ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Automatic Refactoring: Locality Friendly Interface Enhancements for Numerical Functions

Master's Thesis

Benjamin Hess
hessbe@student.ethz.ch

Institute of Computer Systems
Department of Computer Science
ETH Zurich

**Supervisors:**
Prof. Markus Püschel (pueschel@inf.ethz.ch)
Prof. Thomas Gross (thomas.gross@inf.ethz.ch)

April 1, 2013

## Abstract

Recent improvements in processor architectures such as multiple cores and larger single-instruction multiple-data (SIMD) vector units increased the discrepancy between processing speed and memory bandwidth. Today, the memory bandwidth has become the biggest bottleneck in high performance domains. Numerical functions are often target to heavy optimizations to get as much performance as possible. These functions presume a specific data layout and have a fixed domain and range. Adjusting unsuitable data according to these requirements can take up a significant amount of the runtime due to heavy memory operations. By integrating layout, domain and range adjustments into numerical functions, memory bandwidth is saved as the adjustments happen in-place during execution of the function.

Four transformations are provided by the developed tool to refactor the most common adjustments into numerical functions. Restrictions on the to transformed function ensure the correctness of the transformations. These transformations enable the function to directly work on previously incompatible data which makes the manual adjustments superfluous and saves memory bandwidth by not needing to adjust the data manually before calling the function. The runtime of the transformed function is highly dependent on the used function and transformation type, ranging from 50% slower to up to 5 times faster compared to applying the adjustments manually before or after the function.

The developed tool shows that automatic refactoring with a subset of C is possible and allows a developer to enhance numerical functions with minimal additional effort providing more flexible and high-performance versions of the functions with only having to maintain the original function.

# Contents

# 1 Introduction

## 1.1 Motivation

Modern computer architectures offer a lot computing power with CPUs running at around 3-4 GHz with multiple cores. With more processing power the desire to process bigger datasets increases but the memory bandwidth did not increase as much as the processing power. Therefore, the memory subsystem is in many computations the biggest bottleneck and limit the maximal obtainable performance. Optimizations which should increase the performance often need to reduce memory bandwidth needs or adapt the algorithm to caching hierarchy. To reduce memory bandwidth usage, as many calculations as possible need to be done with loaded data. Fusing multiple steps together reduces the needed memory bandwidth largely.

Numerical functions often assume an domain, range and data contiguous in memory. If there are mismatches, they need to be corrected in additional steps before or after using the function. To correct the domain or range, often a scaling is sufficient. If the data is not contiguous, there are certain patterns which can be corrected by copying the elements together. All those correction steps are mostly using memory bandwidth and do only a few operations. Most of these steps can be avoided when the author of the function designed the function with these requirements built in.

Adding these requirements in a third party function can be an erroneous and tedious task, even more if the function is optimized or generated. The developer needs a long time to understand the internals of the function and to add the additional functionality correctly. In large functions, some needed changes can be overlooked and produce incorrect results.

The idea is to provide a tool which can automatically add several of these tasks to a numerical function and enhance their functionality. The additional work for a developer to use the tool should be minimal still providing the desired enhanced function.

## 1.2 Example

In this section, the application of the work is shown on an example. A good explaining example is any image filter function. The filter is taken as a black box, which calculates an unknown mathematical function on an image. It processes a whole image with a domain from 0.0 to 1.0. The goal is to use this filter on a part of an image and store this part on disk. The image is read from a bitmap on the disk which has one byte each color and pixel. This gives an input range from 0 to 255.

**Traditional Approach.** First, the filter only processes a whole image but only a part should be processed. Therefore, this part needs to be copied into a smaller image of exactly the size of the smaller part in order that the filter not processes the whole image. The next problem is, that the filter has a domain of 0.0 to 1.0. To fix it, every pixel has to be multiplied by $\frac{1}{255}$ before passing it to the filter. After the application of the image filter, the result is still in a range from 0.0 to 1.0. To store it on the disk, it needs to be again in the range 0 to 255. Scaling it back by multiplying by 255 brings it into the correct range. All the steps are summarized again:

1. extract part of the image

2. scale it by $\frac{1}{255}$
3. apply image filter
4. scale it back by 255

**Approach with Transformations.** The tool offers several transformation to handle the previous mentioned steps. The image filter gets modified to handle more generic cases. The extraction step is handled by the 2D optimized blockstride transformation. With the additional parameters added by the transformation, the image filter can work directly on the part of the image without copying it first into a separate image. The pre- and postscaling can be added to the filter with the scaling transformation. At the end, the image filter supports working on parts of an image and scaling it before and after the filtering. The 4 steps in the previous paragraph are now merged together in one call to the filter with additional parameters.

**Results.** Merging the 4 steps together saves memory bandwidth, as the transformed filter only processes the image once instead of 3 times. When using the unmodified filter, the data has to be extracted and scaled beforehand and scaled afterwards in an extra loop. On the downside, the transformations increased the cost of every read and write access of the image. Therefore, it highly depends on the filter if the transformed version performs better than the previous approach. The more often every element is accessed, the more likely is the runtime slower with the transformed filter.

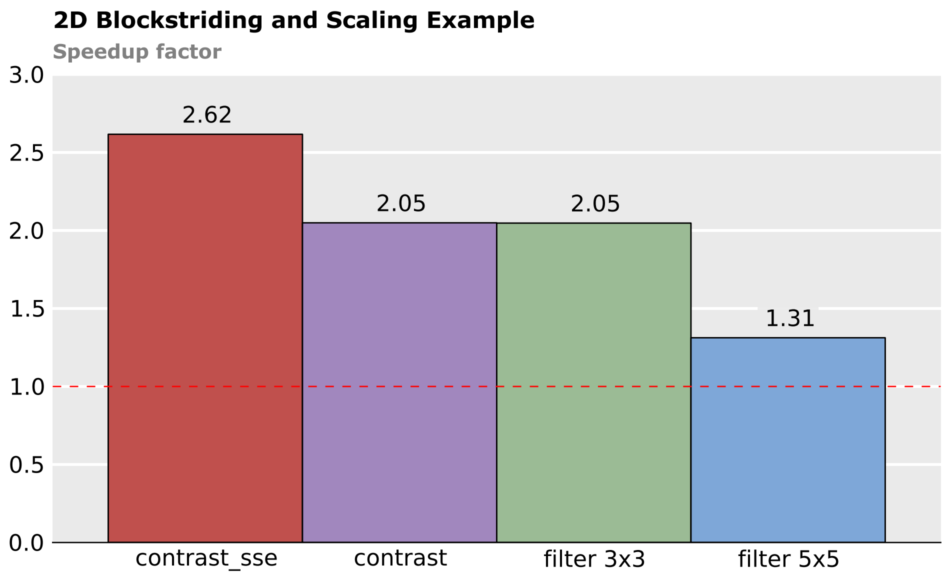

Figure 1: Speedup of using 2D blockstriding, pre- and postscaling on a 2D example

Figure 1 shows the quotient of the runtime before the transformation and after the transformation. The three functions are described in chapter 6.2. The transformations on these examples are quite beneficial, up to 2.6 times faster than without transformations.

The effect of the previously mentioned overhead can be seen on *filter 3x3* and *filter 5x5*. The bigger the filter is, the more often every element is read ($k * k$ times, $k$ being the kernel size) resulting in a less bigger speedup.

# 2 Concept

The main goal of the transformations developed in this work is to increase the performance compared to apply the additional functionality outside the original function. The tool should work completely automatic without any interaction with the developer. As many performance critical numerical functions are written in C and for the simplicity of the language (imperative programming language), the tool processes functions written in C. Although C is a relatively simple language, there are still restrictions on the input function which are described in the chapter 3.

The tool processes a C file and has a function name as input and creates a new C file with the same content including the transformed function. Because the whole file is parsed into an abstract syntax tree (AST), all the necessary files to correctly parse and type-check the file must be provided. The generated AST is then analyzed and modified according to the chosen transformation and finally converted back into source code.

The application is written in Python 2.7 [9] and uses the libclang library from the clang compiler tool chain [3] to parse the source code into an AST. The parsing library uses the same code base as the clang compiler itself which guarantees a correct and fast parsing.

## 2.1 Transformations

In this chapter, the implemented transformations are described. For each transformation, there is an example to show the usage of the transformation and equivalent code which shows the logic the transformation adds to the function. The following four basic transformations are supported:

- Scaling
- Striding
- Permuting
- Blockstriding

### 2.1.1 Scaling

Scaling adjusts the range of the input or output data of a function and therefore changes the domain and range of the function. The adjustment of the domain scales the data before the function is applied and is called prescaling. Adjusting the range of the function scales the data after the application of the function and is called postscaling.

Prescaling transforms every read access on the input data and multiplies it with the scaling factor before the function uses the value for any calculation. To ensure the prescaling is correct, only read accesses are allowed on the scaled vector, else it can not be guaranteed that values are only scaled once. Listing 1 shows the logically equivalent code without any transformation applied.

Postscaling is very similar to prescaling but instead of transforming every read access it needs to transform the write accesses and scale the value before getting written to the vector. Again, the vector can only be written, never read, in order to allow postscaling.

```
for(int x = 0; x < size; x++) {
  src[x] *= scale;
}
f(src, dst, size);
```

Listing 1: Logic of prescaling

**Vector scaling.** Pre- and postscaling scale every element with the same constant scaling factor. Vector scaling, on the contrary, scales every element with a separate factor. The equivalent code to apply vector prescaling is shown in listing 2.

```
for(int x = 0; x < size; x++) {
  src[x] *= scale[x];
}
f(src, dst, size);
```

Listing 2: Logic of prescaling with a vector

**Example.** A simple example is to adjust the domain of an image filter function. The original function expects the input data in a range from 0.0 to 1.0 but image data often has an range of 0 to 255. By scaling the input by $\frac{1}{255}$, the image appears to the function as it would have been in the correct domain. Furthermore, postscaling by 255 adjusts the range back again to the original range after the application of the function. The transformed function with the additional scaling parameters can have any arbitrary domain or range starting from 0.

### 2.1.2 Striding

The striding transformation, as opposed to the scaling transformation, works on the data layout not on the value range of the data. Elements are strided when they are not consecutive in memory as seen in figure 2, where every other element is relevant. This means between every relevant element is a fixed number of non-relevant elements. For example a RGB image with the 3 color channels interleaved, where every value of the same color is 3 elements apart, the other two color channel are in between.

Numerical functions often assume the data in a consecutive memory block and do not support any striding of elements. The mismatch between the data layout presumed by the function and the existing data structure can be fixed by copying the relevant elements together into a new memory block as seen in listing 3. Instead of copying the elements, the striding transformation modifies the function to support arbitrary strided data layouts. To achieve this, the index of every access inside the function has to be multiplied with the striding factor to access the correct element. For example the 5th color value on the mentioned RGB image is not at the 5th position, instead the correct strided position is 15 $(3 * 5)$.

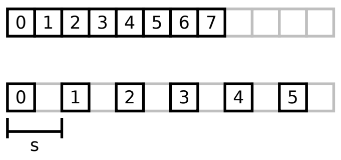

Figure 2: Stride access pattern with a striding factor of 2

```
for(int x = 0; x < size; x++) {
  red_channel[x] = rgb_image[x*3];
}
f(red_channel, dst, size);
```

Listing 3: Logic of a input striding on a RGB image with striding factor 3

**Example.** Again a RGB image with interleaved color channels is a good example. A gray-scale image filter which expects an image with one color channel can not directly work on the red color channel of an interleaved RGB image. With the striding transformation, it can be specified that the data is strided 3 elements apart and the function can work directly on the interleaved RGB image.

### 2.1.3 Permuting

The third transformation permutes the elements either before or after the application of the function. Is uses a lookup table to translate the original index into the permuted index. The modifications happen at the same place as the striding transformation, but instead of multiplying the index by the striding factor, the index is used to access the permutation vector to get the permuted index. The listing 4 shows an example of the logic of the transformation of an input permutation.

```
for(int x = 0; x < size; x++) {
  permuted[x] = src[permutation[x]];
}
f(permuted, dst, size);
```

Listing 4: Logic of a input permutation

**Example.** A bit-reversal permutation is a permutation, where the bit reversed index the permuted index is. This permutation is important for radix-2 Cooley-Turkey FFT algorithms [10], where recursive steps need a bit-reversal permutation of the input and output data.

An index can be split up into single bits with

$$i = \sum_{j=0}^{n-1} a_j 2^j. \tag{1}$$

The reversed index counts the exponent of the base 2 down from $n-1$ to 0 resulting in

$$\tilde{i} = \sum_{j=0}^{n-1} a_j 2^{n-1-j}. \tag{2}$$

For example the index 5 (binary 0101) with 4-bit indices will get index 10 (binary 1010). This permutation can be precomputed into a permutation vector and passed to the transformed function. The function then operates on the bit-reversed permutation of the data.

### 2.1.4 Blockstriding

The blockstriding transformation is a more general case of the striding transformation. The striding transformation has exclusively blocks of one element strided apart. The blockstriding transformation additionally allows arbitrary sized blocks of relevant elements. A block size of one is the same as the striding transformation.

