SHUFFLE2`.

4. Evaluation

Our tool generalizes a given library function to support a chosen access pattern from Section 2. The main goal is that the generalized function (e.g., the one in Fig. 2) then performs faster than the original function surrounded by the extra code needed for the access pattern (e.g., as shown in Fig. 1).

Experimental setup. To validate success we apply our tool to the collection of 13 legacy numerical library functions that implement well-known functions including filters, scans, reductions, sorting, FFTs, and linear algebra operations, as shown in Table 2. We implemented the first 8 ourselves, the remaining 5 are Spiral-generated from [1, 15]. Access patterns not supported or not applicable are shown in the last column. The constraints for reduction exist because there is only one output value; for mergesort because it is inplace. 2d blockstriding refers to the optimization of induction variables discussed at the end of Section 3.2.2. Vectorized functions, which include SSE instrincs, have the prefix sse.

As input sizes we choose $400^2 n,\ 1 \leq n \leq 10$, for 1d inputs, and $400n,\ 1 \leq n \leq 10$, for 2d inputs (i.e., one dimension of the square matrices in filter and sse_mvm).

The platform is an AMD Phenom II X4 Deneb, 3.4 GHz with 8 GiB DDR3 1333MHz, running Windows 7. We used the Intel compiler icc 12.0.0.104 with flags /O3 /arch:SSE2 /fp:fast=2 /qipo_fas /Oa /Ow /Ob2 /Oi /Ot /fast /xHost.

For each function we applied each allowed pattern to the input, the output (not shown), and both. In each case we ran each of the 10 sizes 10 times to compute a total of 100 speedups. These are displayed as box plot that shows median and the range from first to third quartile. The whiskers show the lowest and highest datum with $1.5\times$ of the first and third quartile, respectively. The crosses show all outliers.

For the parameters of the access patterns we choose stride 8 for striding and in addition a block size of 32 for block striding. The permutations for permuting where chosen randomly, as where the factors for scaling (between 0 and 1).

Results. The results are shown in Fig. 12. Every row is a transformation: for the left column it was applied to the input, for the right column to in- and output. The x-axis shows the functions to which the transformation could be applied, the y-axis the speedup. A speedup > 1 is an improvement and higher is better. The median in each case is written inside the plot; black numbers signify speedups, red numbers signify slowdowns.

A first glance at Fig. 12 confirms that out tool provides speedups in the majority of cases; however a few also experience a significant slowdown. Explaining the various speedups is difficult, so we focus on a few cases where we found a plausible explanation.

Striding: For in-striding the typical speed-up is 15–20%; only contrast suffers a severe slowdown. For in/out-striding mergesort performs poorly because operating inplace on strided data has poor spatial locality throughout the computation. On the other hand the small, highly optimized functions implementing FFTs and MVM benefit more since the extra copy steps incur a comparatively high penalty; the same observation holds for the other transformations.

Blockstriding: This transformation has the most cases of significant slowdown (2 for in and 4 for in/out). A possible explanation is that in these cases the index expression in (1) is expensive relative to the actual computation (as is the case for filter). We also applied the optimized version discussed in 3.2.2 which is applicable to filter, contrast, and sse_contrast; the speedup is then about 2x in all three cases [12].

Permuting: For each plot there is only one case of significant slowdown; we do not have a good explanation. The speedups range from insignificant to about a 100%; the scan for in/out even achieves 150%.

Vector scaling: Here our method performs best with no relevant slowdowns.The speedups range from insignificant to 85%.

As the transformations preserve the use of SSE instrinsics (as discussed earlier), the baseline for functions with the prefix "sse" includes also SSE intrinsics. The results shown in Fig 12 for these functions indicate that the tool is very useful for these optimized functions and preserves the developer's insights and effort.

5. Related Work

We identified three directions of related work that we discuss in this section.

Libraries that support access patterns. The problem of assumptions in performance libraries that mismatch the patterns needed in applications is well-known and has been addressed in some popular libraries. For example, in linear algebra the entire BLAS specification is designed to match the access patterns in the higher level routines of LAPACK [2]. This explains extra parameters in gemm (matrix multiplication) and gemv (matrix-vector) multiplication that allow block-strided access (which is equivalent to allowing matrices within larger matrices). Recently it was observed that the gemm interface is not sufficient when multiplying matrices in higher-dimensional tensors [11].

In the recursive library FFTW, any (multi-dimensional) strided access pattern is supported [8]. This is convenient for users but necessary to support the recursion (which produces FFTs at increasing strides). Moreover, FFTW fuses scaling steps into base case FFTs, called twiddle codelets, for locality [7]. Other fusions of context (e.g., to support real FFTs) lead to a total of more than 10 variants for every input size. The variants needed have been determined by the developers. To generate libraries like FFTW automatically, [16] shows how to discover these in Spiral [14, 15] automatically from a mathematical specification and generates the associated code. However, the code is not produced by modifying a standard version as done here but directly from a mathematical description.