With a given striding factor and block size, 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 block size $b$ and the stride $s$, the blockstrided position $i_2$ of the same element can be calculated as

$$i_2 = \underbrace{\lfloor \frac{i}{b} \rfloor}_{\text{block number}} * s + \underbrace{i \bmod b}_{\text{position inside block}}. \tag{3}$$

$\lfloor \frac{i}{b} \rfloor$ calculates in which block the element is located and $i \bmod b$ is the element's position within the block. Listing 5 shows the logic of the transformation. The variable `block_size` represents the chosen block size, meaning how many elements are next to each other, and `stride` is the number of elements from the start of one block to the start of the next block. An example of block size 2 and stride 4 is shown in figure 3.

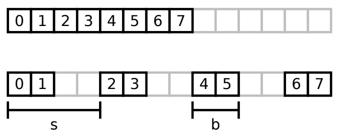

Figure 3: Blockstride access pattern with $b = 2$ and $s = 4$

```
for(int x = 0; x < size; x++) {
  int block_num = x / block_size;
  src[x] = input[block_num * stride + x % block_size];
}
f(src, dst, size);
```

Listing 5: Logic of a blockstride transformation

The index of every access is used to calculate the blockstrided index based on the formula 3.

The blockstrided index is then used to access the vector.

**2D optimized blockstriding.** The generic blockstriding transformation can be applied to every function but needs quite a lot of extra computation for each access. A division, modulo, multiplication and addition is needed to calculate the correct index. Therefore, functions with many accesses can be quite slow with the transformation. Additionally, many functions which work on 2D vectors already use an implicit block size and striding factor and for that reason the additional computations of the generic blockstriding transformation can be omitted. The width of a 2D vector represents the block size and the striding factor at the same time. There are no non-relevant elements between the blocks if the block size and the striding factor are equal which it is the case for an image.

To allow the function to work on smaller part of the 2D vector without copying the data into a smaller buffer (like in figure 6), the block size has to be modified but the striding factor must be preserved. For this reason, the 2D optimized blockstriding transformation separates the block size and the striding factor and adds the striding factor to the parameter list. In order to properly add the striding factor, the function must use the special indexing style `x + y * width`, where `x` is the column and `y` is the row of the 2D access. The complete restrictions are described in the implementation of the transformation in chapter 4.1.5.

```
for(int y = 0; y < height; y++) {
  for(int x = 0; x < width; x++) {
    part[x + y * width] = input[x + y * buffer_width];
  }
}
f(part, dst, width);
```

Listing 6: Logic of a 2D optimized blockstride transformation

**Example.** Any image filter function suits the blockstriding transformation well. With the blockstriding transformation, the filter can work on smaller blocks of the input image. The added striding factor has to be the width of the image and the block size is the chosen block width to process. Furthermore, the width and a height parameter, which a 2D filter usually has, must represent the size of the smaller block of the image. As an example, the listing 7 shows the invocation of a transformed filter. The parameters are chosen in order to process the top left quarter of the image. The `width` and `height` parameter must match the amount of elements to process because the transformation only changes the indexing and these two parameters are usually used in loop termination conditions.

```

f(src,                // source pointer
  dst,                // destination pointer
  width / 2,          // width: block width
  height / 2,         // height: block height
  width / 2,          // block_width
  width);             // stride: image width
```

Listing 7: Example application of the blockstriding transformation

If the function uses the needed indexing style, the 2D optimized blockstriding transformation can be used and the duplicated block width parameter is not added. The only added parameter is the striding factor which is the image width in this example. The result is an image filter which can operate on parts of an image and is as fast as the original function as there are no additional computations.

## 2.2 Combinations

Combinations of transformations are not directly supported by the developed tool, however most transformations can be applied one after the other to have multiple transformations in one function. The transformations can be applied in any order and multiple times. The only limitation is the 2D optimized blockstriding transformation, as it needs the indexing style `x + y*width` described in the chapter 2.1.4. The permuting transformation destroys this indexing style by replacing the index with an access to the permutation vector and therefore a 2D optimized blockstriding can not be applied after a permuting transformation have been already applied.

### 2.2.1 Implications on Subsequent Transformations

Different sequences of the same set of transformations produce different transformed functions as the transformation are not distributive. Some of these transformation sequences have influence on the performance, others affect the added parameters of the following transformations. The most important implications are described in the next few paragraphs.

**Scaling.** The only transformation which does not affect any following transformation is the scaling transformation. It adds one floating point operation for every access and does not modify any indexes. 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. This affects subsequent transformations, as they are dependent on the used index. For example a permutation vector has to be strided if the permuting transformation is applied after the striding transformation. Furthermore, packed vectorized accesses will be split up into single slot accesses increasing the total access count. The subsequent transformations also operate on the increased access count and introduce a bigger overhead as if they are 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 style is `perm[i]`, where `i` is the previously used index. This style is not compatible to the 2D optimized blockstriding transformation as mention in the chapter 2.2 and therefore the 2D optimized blockstriding transformation can not 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 for the blockstriding transformation. The subsequent added parameters are blockstrided instead of strided. Further, the packed vectorized accesses are split into single slot accesses and the number of accesses is increased accordingly.

### 2.2.2 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 giving a better performing function. Consecutive scaling, striding or permuting transformations can be easily combined into one transformation. If two consecutive 2D optimized blockstriding transformations are applied, the first transformation has no effect on the function at all, because the second transformation replaces the parameter which was added by the first transformation.

To combine scaling or striding transformations, the factors are multiplied together to get the factor for the combined transformation. For the vector scaling transformation, every element of the scaling vector has to be multiplied either by the scalar scaling factor or by the corresponding element in the second scaling vector.

Two permuting transformations can be combined by getting the first permuted index for every index and get the permuted index from the second permutation vector by using the first permuted index. When `perm1`, `perm2` are the two consecutive permutation vectors, the combination is `perm3[i] = perm2[perm1[i]]`.

# 3 Sufficient Conditions

The tool requires several restrictions on the C language for the input function to work properly. Most of them limit the complexity of the language in order to make the development of the tool manageable in the given time. These restrictions are listen in the chapter 3.2. The more important limitations enable the reasoning on important properties which are needed for the transformations and are explained in the next section.

If any unsupported elements are used, the tool has an undefined behavior. It can result in a valid function with correct or wrong output, invalid C code or a crash of the tool.

## 3.1 Correctness Ensuring Limitations

The limitations described next are used in section 5, where two properties are discussed which enable the transformation to work correctly.

*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. An assignment to `p`, where `p` and `q` are two different pointers, must occur exactly once in the function, although it can be executed many times. This is known as single static assignment (SSA) from compiler design. If `p` and `q` are the same pointer (i.e. `p = p + index`), the assignment is allowed multiple times.

The 2D optimized blockstriding transformation requires a *more specific assignment style*. The assignment must be of the form `p = q + x + y * width` where `p` and `q` are again pointers, `x` and `y` are expressions with an integer type without containing `width` and `width` is an integer parameter or a variable depending on it.

To enable 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 can not determine if an access is already scaled because the accessed value could have been written a scaled value beforehand.

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

## 3.2 Limitations to Reduce Development Time

The C language offers a variety of syntactic elements and control statements. As the time is limited to develop a working tool, the supported language is subject to many more restrictions. The allowed elements are listed in this chapter.

**Function declaration.** The function which the transformation should be applied has to conform the three following points:

- The **return value** can be `void`, `double` or `float`.
- There are no restrictions on the **function name**.
- 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 which store the

length of a vector must be of `int` or `size_t` type, remaining arguments can be any scalar data type.

**Keywords.** Allowed keywords are `break`, `continue`, `const`, `register`, `return`, `char`, `short`, `int`, `long`, `float`, `double`, `if`, `else`, `for`, `signed`, `unsigned`, `sizeof`, `void`, `volatile`

**Global and local variables.** The limitations on global and local variables are the same as the limitations on the functions arguments. Additionally `_mm128` and `_mm128d` data types are allowed to enable vectorized code.

**Syntax.** Allowed are only a few constructs. Namely *for loops* with one iteration variable, *if/else* constructs, *variable declarations* and *assignments*. Assignments are only restricted when the left hand side is any kind of pointer and is described in chapter 3.1.

**Pointers.** Due to the nature of the transformations, the most restrictions are on pointers. The following list contain all the restrictions which apply to 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. Supported load intrinsics are `_mm_load_ss`, `_mm_load_sd`, `_mm_load_ps`, `_mm_load_pd`, `_mm_loadh_pd`, `_mm_loadl_pd`, `_mm_loadu_ps`, `_mm_loadu_pd` and supported store intrinsics are `_mm_store_ss`, `_mm_store_sd`, `_mm_store_ps`, `_mm_store_pd`, `_mm_storeh_pd`, `_mm_storel_pd`, `_mm_store1_pd`, `_mm_store1_ps`, `_mm_storeu_ps`, `_mm_storeu_pd`.

**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 Implementation

## 4.1 Transformations

In this chapter, the transformations are described in detail how they are implemented. For every transformation, the affected vectors can be chosen and the transformation logic is applied for every chosen vector.

### 4.1.1 Scaling

The scaling is divided into prescaling and postscaling. Prescaling applies when data is read from a vector, postscaling when data gets written to a vector. The steps for a prescaling transformation are:

1. First, the tool creates a list of vectors which have to be handled. This step uses the dependency graph to check the dependencies of the to-scale parameters and store every dependent pointer of them. Additionally, new variables are inserted to track the total offset when a pointer is assigned.
2. With the list of variables which need to be handled, it first modifies the normal array subscripts (pointer dereferences are equivalent to array subscripts).
   - For a **scalar scaling** an access `p[i]` gets replaced with `p[i] * scaling_factor`
   - If it is a **vector scaling** transformation, the tool needs to calculate the total offset of the access and use it as an index for the scaling vector. The total offset can be obtained due to the restrictions given in chapter 3.1. The 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 created in step 1.
3. After all array subscripts are transformed, it 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 a SSE register.
   (a) The first step is to isolate the load intrinsics and assign the result to a temporary variable `tmp`
   (b) If a vector scaling transformation is applied, two or four new scaling factors have to be 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) Afterwards, 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 very 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.

### 4.1.2 Striding

The striding transformation moves the relevant elements by a striding factor apart. In contrast to the scaling transformation, the striding transformation does not modify any data, it modifies solely indices. The moving of the relevant elements is achieved by multiplying the total offset of an access by the striding factor. The detailed steps are:

1. The first step is checking the dependencies. A dependency of vectors are based on an assignment `p2 = p + offset`, from which can be concluded that `p2` has a by `offset` bigger total offset than `p`. This offset has to be multiplied with the striding factor resulting in `p2 = p + offset * stride`. This modification ensures that a strided index of an access to any vector accesses is correct.

2. After the dependencies are handled, every relevant array subscript (or dereference) is modified. The index of an access is again multiplied with the striding factor resulting in the correct strided element access due to the modification in the first step (`p2[index]` gets replaced with `p2[index * stride]`)

3. Vectorized code is more complicated as the data elements, which were consecutive in memory before the transformation, are strided apart. Many intrinsics load/store multiple elements which are consecutive in memory. 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 2 or 4 single slot intrinsics. The single slot intrinsics use the address from the previous step. For each single slot load/store intrinsic, the address is increased by the striding factor to get the subsequent element.

   (c) After loading the single elements, shuffle intrinsics combine the single elements into one SSE register.

The intrinsic call `x1 = _mm_load_ps(src + x + y * width)` has an offset of `x + y*width`. This call will get transformed into four single slot loads with shuffling shown in listing 8. Storing strided elements are done by using single slot store intrinsics and a shuffle intrinsic to rotate the elements in the register one element to the right. An example of the transformation of `_mm_store_ps(dst+x+y*width, x1)` is shown in listing 9.

```
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;
```

Listing 8: example of striding transformation of `x1 = _mm_load_ps(src + x + y * width);`

```
a_0 = x1;
_mm_store_ss(dst + stride * (x + y * width + 0), a_0); // store 1st element
a_0 = _mm_shuffle_ps(a_0, a_0, 57);                    // rotate 1 element right
_mm_store_ss(dst + stride * (x + y * width + 1), a_0); // store 2nd element
a_0 = _mm_shuffle_ps(a_0, a_0, 57);                    // rotate 1 element right
_mm_store_ss(dst + stride * (x + y * width + 2), a_0); // store 3rd element
a_0 = _mm_shuffle_ps(a_0, a_0, 57);                    // rotate 1 element right
_mm_store_ss(dst + stride * (x + y * width + 3), a_0); // store 4th element
a_0 = _mm_shuffle_ps(a_0, a_0, 57);                    // rotate 1 element right
```

Listing 9: example of striding transformation of `_mm_store_ps(dst+x+y*width, x1);`

### 4.1.3 Permuting

The steps of the permuting transformation are very 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 vector. To access the permutation vector, the total offset is needed, as it has to be the exact same index as used with the vector. Therefore additional variables are added to track the offset of every variable. For every assignment of a pointer (`p2 = p + offset`) an additional offset variable `p2_offset` is inserted. The inserted statement to be able to calculate the offset of `p2` is `int p2_offset = p_offset + offset`.With those additional variables, the tool can easily determine the total offset of an access of any pointer and use this offset to get the permuted index. An access to `p2[value]` has a total offset of `p2_offset+value` and is used to access the permutation vector.

An access to `p2` has an implicit offset included from its origin. As the permutation vector stores the indices of the elements in the parameter, which has a zero offset, the implicit offset of `p2` has to be removed to access the correct permuted element. Therefore, the pointer `p2` is replaced with its origin resulting in `origin[perm[p2_offset + value]]`.

```
a_0 = _mm_load_ss(x + x_perm[((j) + p_offset_0 + 0)]);
a_1 = _mm_load_ss(x + x_perm[((j) + p_offset_0 + 1)]);
a_2 = _mm_load_ss(x + x_perm[((j) + p_offset_0 + 2)]);
a_3 = _mm_load_ss(x + x_perm[((j) + p_offset_0 + 3)]);
a_0 = _mm_shuffle_ps(a_0, a_1, 0);
a_2 = _mm_shuffle_ps(a_2, a_3, 0);
a_0 = _mm_shuffle_ps(a_0, a_2, 136);
x1 = a_0;
```

Listing 10: example of permuting transformation of `x1 = _mm_load_ps(p+j);`

Packed SSE load and store intrinsics are split up into their single slot equivalent intrinsics like in the striding transformation as the permuted elements are not consecutive in memory anymore. Again the same logic as with scalar accesses is used. Instead of multiplying the index, the total offset of the access is used to index the permutation vector. An example of `_mm_load_ps` can be seen in the listing 10. The intrinsic `_mm_load_ps` loads 4 elements at once, therefore 4 consecutive elements of the permutation vector are used to get the permuted indices. Again, the single values loaded with the single slot intrinsics have to be merged together into one SSE register with shuffling intrinsics.

### 4.1.4 Blockstriding

The blockstriding transformation is a more general case of the striding transformation and can handle arbitrary block sizes. For vector accesses, the formula (3) is used to calculate the correct index. With the total offset of the access as $i$, the final index $i_2$ can be calculated and used as an index to the origin of the accessed vector. The block size and the striding factor are added to the parameter list to use in the formula (3).

Handling of vectorized code is more complicated than in the striding transformation. In the striding transformation, every packed load or store has to 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. These cases depend on the alignment of the parameters, block size and 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. The three different cases are listed in table 1. 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 use 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 according code block. This branching allows to use higher performing code if the alignment conditions are fulfilled.

| 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

### 4.1.5 2D Optimized Blockstriding

The most complex transformation is the 2D optimized blockstriding. A function that operates on a 2D vector usually uses the full width and height of the vector. These two values are often passed as integer parameters. If a developer needs the function to only operate on a part of the 2D vector, it is not enough to just change the width parameter. If the function has accesses across rows, it still needs to consider the full width of the buffer to skip a full row despite only working on a smaller width. The transformation adds a new parameter to pass the buffer width and the original width parameter is then used to pass the smaller width to the function. The 2D optimized transformation only works on 2D vectors with a special indexing style described in the next paragraph.

**Indexing Style.** The index must contain an integer parameter or a variable which is dependent on a integer parameter (namely the width parameter). If this parameter is contained multiple times in the calculation, every occurrence is replaced. A typical index calculation looks like `x + y * width`, where `width` is the integer parameter (block size and striding factor) and `y` is a loop counter. The expression `y * width` is the row calculation which has to be modified to allow the function to work on a smaller width than the buffer width.

```
void f(const double* input_img, double* output_img, const int width, const int
    height) {
    for(int y = 1; y < height-1; y++) {
        for(int x = 1; x < width-1; x++) {

            int index = x+y*width;

            double tmp = input_img[index-width] + input_img[index+width] +
                input_img[index-1] + input_img[index+1] + input_img[index] * 2;
            output_img[index] = tmp / 6;
        }
    }
}
```

Listing 11: example to show blockstride transformation

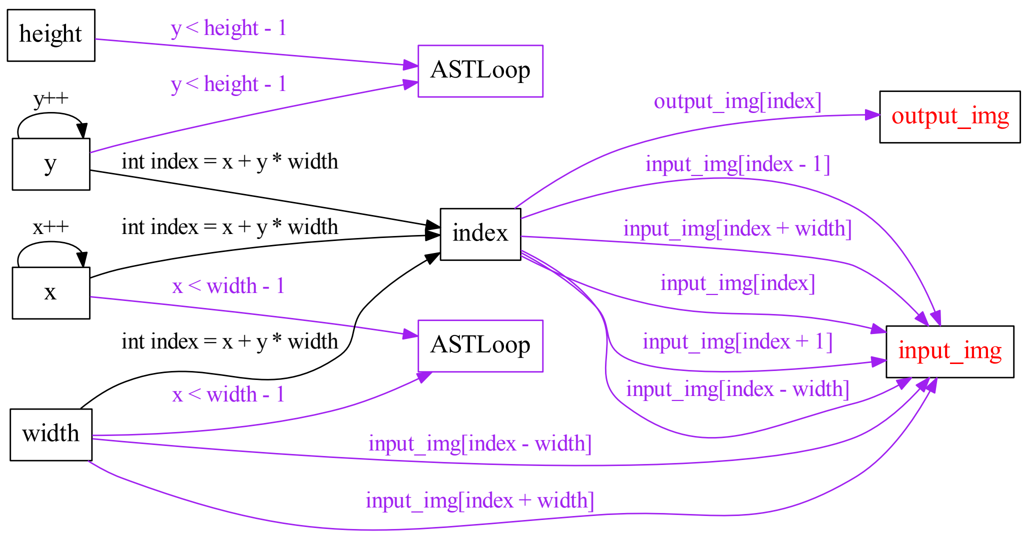

Figure 4: Dependency graph of listing 11

To show the internals of the transformation the example in listing 11 is used. It is a simple average filter without any border handling which uses the indexing style needed by the transformation. The function is modified in order that only a part of the vector `input_img` is processed. The vector `output_img` is already of the smaller size and does not need any modifications in the function. The detailed steps to modify the function to work on a smaller area of `input_img` are:

1. At first, the width parameter of the vector has to be found out. It uses a heuristic approach, shown in figure 5, to find the width of the vector. In the example, the parameter `int width` is found as the width of `input_img` after the subscript refinement step.

2. Next step is to check the usages of this parameter (on line 5 and 7 in the example). The dependency graph is used to find the three occurrences of `width`. Figure 6 shows the generated dependency graph from the listing 11. The dependency graph is explained in chapter 4.2. The four outgoing edges of the node `width` indicate, that the variable is used in the calculation of the variable `index`, in a loop condition and in array subscripts of `input_img`.

   The dependency graph is used to handle cases where the width parameter is used in non-relevant vectors (in this example `output_img` via the node `index`). For this reason, all paths between the relevant vector (`input_img`) and the width parameter (`width`) are marked in a first step. There are two paths directly from `input_img` to `width` and several paths via `index`.

3. After the marking, the graph looks like in figure 6. The green nodes and edges are marked. To check if `width` is used in non-relevant parts of the function, every outgoing marked edge of `width` has to be checked. For every of these edges, it is checked if any marked node on any path to `input_img` has any unmarked edges, meaning if a variable is used somewhere else without needing the changes of the transformation. In the used example, the two occurrences of `width` on line 7 are replaced as there is no unmarked out-edge from the node `input_img`. On

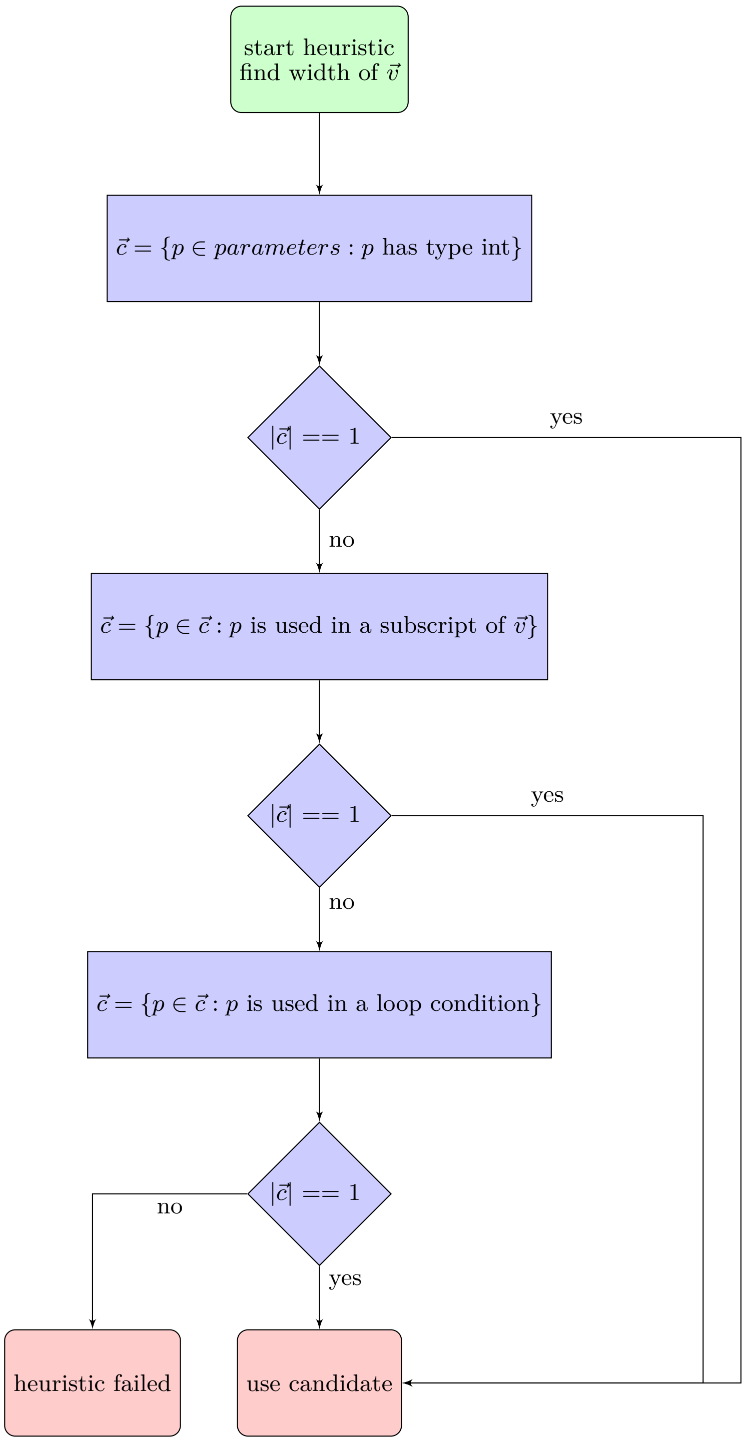
start heuristic
find width of $\vec{v}$
$\vec{c} = \{p \in parameters : p \text{ has type int}\}$
$|\vec{c}| == 1$
yes
no
$\vec{c} = \{p \in \vec{c} : p \text{ is used in a subscript of } \vec{v}\}$
$|\vec{c}| == 1$
yes
no
$\vec{c} = \{p \in \vec{c} : p \text{ is used in a loop condition}\}$
$|\vec{c}| == 1$
no
yes
heuristic failed
use candidate

Figure 5: Heuristic flow chart of the blockstride transformation

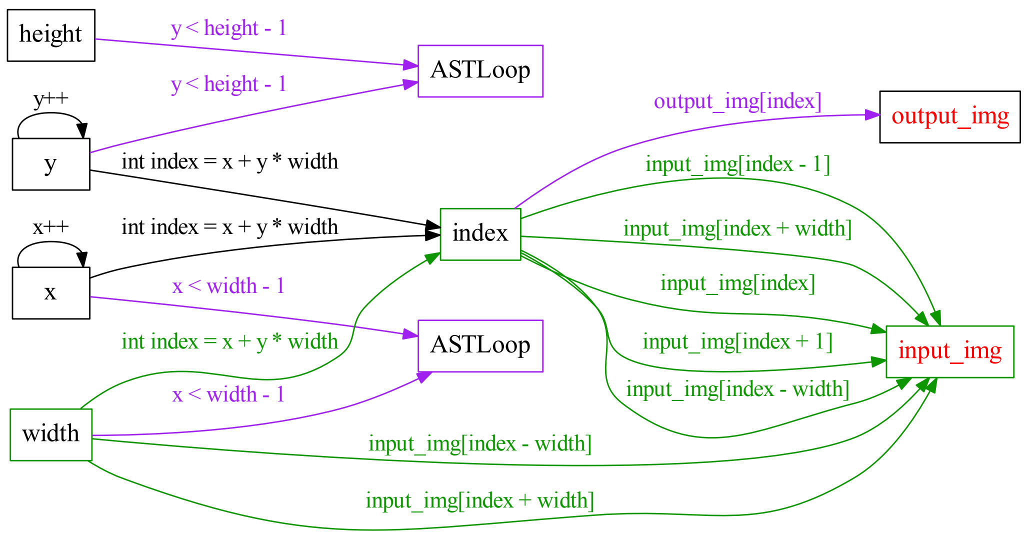

Figure 6: All paths between `width` and `input_img` have been marked in the dependency graph of listing 11.

line 5, `width` is used to calculate the variable `index`. The node `index` has an unmarked out-edge to `output_img` (used as index in `output_img`) and therefore, the variable has to be duplicated to not change the index of the subscript of `output_img` on line 8. The transformation creates a new variable, modifies `width` and replaces `index` with the newly created variable for every marked out-edge of the node `index`. This keeps the usage of `index` in line 8 unaltered

The result of the transformation of the example in listing 11 is shown in listing 12. On line 4 is the duplicated `index` variable with the adjusted width parameter. Furthermore, the two usages of `width` on line 6 got replaced by `buffer_width`.

```
 1  void f_prime(const double* input_img, double* output_img, const int width, const
        int height, const int buffer_width) {
 2      for(int y = 1; y < height - 1; y++) {
 3          for(int x = 1; x < width - 1; x++) {
 4              int index_0 = x + y * buffer_width;
 5              int index = x + y * width;
 6              double tmp = input_img[index_0 - buffer_width] + input_img[index_0 +
                    buffer_width] + input_img[index_0 - 1] + input_img[index_0 + 1] +
                    input_img[index_0] * 2;
 7              output_img[index] = tmp / 6;
 8          }
 9      }
10  }
```

Listing 12: example to show blockstride transformation

At the end, every usage of `width` which influences `input_img` are replaced with `buffer_width` and the function processes only the smaller width with still considering the full buffer width.