Compiler optimizations. Existing compilers do not perform the kind of optimizations targeted by our tool. If the compiler is able to inline a library function `f` then loop fusion (or loop jamming) might be able to merge the copy loop with a loop inside `f`. However, most C compilers inline only some standard or math library functions that are recognized by name [9] and do not inline recursive functions, which are handled by our approach. And while loop fusion merges adjacent loops, it requires that there are no flow-, anti-, or output dependences that connect the variables of the first loop with those of the second [10]. But copy code to adjust the context inserts exactly these kinds of dependences and the analysis for these is often not available for C programs. Furthermore, many library functions are complex and may not offer a simple loop to fuse with. And if the library function contains vector code with intrinsics, the copy loop and the loops in the body of the library function may have different trip counts.

An automatic optimization of code that follows the pattern "copy-library call-copy" (and that would be done by a compiler's optimizer) is possible but difficult in practice. An optimizer must ensure that the array set up by the copy step is used *only* by the

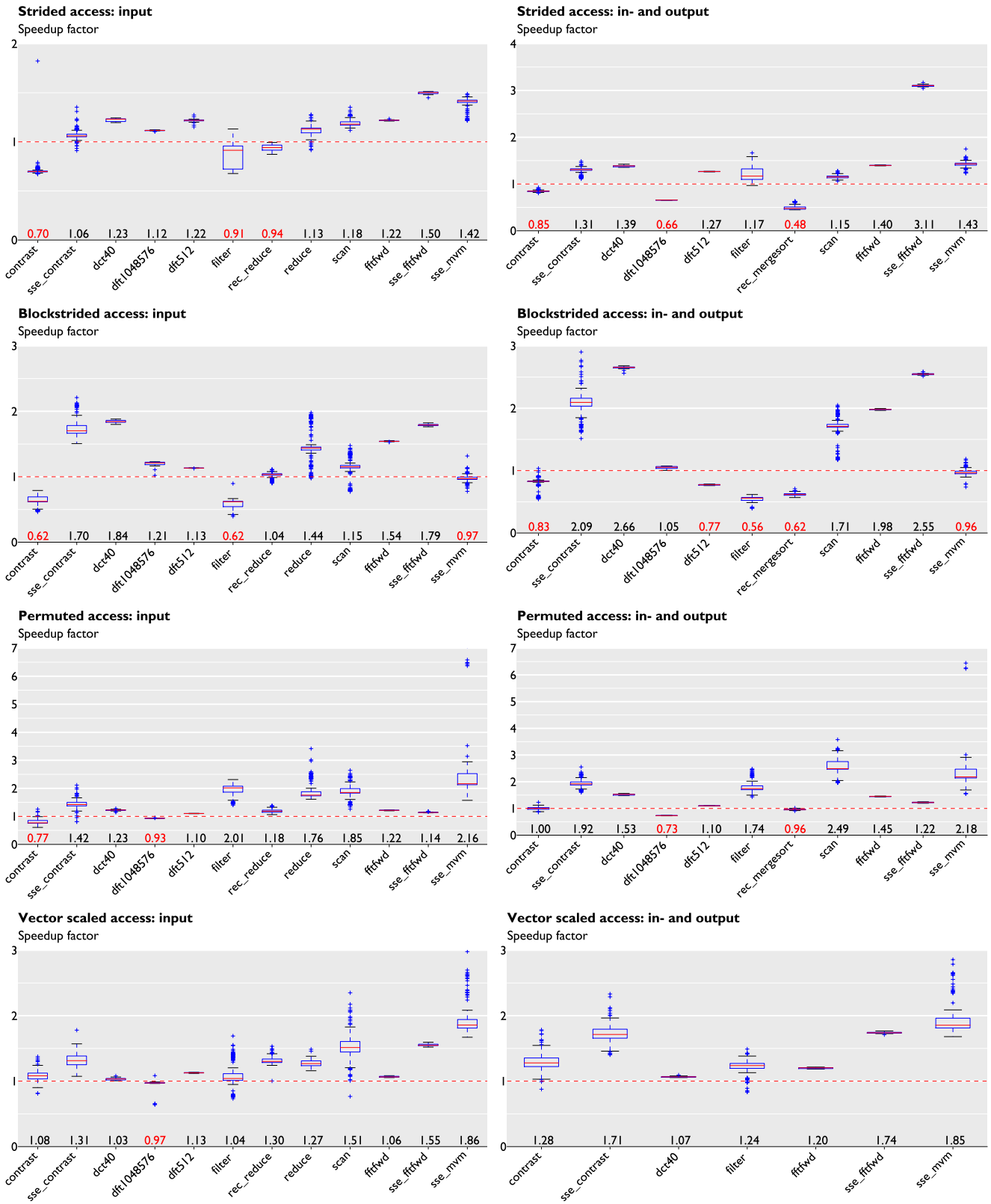

Figure 12: Benchmarks with the four types (rows) of transformations, applied to input only (left column) or to in- and output (right column).