## 4.2 Dependency Graph

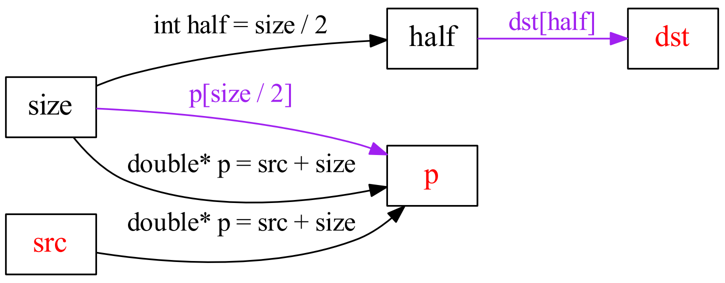

Figure 7: Dependency graph generated from listing 13

As described in the previous section, several transformations need the total offset and the origin of a vector access. Furthermore, the 2D optimized blockstriding transformation also needs the information of integer variable usage in index calculations to determine the correct width parameter. To provide this information, a dependency graph for every input function is created. The nodes of the graph represent the variables used in the function. The edges between the nodes are statements which create a dependency between the variables. The tracked properties are:

1. Dependencies between pointers from assignments of style `p2 = p1 + index` (`p2` is dependent on `p1`)
2. Dependencies between integer variables from assignments. The left hand side variable is dependent on every variable occurring on the right hand side of the assignment.
3. Usages of integer variables throughout the function. For example the vector access `v[x*width + y]` uses `x`, `y` and `width` as an index and is shown as purple edges

From the example function in listing 13 the graph in figure 7 is generated. The first line creates the dependency between `p` and `src`. The second line creates the dependency between the two integer variables `half` and `size`. `half` and `size` are used in the array accesses in the last line of the example, which is shown as the purple edge.

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

Listing 13: Example to show a minimal dependency graph

# 5 Analysis

In this chapter, formal reasoning is given to proof the properties needed by the transformations. The properties are described in the next chapter and the reasoning of the properties over single statements are given in chapter 5.2. The reasoning over single statements is then combined in order to proof that the properties are fulfilled over a whole function.

## 5.1 Properties

The tool basically relies on 2 properties which enable the transformations to be correct. The first property is, that the tool needs to know the origin of every pointer variable. The origin of a pointer is the parameter on which the pointer depends on. The second property is the total offset, which is the amount of elements between the pointer to its origin. To show that these two properties can be fulfilled, the restrictions on the input language in section 3.1 are needed.

The listing 14 is a example of a function which copies a 2 dimensional vector. It is used to show the application of origin and offset. On line 3 the pointer `row` is assigned. As `input` is the pointer on the right hand side of the assignment, row is depending on it and because `input` is a parameter, the origin of `row` is `input`. The offset of `row` is `y*width` as it is the difference between `row` and `input`. The next assignment is on line 5. The origin of the pointer `element` is the same as the origin of `row` and the offset of `elements` is the offset of `row` plus the variable `x`.

The offset of parameters are assumed to be 0 except if recursion is used. An additional parameter is then used to pass the offset during recursion. The origin of a parameter is the parameter itself.

```
 1 void f(double* input, double* output, int width, int height) {
 2   for(int y = 0; y < height; y++) {
 3     double* row = input + y * width;
 4     for(int x = 0; x < width; x++) {
 5       double* element = row + x;
 6
 7       output[x+y*width] = *element;
 8     }
 9   }
10 }
```

Listing 14: Example of a function to show the origin and offset property

## 5.2 Reasoning

To be able to reason about a function whether the transformation works or not, Hoare logic [5] is used. Hoare logic uses Hoare triples of the form $\{P\}C\{Q\}$ where $P$ is the precondition, $Q$ the postcondition and $C$ a command. In order that the transformations work, at every state of the function the origin and offset of the used pointer variables must be known. For this reason, the two properties are defined to provide the needed information. The first property is denoted as $\mathcal{O}$ and contains the origin of every pointer variable. Two operations are defined, $\mathcal{O}(p)$ is the origin of pointer $p$ and $\mathcal{O}[p = o]$ sets the origin of pointer $p$ to $o$. The second property is $\mathcal{I}$ and contains the offset of every pointer variable. Again $\mathcal{I}(p)$ is the total offset of $p$ and $\mathcal{I}[p = o]$ sets the offset of $p$ to $o$. Both properties are fulfilled if at the execution state the offset and origin of every used pointer variable is known.

### 5.2.1 Axioms

In this section, it is shown that for every axiom the two properties $\mathcal{O}$ and $\mathcal{I}$ are fulfilled.

A **skip** statement is a command, which does not have any influence on the state and therefore preserve the properties $\mathcal{O}$ and $\mathcal{I}$. The corresponding axiom for a pointer assignment is

$$\frac{}{\{\mathcal{O},\mathcal{I}\}\ \textbf{skip}\ \{\mathcal{O},\mathcal{I}\}}. \tag{4}$$

A pointer assignment is the only command, which modifies the set of pointer variables. Therefore, every other statement is seen as a skip statement. The axiom is

$$\frac{}{\{\mathcal{O},\mathcal{I}\}\ p_2 = p_1 + i\ \{\mathcal{O}[p_2 = \mathcal{O}(p_1)], \mathcal{I}[p_2 = \mathcal{I}(p_1) + i]\}}. \tag{5}$$

According to the limitation in 3.1, there is exactly one pointer assignment, where $p_2 \neq p_1$ is true. Therefore, this assignment is the only possible source of the origin for $p_2$. The origin of $p_2$ is the origin of $p_1$ and the offset of $p_2$ is the offset of $p_1$ plus $i$.

The second allowed pointer assignment is $p_1 = p_1 + i$. The origin is not modified and the index is increased by $i$ by this pointer assignment. The resulting axiom is

$$\frac{}{\{\mathcal{O},\mathcal{I}\}\ p_1 = p_1 + i\ \{\mathcal{O}, \mathcal{I}[p_1 = \mathcal{I}(p_1) + i]\}}. \tag{6}$$

### 5.2.2 Decomposition Rules

The decomposition rules separate whole function into simpler rules until an axiom can be applied.

$$\frac{\{P\}\ S\ \{Q\}\,,\ \{Q\}\ T\ \{R\}}{\{P\}\ S;\ T\ \{R\}} \tag{7}$$

$$\frac{\{P \land B\}\ S\ \{Q\}\,,\ \{\neg B \land P\}\ T\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{endif}\ \{Q\}} \tag{8}$$

$$\frac{\{P \land B\}\ S\ \{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{done}\ \{P\}} \tag{9}$$

Additionally to the common rules above, a rule to handle a for-loop is needed as the only allowed loop structure is a for-loop. The rule

$$\frac{\{P\}\ B\ \{P\}\,,\ \{P\}\ \textbf{while}\ C\ \textbf{do}\ S\ ;\ I\ \textbf{done}\ \{P\}}{\{P\}\ \textbf{for(}\ B;\ C;\ I)\ \textbf{do}\ S\ \textbf{done}\ \{P\}} \tag{10}$$

is used to decompose a for loop into a single statement and a while loop.

### 5.2.3 Initialization of the properties

At the beginning of a function, the two properties are initialize with the available parameters. The origin of a parameter is the parameter itself ($\mathcal{O}[p = p]$) and the offset is either 0 ($\mathcal{I}[p = 0]$) or the offset passed in the corresponding integer parameter ($\mathcal{I}[p = \text{p\_offset}]$). This parameter is only present if recursion is used.

# 6 Experiments

The main goal of the tool is to increase the performance of the transformed functions. For this reason, comprehensive tests are made to evaluate the gained speedup.

The runtime of the resulting transformed function and the original function with code added to simulate the transformation (called reference) are measured and compared. Furthermore, functional tests are done to check the correctness of the transformations.

## 6.1 Experimental Setup

All performance measurements were made on a computer with the following specifications:

| | |
|---|---|
| Processor | AMD Phenom II X4 Deneb @ 3.4 GHz |
| Memory | 8 GiB DDR3 1333MHz |
| OS | Windows 7 64bit |
| Compiler | Intel(R) C++ Compiler XE IA-32, Version 12.0.0.104 |
| Compiler flags | /O3 /arch:SSE2 /fp:fast=2 /qipo_fas /Oa /Ow /Ob2 /Oi /Ot /fast /xHost |

The compiler flags are quite important. For example omitting the flag `/fast` affects the block-striding transformation enormously, resulting in a similar runtime as with the Microsoft compiler seen in appendix B.

## 6.2 Test Cases

Various input functions are used to measure the performance impact of the transformation. The first part of the functions are hand written, the remaining functions are generated by spiral [8]. In the table 2 is a short explanation of every test function, the used data type and a list of transformations which are not available due to limitation violations.

## 6.3 Functional Tests

Functional tests are done to verify the correctness of the transformations. Additionally to every performance test in the next chapter, the correctness of the output of the transformed function is verified. A test case is correct if the output of the transformed function equals the output of the reference.

Except for minor floating point rounding errors due to a different ordering of floating point operations were the output exactly the same. Therefore, the transformations on the used test cases are functional and are valid.

### 6.3.1 Combination Functional Tests

Random combinations of transformations are applied to check if the added source code by the transformations is correct. Two sets of transformations are used to create random combinations. The first set contains all transformation except the 2D optimized blockstriding transformation and the second set consists of all transformation except the permuting transformation. The permuting transformation is not compatible to the 2D optimized blockstriding transformation as it

| Function | Description | DataType | Invalid Transformations |
|---|---|---|---|
| **filter** | Generic convolution with a kernel of size 3x3 | float | - |
| **contrast** | Adjust the ranges from the minimum to the maximum value in an image. The calculation for every pixel is $\frac{input-min}{(max-min)*new\_max}$ | double | - |
| **sse_contrast** | vectorized version of the test 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, 2d blockstriding |
| **reduce** | Sum reduction of all elements | double | postscaling, 2d blockstriding |
| **rec_reduce** | 2-way divide and conquer version of reduce | double | postscaling, 2d blockstriding |
| **scan** | Calculates the prefix sum of a vector | double | postscaling, poststriding, 2d blockstriding |
| **dft512** | Discrete Fourier transformation for 512 elements generated by spiral | double | postscaling, 2d blockstriding |
| **dft1048576** | Discrete Fourier transformation for 1 048 576 elements generated by spiral | double | postscaling, 2d blockstriding |
| **fftfwd** | Fast Fourier transformation for 128 elements generated by spiral | double | 2d blockstriding |
| **sse_fftfwd** | Vectorized fast Fourier transformation for 128 elements generated by spiral | double | 2d blockstriding |
| **dct40** | Discrete cosine transformation for 40 elements generated by spiral | double | 2d blockstriding |

Table 2: List of the used functions in the performance measurements

changes the needed indexing style to perform the transformation and therefore the two sets of transformations are used in separate test runs.

A combination of up to 40 transformations worked with both sets of transformations without any compilation error. At that stage, either the processing time to transform the function gets really large (especially for a vectorized input function as the blockstriding transformation multiplies the code size by factor of three) or the parsing library raises an assertion error during the parsing of the file (note: a compilation error is not raised, which would be the case if the given source code contains an error).

## 6.4 Performance Tests

For the performance tests, each test case is run 10 times to compensate variations on the runtime. For test cases with variable input size, the input size varies from small (few megabytes) to large (few gigabytes), each is again run 10 times. To see if the transformation actually perform better, the transformation is simulated in an extra step before or after the untransformed function call which is referred as reference. The testing framework measures the time taken to execute the transformed function and the reference. All the performance plots show the factor $\frac{\text{runtime reference}}{\text{runtime transformed function}}$. Values smaller one mean the transformed function runs slower than reference, larger one means it runs faster.

This chapter shows the performance impact of the developed transformations.

### 6.4.1 Overall Performance

In this chapter, the performance of the test cases for every transformation is shown. All data used to call the functions is randomly generated.

**Scaling.** Figure 8 shows the performance impact of the scalar scaling transformation. The scaling parameter is randomly chosen for each test run. The results of the transformation are all equally fast or faster than the reference. This is not really surprising as the transformation increases the operation count per element by one but does not increase the needed memory bandwidth as opposed to the reference, which increases the needed memory bandwidth.

The spiral generated functions (*dct40*, *dft1048576*, *dft512*, *fftfwd*, *sse_fftfwd*) do benefit less than the hand-written, in average these functions run 22% faster than the reference. These functions are highly optimized and run already at a top performance, therefore the possible speed up is more limited as for the handwritten test functions.

The results of the vector scaling transformation are shown in figure 9. It shows a similar but not as stable results than the scaling transformation. *filter* and *dft1048576* are the only test cases which run slower than the reference. The runtime of the different test runs of *sse_mvm* differ too much if only the output or the input is scaled and therefore the range of the results is quite large from a light slowdown to a 5 times faster runtime.

The average runtime over all test cases is 1.64 times faster for scaling and 1.30 times faster for vector scaling.

**Striding.** Figure 10 shows the performance change with the striding transformation. The striding test cases use a striding factor of 8. The results below one is mainly due to the change of the access pattern, resulting in less optimal code for the cache hierarchy. For *rec_mergesort* the

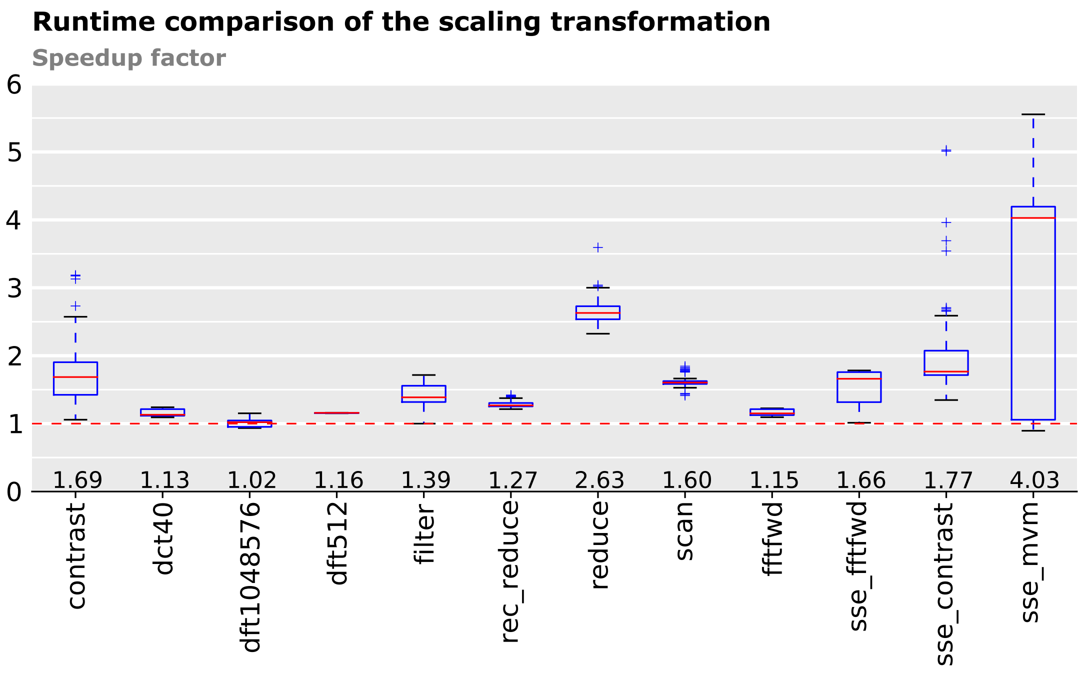

Figure 8: Speedup of scaling transformation

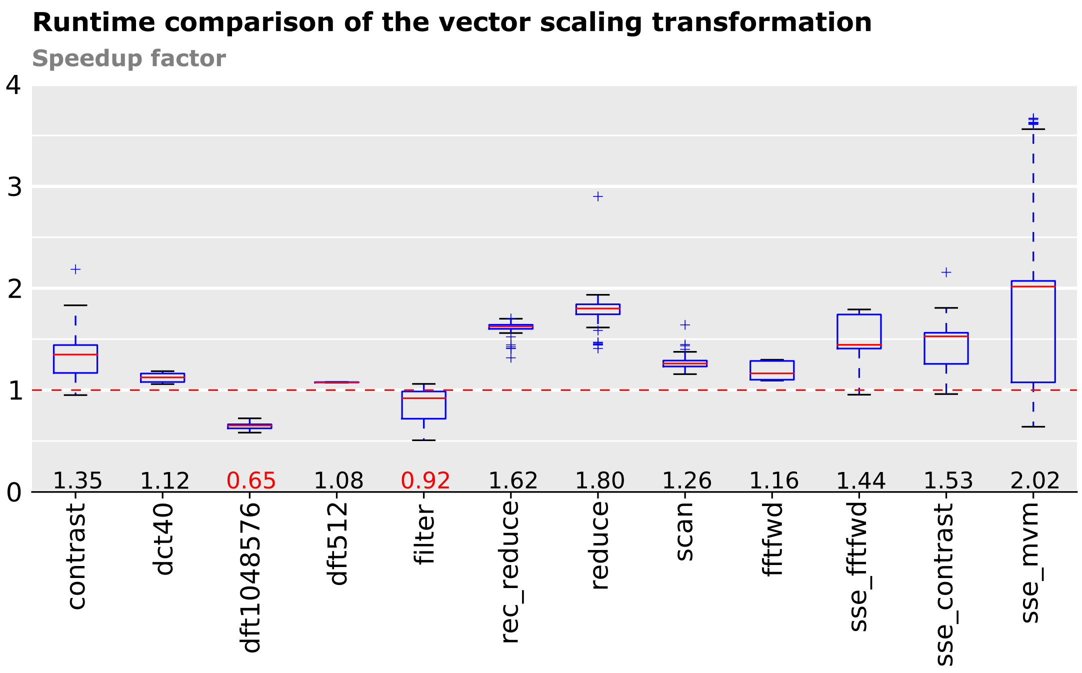

Figure 9: Speedup of vector scaling transformation

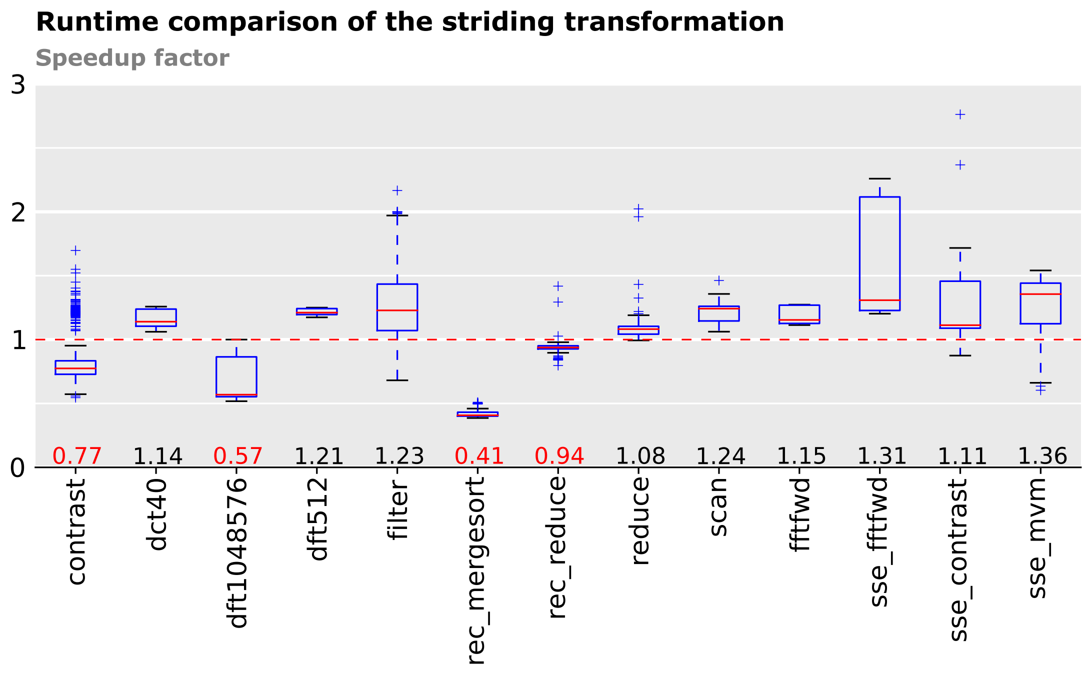

Figure 10: Speedup of striding

transformed function results in a runtime more than twice as long as the reference. A possible cause can be its larger amount of accesses ($\log(n)$ accesses for each element compared to usually $\mathcal{O}(1)$) combined with less cache effect because the elements are further apart.

More interesting are the results of the vectorized tests. All three vectorized tests seem to benefit from the transformation quite well, which is surprising as striding forces the usage of single slot load and store intrinsics. A possible cause might be that modern architectures handle single slot operations quite efficient and the slowdown using single slot operations is less than copying the data into a consecutive memory block.

To conclude the striding performance tests, the average speedup is 8.1% regardless that the transformation works against the caching memory system.

**Permuting.** The results of the permuting transformation are shown in figure 11. The results are better than expected as a random permutation is used which is counterproductive for the cache memory system. The big range of *sse_mvm* is again due to different tests combined into the plot. A reason for the huge speedup on *sse_mvm* could not be found.

On average the permuting transformation is 35.2% faster than the reference (not including *sse_mvm* due to the exceptional speedup comared to all other tests).

**Blockstriding.** Figure 12 shows the results of the blockstriding transformation. As expected, the resulting code can be quite slow compared to the reference. This is due to the additional index calculation for every access. More surprising is, that several test cases still can get a significant speedup.

The results depend highly on the input function, the used compiler and the optimization flags. Without the `\fast` flag, the results look a lot worse, similar to the results of the Microsoft compiler

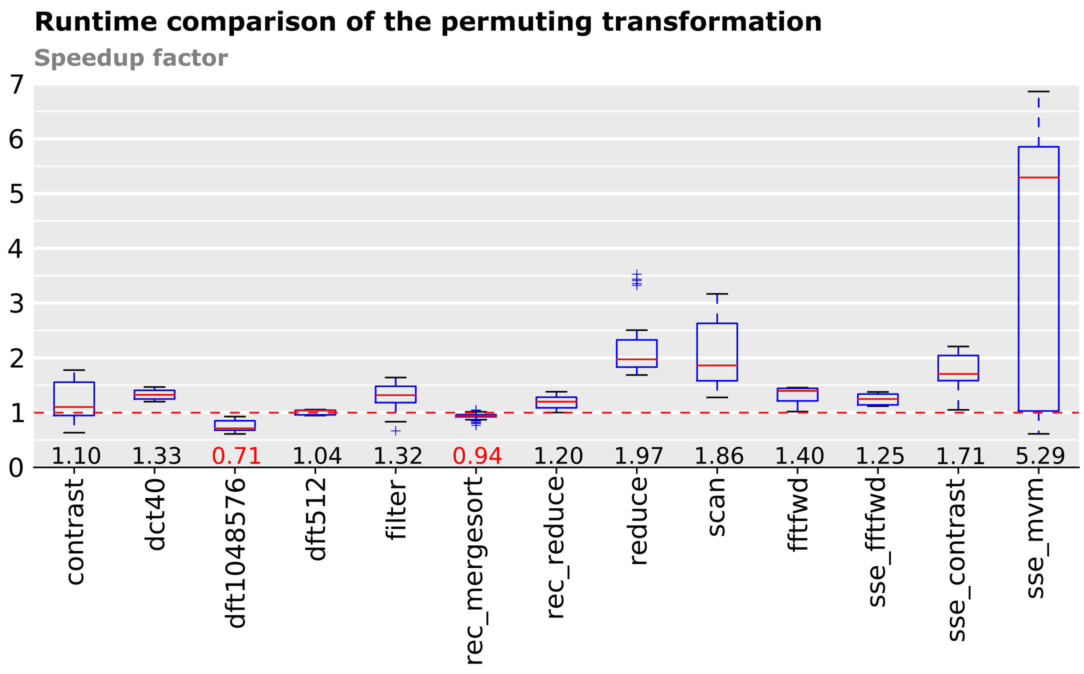

Figure 11: Speedup of permuting

in appendix B.