Function	Description	DataType	Invalid/Not applicable Transformations
filter	Generic 2d convolution with a kernel of size 3x3	float	-
contrast	Normalizes the range of values in an image	double	-
sse_contrast	Vectorized version of contrast	double	-
sse_mvm	Vectorized matrix-vector multiplication	float	-
rec_mergesort	Recursive 2-way mergesort with $\mathcal{O}(n)$ extra storage for merging.	double	scaling, in- and out-(block)striding
reduce	Sum reduction of all elements	double	postscaling, out-(block)striding
rec_reduce	Divide-and-conquer sum reduction	double	postscaling, out-(block)blockstriding
scan	Calculates the prefix sum of a vector	double	postscaling, poststriding, out-(block)striding
dft512	Fast Fourier transform (FFT) for 512 elements generated by Spiral	double	postscaling, 2d blockstriding
dft1048576	FFT for 2^{20} elements generated by Spiral	double	postscaling, 2d blockstriding
fftfwd	Real FFT for 128 elements generated by Spiral	double	2d blockstriding
sse_fftfwd	Vectorized real FFT for 128 elements generated by Spiral	double	2d blockstriding
dct40	Discrete cosine transform for 40 elements generated by Spiral	double	2d blockstriding

Table 2: List of the functions used for performance evaluation.

library call that immediately follows. A conservative analysis may fail (even for single-threaded programs) if programmers re-use array temporaries. The tool presented here shifts the responsibilty to ensure the absence of aliasing to the developer and, in return, frees the developer from modifying the library code by hand.

Software product lines. Our work can also be phrased in the language of software product lines (SPLs) [4] in that our tool adds features to a base implementation to create variants. The set of all these variants is an SPL. In software engineering, several approaches are known to create SPLs including feature and aspect oriented programming or components or frameworks [3, 5, 13]. However, in that work, the software considered is usually more complex, the features are preimplemented, higher level languages are used, and the goal is not performance.

6. Concluding remarks

We described a tool that can extend a numerical C library function to allow for more general access patterns. Even though the tool is quite simple and has constraints on the input it can process, it proved to be able to process many typical functions including more complex ones such as vectorized matrix multiplication or highly optimized FFTs. In most cases, the functions generated by this tool provide substantial performance benefits compared to a standard function with added glue code. It would be interesting to consider a larger set of common access patterns to explore the limitations of our approach. Finally, although the tool is limited to C, the same design could be applied to other languages, e.g., those that provide additional meta information about structures (e.g., the dimensions of an array), which could be leveraged to reduce the number of additional parameters.

Finally, our results are also a feedback to library developers. It is important to understand the context and patterns employed by clients. It almost always does pay off to support the most common usage setups, and a tool as discussed here provides a simple way to generalize library functions.

References

[1] Spiral website. http://spiral.net/codegenerator.html.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 3rd edition, 1999. ISBN 0-89871-447-8 (paperback).

[3] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.

[4] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 3rd edition, 2001.

[5] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, 2000.

[6] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, 1990. http://www.netlib.org/blas/blasqr.pdf.

[7] M. Frigo. A fast Fourier transform compiler. In *Programming Languages, Design and Implementation (PLDI)*, pages 169–180, 1999.

[8] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[9] Intel. *Intel C++ Compiler XE 13.1 User and Reference Guide*. Santa Clara, CA, 2013. Document number: 323273-131US.

[10] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[11] E. D. Napoli, D. Fabregat-Traver, G. Quintana-Orti, and P. Bientinesi. Towards an efficient use of the BLAS library for multilinear tensor contractions. http://arxiv.org/pdf/1307.2100.pdf, 2013.

[12] Not-shown. Not shown due to anonymous review process. Technical report, 2013.

[13] D. L. Parnas. Designing software for ease of extension and contraction. In *Proc. International Conference on Software Engineering (ICSE)*, pages 264–277, 1978.

[14] 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.

[15] M. Püschel, F. Franchetti, and Y. Voronenko. *Encyclopedia of Parallel Computing*, chapter Spiral. Springer, 2011.

[16] Y. Voronenko, F. de Mesmay, and M. Püschel. Computer generation of general size linear transform libraries. In *Code Generation and Optimization (CGO)*, pages 102–113, 2009.