The more stable results of the 2D optimized blockstriding transformation are shown in figure 13. The blockstriding transformation optimized for 2D indexing performs a lot better than the generic blockstriding transformation, as it does not add any computation at all, it only modifies the present computations. This implies the indexing style, described in the chapter 2.1.4, is used in the function. However, if the transformation can be applied, the resulting function runs a lot faster than the reference due to less memory bandwidth needed.

On average, the blockstriding tests run 30% faster and 125.1% faster with the 2D optimized blockstriding transformation.

### 6.4.2 CPU bound vs. memory bound

All test cases shown until now are mostly memory bound functions. As the transformations all save memory bandwidth with merging multiple steps into one, the performance gain can be huge with memory bound functions. As soon as the CPU is the limiting factor, the impact on the runtime starts to get smaller quickly. Figure 14 shows the effect of getting CPU bound. The test functions *contrast* and *contrast_sse* are modified in order to increase the floating point operations on every element. The operation count gets increased from 4 to 80 per element. No additional vector accesses are added to just increase the load on the CPU. Both CPU bound tests perform clearly worse using the scaling transformation.

All transformations only have an influence on the part of the function which reads or writes the vectors. If the fraction of the runtime spent on vector accesses gets smaller, the overall runtime of the transformed function will be closer to the reference as the transformation has less influence

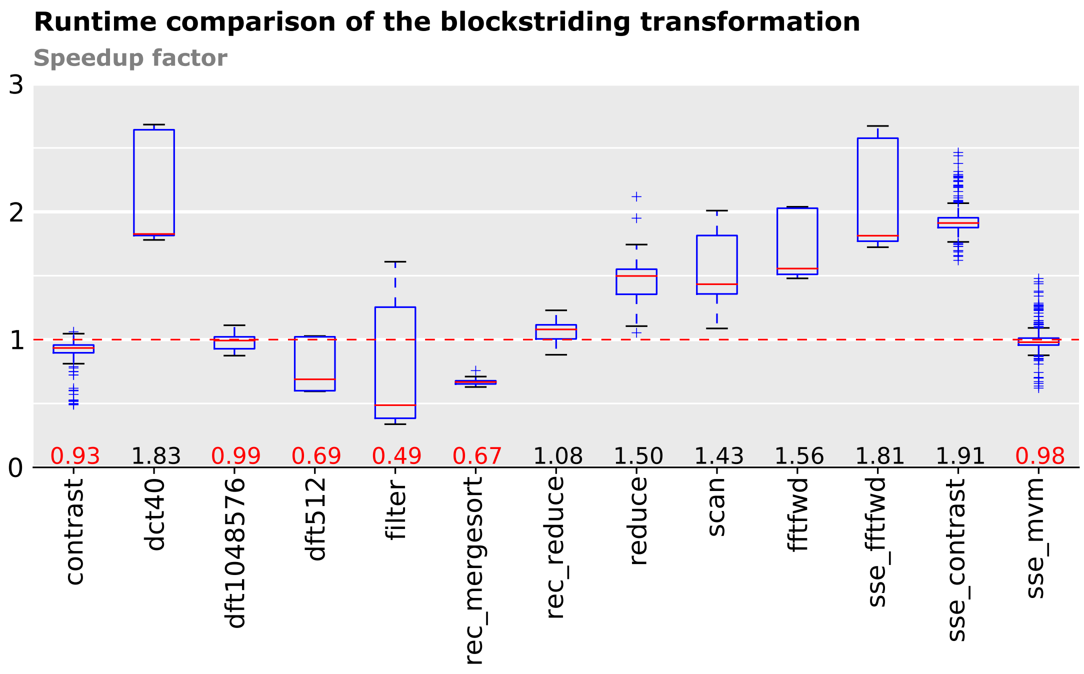

Figure 12: Speedup of blockstriding

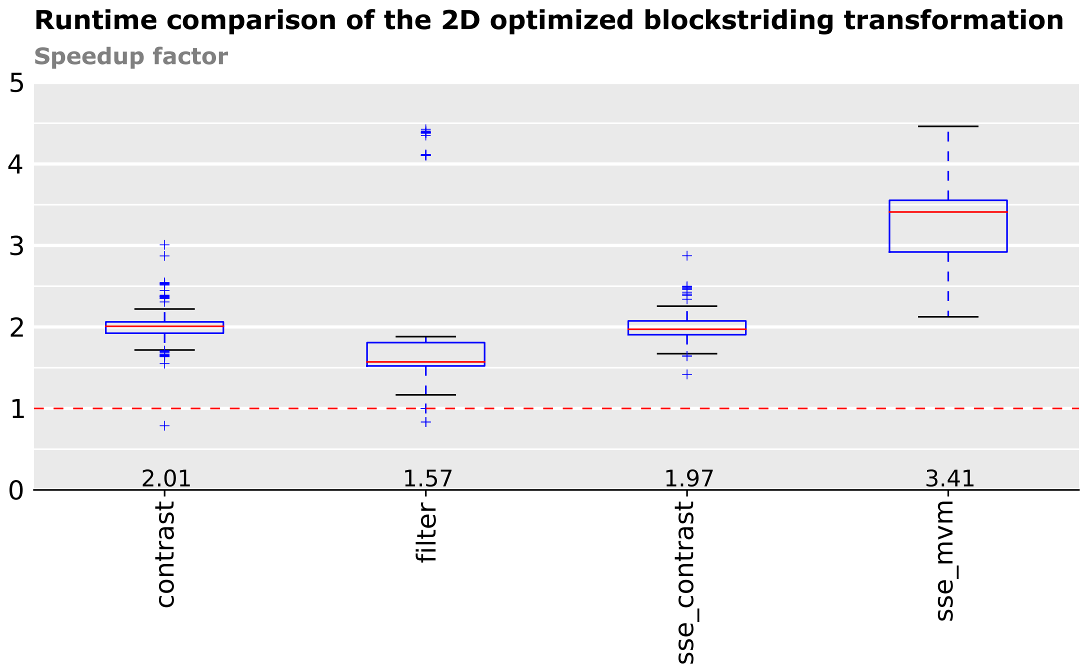

Figure 13: Speedup of 2D optimized blockstriding

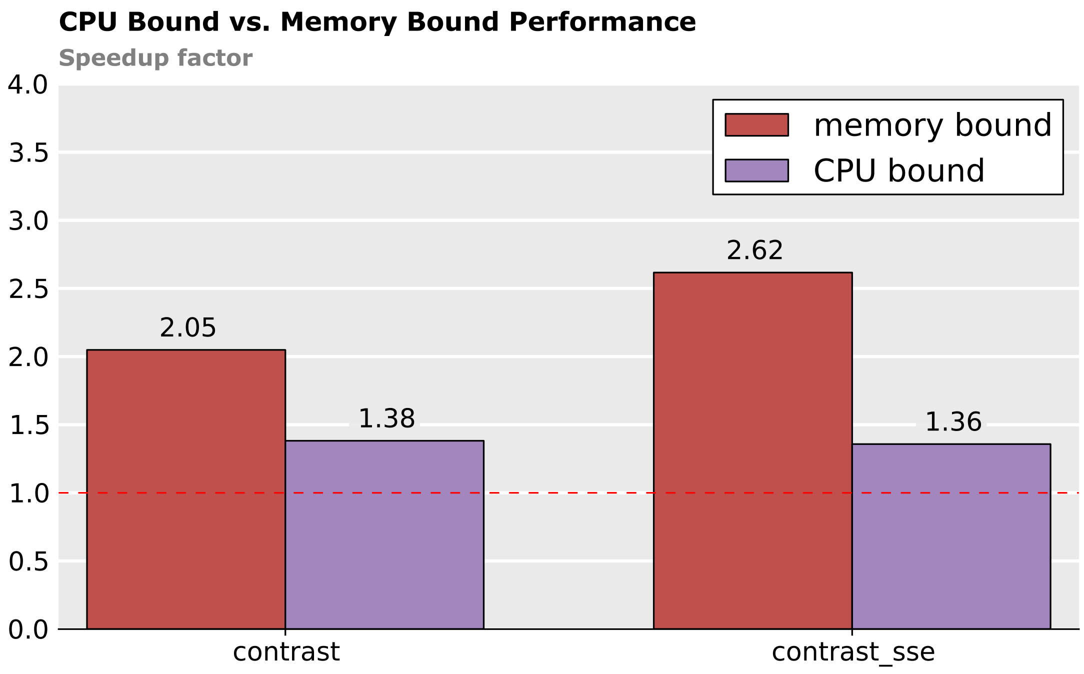

Figure 14: Effects on the runtime of CPU bound functions

on the overall runtime.

### 6.4.3 Impact of input sizes

In the previous performance plots, the results of different input sizes are combined into the final result. In contrast to the previous plots, the performance test in this chapter show the performance gain with different input sizes.

The functions *sse_mvm* and *contrast* are tested with various input sizes $n$. The vector in *sse_mvm* is $n \times 1$ and the matrix $n \times n$. The input size of *contrast* is $n \times n$.

**sse_mvm.** Figure 15 shows the results of the function *sse_mvm*. The speedup on input sizes smaller than 1500 varies more as the cache has a bigger effect on the runtime. The speedup gets steady around 1000 to 1500 input size, where the input byte size increases from around 4MiB to 8.5MiB. As the third level cache of the CPU is 6MiB, the reduction of the needed memory bandwidth has a bigger effect on the runtime if the size exceeds the caches.

There is a significant drop around an input size of 4100. At closer inspection, there are significant slowdowns at input sizes 4072, 4096 and 4120 and a smaller slowdown at input sizes $\pm 4$ of the 3 sizes mentioned before. Measuring the performance counters with AMD CodeAnalyst [1] showed, that during the slow test cases the data cache miss rate is 4 times higher, from around 5% cache misses to around 20%. As the slowdown is only temporary at specific input sizes, it can not be due to compulsory or capacity misses but it can be due to conflict misses.

**contrast.** Compared to *sse_mvm*, the *contrast* function uses double precision and therefore the input size is twice as big and the cache affects the runtime less than for the test case *sse_mvm*. The speedup is steady from an input size of 1000 and higher.

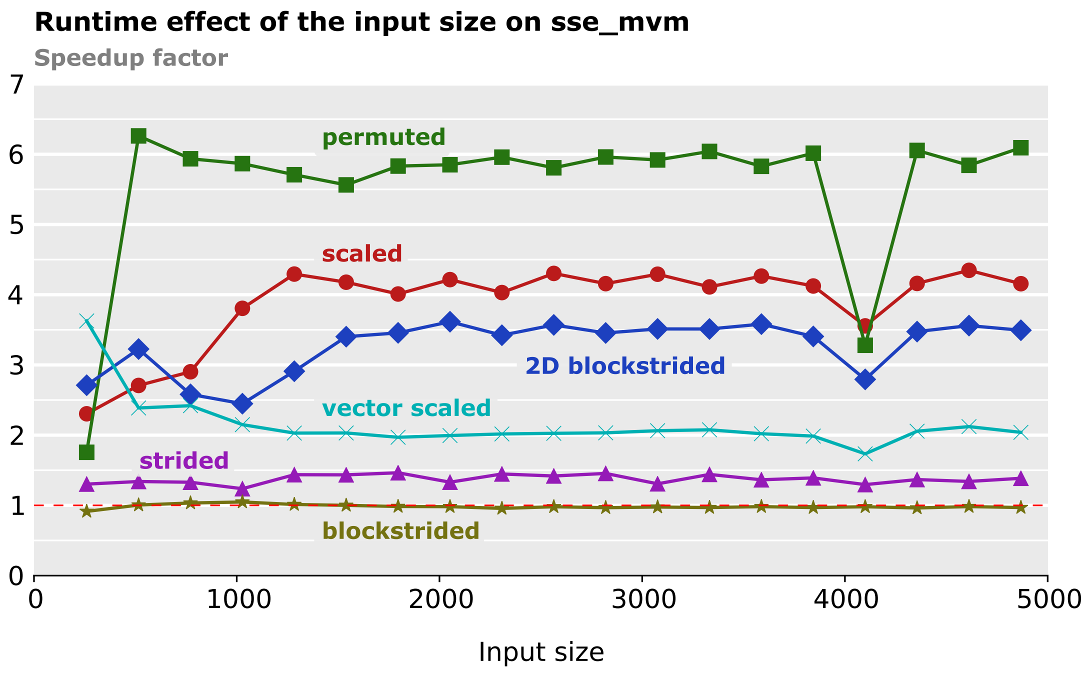

Figure 15: Effects of input size for *sse_mvm*

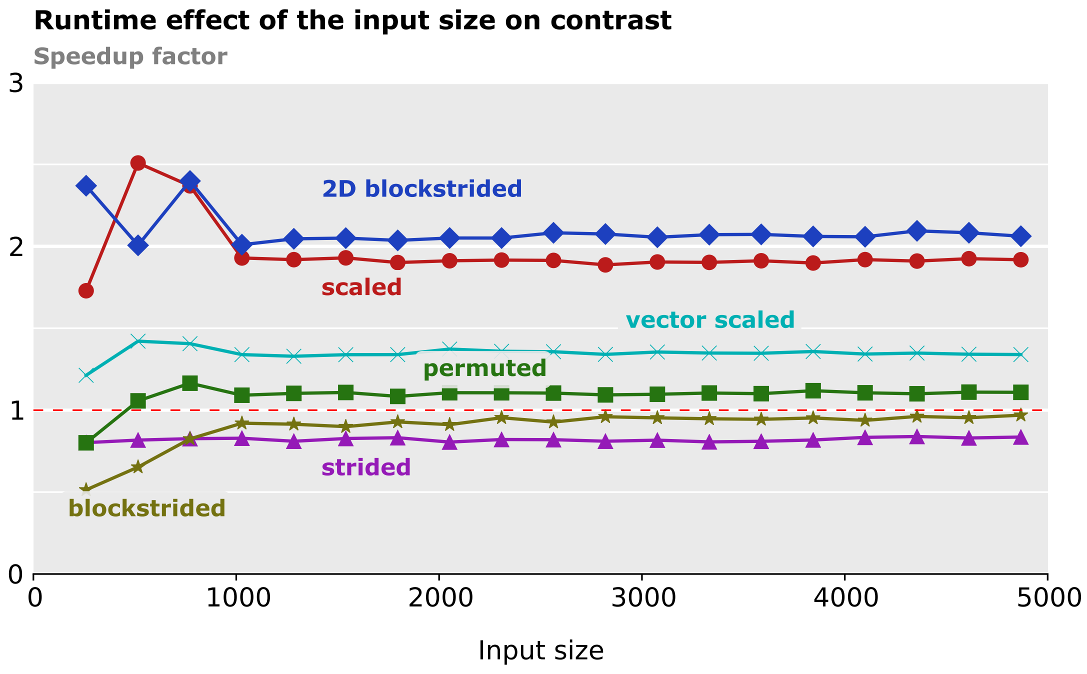

Figure 16: Effects of input size for *contrast*

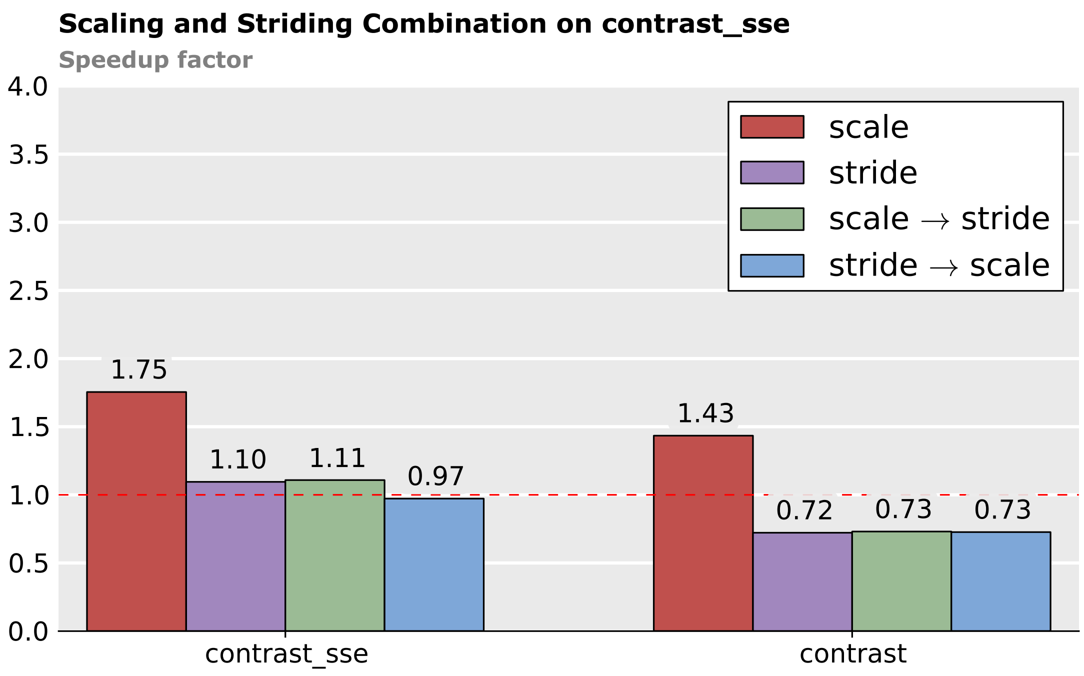

Figure 17: The order of scaling and striding affects the possible performance gain.

To conclude, the speedup is steady if the dataset is too large for the cache because the system is running at its memory bandwidth limit and the size of the input does not have an influence on the speedup.

### 6.4.4 Combination Tests

The effect on the runtime of a different order of a combination of transformations is shown in figure 17. The striding and scaling transformation are first applied alone and the speedup is measured. *contrast_sse* profits from the transformation alone, *contrast* only profits from the scaling transformation. As *contrast_sse* is a vectorized test case, the striding transformation splits the packed vectorized accesses into single slot accesses and increases the number of accesses. The scalar function `contrast` has the same access count before and after the transformation.

If the scaling transformation is applied before the striding transformation, the packed accesses in *contrast_sse* are scaled and then the loads are split up into single slot loads. Therefore, every packed access is replaced with two single slot loads and one packed scaling operation. If the order of the transformations is reversed, the packed accesses is replaced with 2 single slot loads and then 2 single slot scaling operations as the scaling transformation does not know anything about the packed accesses and scales every single slot access. The second version runs around 12% slower than the more efficient combination, which is enough to be slower than the reference. Without any vectorized code in *contrast*, the order does not have an influence on the runtime and the combinations run as slow as the slower of the two transformation alone.

# 7 Applications

In this chapter, applications of the developed tool, the transformed functions and the resulting performance change are shown.

## 7.1 Scaled Strided Blockstrided Contrast Adjustment

The function *contrast* first identifies the maximal and minimal value from the input buffer and then adjusts the input value accordingly to increase the difference between the brightest and the darkest pixel. The whole function is shown in listing 15. It is a simple and short function in order to clearly show the changes made by the transformations. There are very limited interface options, basically providing parameters for the data, the size of the given data and the maximal value after the application of the function.

```
void contrast(const double* const input, double * output, size_t width, size_t
    height,
        const double new_max)
{
    double max = 0;
    double min = 0;
    // get minimal and maximal value of input
    for(int y = 0; y < height; y++) {
        for(int x = 0; x < width; x++) {
            double val = input[x+y*width];

            if(max < val) {
                max = val;
            }
            if(min > val) {
                min = val;
            }
        }
    }
    // increase contrast
    for(int y = 0; y < height; y++) {
        for(int x = 0; x < width; x++) {
            double tmp = (input[x+y*width]-min)/(max - min) * new_max;
            output[x+y*width] = tmp;
        }
    }
}
```

Listing 15: source code of contrast before transformations

As an example, three transformations are used on the function *contrast*, namely scaling, striding and 2D optimized transformation. With these transformations, it is possible to use the function on one channel on a part of a RGB image. For this function, the scaling transformation is superfluous as the function itself does a scaling, however in this example it is used to show the effects of the transformation. The three transformation add the following features to the function:

- **Scaling** is used to bring the values from a range from 0 to 255 in to the range from 0.0 to 1.0 and at the end back to the original range.
- **Striding** is used to extract one color channel of the three channels. A striding factor of 3 is used to read one color channel and the starting color value determines which channel is used.
- **2D optimized Blockstriding** is used to apply the function only on a part of the image. The 2D optimized blockstriding transformation can be used because *contrast* uses the index style `x + y * width`

The result of the 3 transformations can be seen in listing 16. The changes are minor as there are only 2 read and 1 write accesses. The index of the two read accesses on line 6 and 17 is multiplied with the striding factor of the striding transformation and `width` is replaced with `buffer_width` by the 2D optimized blockstriding transformation. Furthermore, the read value is scaled by `prescale_input_0` from the scaling transformation. The written value on line 18 is scaled by `postscale_output_0`.

The **runtime** is measured for three different sizes of an RGB image. For a small size of 512 by 512 pixels, the transformed function is around 10% faster, for larger sizes (2048 by 2048 and 4096 by 4096) the speedup increases to 30% due to less cache effects for the reference function.

```
 1 void contrast_prime(const double* const  input, double* output, size_t width,
       size_t height, const double new_max, const int buffer_width_input_0, const
       double prescale_input_0, const double postscale_output_0, const int
       stride_input_0) {
 2     double max = 0;
 3     double min = 0;
 4     for(int y = 0; y < height; y++) {
 5         for(int x = 0; x < width; x++) {
 6             double val = (prescale_input_0 * (input[stride_input_0 * (x + y *
                   buffer_width_input_0)]));
 7             if(max < val) {
 8                 max = val;
 9             }
10             if(min > val) {
11                 min = val;
12             }
13         }
14     }
15     for(int y = 0; y < height; y++) {
16         for(int x = 0; x < width; x++) {
17             double tmp = ((prescale_input_0 * (input[stride_input_0 * (x + y *
                   buffer_width_input_0)])) - min) / (max - min) * new_max;
18             output[x + y * width] = (postscale_output_0 * (tmp));
19         }
20     }
21 }
```

Listing 16: source code of contrast with applied 2D blockstriding, scaling and striding transformations

Given an RGB image with size 2048 by 2048 and the need to process a 1024 by 1024 rectangle from the image, the scaling and blockstriding of the data for the function *contrast* is shown in listing 17. The part of the image has to be copied together and scaled before the function and scaled after the function call in contrast to the transformed function, which can be called directly with `contrast_prime(image + (x + y * width)* 3, output, width, height, new_max, buffer_width, 1/255.0, 255, 3);`.

```c
int buffer_width = 2048; //width of image
int width = 1024; // width to process
int height = 1024; // height to process

// start coordinates of the rectangle to process
int x = 10;
int y = 10;

for(int i = 0; i < height; i++) {
    for(int j = 0; j < width; j++) {
        input[i * width + j] = image[((y + i) * buffer_width + j + x) * 3] *
            (1/255.0);
    }
}

contrast(input, output, width, height, new_max);

for(int i = 0; i < height; i++) {
    for(int j = 0; j < width; j++) {
        output[i * width + j] = output[i * width + j] * 255;
    }
}
```

Listing 17: manual scaling and blockstriding for the function *contrast*

## 7.2 Vectorized Matrix-Vector Multiplication

The influence of vectorized code on the transformation is shown with the vector-matrix multiplication already used during the experiments. Listing 18 contains the source code of the function. It has 2 input pointers: the vector `x` and the matrix `M`. The pointer `y` is used to store the result. For simplicity reasons, the function only supports square matrices.

```c
void f(const float * x, const float * M, float * y, const int size)
{
    __m128 m1,m2,m3,m4;
    __m128 x1, y1;
    for (int i = 0; i < size; i=i+4)
    {
        y1 = _mm_set1_ps(0);
        for (int j = 0; j < size; j = j+4)
        {
            int index = i*size+j;
            x1 = _mm_load_ps(x+j);
            m1 = _mm_load_ps(M+index);
            m2 = _mm_load_ps(M+index+size);
            m3 = _mm_load_ps(M+index+2*size);
            m4 = _mm_load_ps(M+index+3*size);

            m1 = _mm_mul_ps(m1, x1);
            m2 = _mm_mul_ps(m2, x1);
            m3 = _mm_mul_ps(m3, x1);
            m4 = _mm_mul_ps(m4, x1);

            m1 = _mm_hadd_ps(m1, m2);
            m3 = _mm_hadd_ps(m3, m4);
            m1 = _mm_hadd_ps(m1, m3);
            y1 = _mm_add_ps(y1, m1);
        }
        _mm_store_ps(y+i, y1);
    }
}
```

Listing 18: source code of sse_mvm

To show the permutation and vector scaling transformation it is assumed that the vector `x` is reversed and the elements need to be scaled by the reciprocal value of the index of the element ($\frac{1}{i}$). At first, the vector scaling transformation is used as the scaling vector is not reversed and afterwards the permuting transformation to reverse the vector `x`.

The output after the two transformations is shown in listing 19. On line 9 are the next scaling factors loaded and then the scaling multiplication is on line 19. The permuted values are loaded on the lines 11 to 17.

```
1  void f_prime(const float* x, const float* M, float* y, const int size, const
       float* prescale_vec_x_0, const int* perm_x_0) {
2      __m128 a_0,a_1,a_2,a_4;
3      __m128 prescale_vec_x_0_sse;
4      __m128 m1,m2,m3,m4, x1, y1;
5      for(int i = 0; i < size; i = i + 4) {
6          y1 = _mm_set_ps1(0);
7          for(int j = 0; j < size; j = j + 4) {
8              int index = i * size + j;
9              prescale_vec_x_0_sse = _mm_load_ps(prescale_vec_x_0 + j);
10             int tmp_0 = j;
11             a_1 = _mm_load_ss(x + perm_x_0[(tmp_0 + 0)]);
12             a_2 = _mm_load_ss(x + perm_x_0[(tmp_0 + 1)]);
13             a_3 = _mm_load_ss(x + perm_x_0[(tmp_0 + 2)]);
14             a_4 = _mm_load_ss(x + perm_x_0[(tmp_0 + 3)]);
15             a_1 = _mm_shuffle_ps(a_1, a_2, 0);
16             a_3 = _mm_shuffle_ps(a_3, a_4, 0);
17             a_1 = _mm_shuffle_ps(a_1, a_3, 136);
18             a_0 = a_1;
19             a_0 = _mm_mul_ps(prescale_vec_x_0_sse, a_0);
20             x1 = a_0;
21             m1 = _mm_load_ps(M + index);
22             m2 = _mm_load_ps(M + index + size);
23             m3 = _mm_load_ps(M + index + 2 * size);
```

```
            m4 = _mm_load_ps(M + index + 3 * size);
            m1 = _mm_mul_ps(m1, x1);
            m2 = _mm_mul_ps(m2, x1);
            m3 = _mm_mul_ps(m3, x1);
            m4 = _mm_mul_ps(m4, x1);
            m1 = _mm_hadd_ps(m1, m2);
            m3 = _mm_hadd_ps(m3, m4);
            m1 = _mm_hadd_ps(m1, m3);
            y1 = _mm_add_ps(y1, m1);
        }
        _mm_store_ps(y + i, y1);
    }
}
```

Listing 19: source code of the permuted and scaled sse_mvm

The runtime is measured for vector sizes from 260 to 16132 elements. Although the packed load intrinsics are split up into single slot intrinsics, the runtime of the transformed function is 1% faster than the reference. The effects on the runtime are rather small because only the accesses to the vector `x` are modified and the size of `x` ($n$) is marginal compared to the matrix `M` ($n^2$).

# 8 Related Work

## 8.1 Locality Optimizations

Locality optimization is a well studied field. The different approaches mostly target generic compiler optimizations. Locality can be improved by changing the data layout of the used data structures during compilation, for example array unification in [6] which combines multiple arrays into one. Another approach is to modify, fuse or split nested loops to be more cache friendly as presented in [13] and many following papers. Also combinations of loop and data transformations are developed and evaluated recently in [4]. These approaches are strictly compiler optimizations and do not add functionality to the handled functions. As opposed to these optimizations, the presented tool augments functions with various transformations to enhance the flexibility of the procedures. The added functionality allows the developer to fuse loops together and therefore improve the locality.

Most of these optimizations target single procedures, which is a common limitation for optimizations in compiler design. Nevertheless, inter-procedural locality optimization have been shown in [7] to be more effective than loop and data optimizations alone. The developed tool allows the developer to inject functionality which could have been in a separate procedure into the target procedure. This is similar to inter-procedural optimizations as it merges loops together, however the one of the loops is from a fixed set of the transformations.

All these locality optimizations are generic compiler optimizations and usually work on an abstract syntax tree of the compiler without the need to output any source code. S. Carr shows in [2] a tool which does a Fortran source-to-source transformation to improve cache performance and instruction level parallelism. The presented source-to-source transformation approach is applied manually by the developer before the compilation and change the functionality and parameters of the source code. The enhanced functions can be stored and used later in the code base. This is not possible in a compiler optimization as the compiler verifies the correctness of the source code before applying optimizations and additional parameters can not be added in this phase of the compilation.

## 8.2 Refactoring

Refactoring usually does not change the external behavior and is targeted to improve the internal structure of source code. Semi-automatic approaches usually recommend refactoring based on different algorithms. For example Shimomura et al. show in [11] an approach with a genetic algorithm-driven refactoring for Java programs which recommends design patterns to improve the code quality. In this works, the behavior of a function is changed and therefore not a refactoring traditionally. It can be seen as a refactoring in terms of that the tool refactors specific functionality into the function. The lack of generic refactoring gives the advantage of being automatic than generic refactoring approaches.

# 9 Future Work

## 9.1 Static Parameters Optimization

The transformations add new parameters in order to allow arbitrary values for the transformations. However, if this flexibility is not needed as the value will be fixed throughout the usage of the function, it is more suitable to compiler optimizations if these static values are inserted as literals in the function. This allows the compiler to optimize more aggressive resulting in faster code.

## 9.2 More Generic Transformations

The transformations analyze and modify either the access to a vector or an index of an access, a more generic transformation, which allows arbitrary code inserted, could allow many other transformations implemented by users. Basically, a scale transformation changes an access $v[i]$ into $v[i] * \text{scale}$. The resulting value is a function of the original value which could be generalized into $f(v[i])$. The same can be applied to the striding transformation in order that the new index is a function of the original index resulting in $v[f(i)]$. Combining these two ideas gives a generic transformation which replaces any read access $v[i]$ with $f_{\text{value}}(\text{origin}(v), f_{\text{index}}(\text{offset}(v[i])))$. The offset and origin properties are needed for transformation which depend on the total offset of an access like permutation or vector scaling. Write accesses and vectorized code can be handled similarly to the conversion of read accesses explained before.

With the new basic framework, the implemented transformations and many other transformations based on reading/writing vectors can be implemented. For example a RGB to gray-scale transformation can be implemented which results in $f_{\text{index}}(i) : i{*}3$ and $f_{\text{value}}(o, i) : \frac{o[i]+o[i+1]+o[i+2]}{3}$. This transformation converts an RGB image on-the-fly to a gray-scale image.

## 9.3 Abstract Interpretation

With the gained insight of the properties needed for the transformations, it might be beneficial to model the offset and origin properties as a static program analysis based on abstract interpretation. Every step of the analysis can be proofed to be sound. This approach can give less strict limitations on the input function. For example if the same pointer is assigned in the if and else block, the tool still can get the offset and the origin of the assigned pointer.

## 9.4 Roofline Model

The results vary quite a lot from a huge speed up to a significant slow down. The functions generated by spiral are highly optimized to yield maximal performance. These functions run closer to the maximal performance of the computer than the non-optimized functions and therefore have less possibility to gain performance from the transformations. The Roofline model [12] shows the performance in relation to the maximal attainable performance. With this additional information, it is expected to get a better understanding if performance will increase or decrease with the addition of the transformations.

# 10 Conclusions

The presented work shows an approach to automatically enhance numerical functions to improve locality by providing several transformations.

The additional effort to create multiple variants of a function is minimal and does not need any interaction during the transformation. Therefore, an optimal variant for many use-cases can be generated with only maintaining the original function. This reduces development time and increases the performance compared using the solely the original function.

The limitations on the input functions are the biggest drawback of the approach, restricting the set of input functions. However, a numerical function might only need minor modifications in order to fulfill the limitations. Additionally, these limitations may be eased with a more advanced analysis of the input function.

Although the transformations are in general beneficial for the runtime, it is advisable to evaluate the runtime of the transformed function as the speedup highly depends on the access pattern of the function. Numerical functions accessing every element once are more likely to benefit from transformations than functions with higher access count such as sorting.

With further work put into a more advanced analysis and additional transformations, the approach provides a real benefit for many applications without being too restrictive on the input function.

## References

[1] *AMD CodeAnalyst Performance Analyzer*. Mar. 2013. URL: http://developer.amd.com/tools/heterogeneous-computing/amd-codeanalyst-performance-analyzer/.

[2] S. Carr. "Combining optimization for cache and instruction-level parallelism". In: *Parallel Architectures and Compilation Techniques, 1996., Proceedings of the 1996 Conference on.* Oct, pp. 238–247. DOI: 10.1109/PACT.1996.552672.

[3] *clang: a C language family frontend for LLVM.* Mar. 2013. URL: http://clang.llvm.org/.

[4] Wei Ding and Mahmut Kandemir. "Improving last level cache locality by integrating loop and data transformations". In: *Proceedings of the International Conference on Computer-Aided Design.* ICCAD '12. San Jose, California: ACM, 2012, pp. 65–72. DOI: 10.1145/2429384.2429398. URL: http://doi.acm.org/10.1145/2429384.2429398.

[5] C. A. R. Hoare. "An axiomatic basis for computer programming". In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. DOI: 10.1145/363235.363259.

[6] Mahmut T. Kandemir. "Array Unification: A Locality Optimization Technique". In: *Proceedings of the 10th International Conference on Compiler Construction.* CC '01. London, UK, UK: Springer-Verlag, 2001, pp. 259–273.

[7] M. Kandemir et al. "A framework for interprocedural locality optimization using both loop and data layout transformations". In: *Parallel Processing, 1999. Proceedings. 1999 International Conference on*, pp. 95–102. DOI: 10.1109/ICPP.1999.797393.

[8] M. Puschel et al. "SPIRAL: Code Generation for DSP Transforms". In: *Proceedings of the IEEE* 93.2 (Feb.), pp. 232–275. DOI: 10.1109/JPROC.2004.840306.

[9] *Python Programming Language.* Mar. 2013. URL: http://python.org/.

[10] J.J. Rodriguez. "An improved bit-reversal algorithm for the fast Fourier transform". In: *Acoustics, Speech, and Signal Processing, 1988. ICASSP-88., 1988 International Conference on.* Apr, 1407–1410 vol.3. DOI: 10.1109/ICASSP.1988.196862.

[11] T. Shimomura, K. Ikeda, and M. Takahashi. "An Approach to GA-Driven Automatic Refactoring Based on Design Patterns". In: *Software Engineering Advances (ICSEA), 2010 Fifth International Conference on.* 2010, pp. 213–218. DOI: 10.1109/ICSEA.2010.39.

[12] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures". In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. DOI: 10.1145/1498765.1498785. URL: http://doi.acm.org/10.1145/1498765.1498785.

[13] Michael E. Wolf and Monica S. Lam. "A data locality optimizing algorithm". In: *SIGPLAN Not.* 26.6 (May 1991), pp. 30–44. DOI: 10.1145/113446.113449.

# List of Figures

# List of Tables

# Listings

# A Manual

## A.1 Prerequisites

- Install Python 2.7 [9]
- Copy libclang's Python bindings (from the clang source code in `bindings\python\clang`) in a Python include path in order that `import clang.cindex` works
- Copy the libclang shared library (libclang.so on Unix and libclang.dll on Windows) in a LD_LIBRARY_PATH (Unix) or PATH (Windows). On Unix `sudo make install` in the clang's source directory should be sufficient.

During the development, libclang was built from the svn repository using revision 172934. Building *clang* from source is described on `http://clang.llvm.org/get_started.html#build`.

## A.2 Installation

After installing the prerequisites are installed, the tool does not need to be installed.

## A.3 Usage

The tool can be run with `python main.py` in the tool's directory. The command `python main.py -h` prints the following help text:

```
usage: main.py [-h]
                [-t --transformation {prescale_vector,stride,prescale,postscale,
                 blockstride,permute,postscale_vector,
                 2Dblockstride}]
                [-f --function FUNCTION_NAME]
                [-p --function_prime TRANSFORMED_FUNCTION_NAME]
                [-o --options OPTIONS] [-y] [-I INCLUDE_PATH]
                infile outfile

Automatic Refactoring: Locality Friendly Interface Enhancements for Numerical Functions

positional arguments:
  infile
  outfile

optional arguments:
  -h, --help            show this help message and exit
  -t --transformation {2Dblockstride,blockstride,permute,postscale,
    postscale_vector,prescale,prescale_vector,stride}
                        selects the transformation to be used (default: stride)
  -f --function FUNCTION_NAME
                        function name to transform (default: f)
```

```
-p --function_prime TRANSFORMED_FUNCTION_NAME
                      transformed function name (default: f_prime)
-o --options OPTIONS  transform options
-y                    forces overwriting outfile (default: false)
-I INCLUDE_PATH       provide additional include directories to parse
                      infile, multiple paths with multiple -I parameters
```

Most arguments are straight forward to use except the options argument. It is a string which defines which parameters are processed by the transformation. It is a list of parameter groups. A parameter group is a comma separated list of parameters which use the same additional parameter of the transformation. All parameters listed in the same group share the same additional parameter added by the transformation. Parameter groups are separated by a semi-colon. For example the function has three parameters `x`, `y` and `z`. The parameters `x` and `y` have a different striding than `z`. Therefore, `x` and `y` are in the same group and `z` is in a separate group. This would result in an options string `"x,y;z"` and the striding transformation adds 2 striding parameters, the first for `x` and `y` and the second striding factor is for `z`. The blockstriding transformation is the only transformation which allows only one parameter group, the other transformations allow multiple groups.

## A.4 Examples

The examples are shown for a function with the function header `void f(const double* const input, double * output, size_t width, size_t height, const double new_max)`. For each example, the command and the resulting function header is given. For combinations, multiple commands are given.

1. scalar prescaling `input`:

   ```
   python main.py -t prescale -f f -p f_prime -o "input" in_file.c out_file.c
   ```

   results in

   ```
   void f_prime(const double* const input, double* output, size_t width, size_t height, @\\@const
   double new_max, const double prescale_input_0)
   ```

2. striding `input` and `output` with the same striding factor:

   ```
   python main.py -t stride -f f -p f_prime -o "input,output" in_file.c out_file.c
   ```

   results in

   ```
   void f_prime(const double* const input, double* output, size_t width, size_t height, const double
    new_max, const int stride_input_output_0)
   ```

3. striding `input` and `output` with different striding factors:

   ```
   python main.py -t stride -f f -p f_prime -o "input;output" in_file.c out_file.c
   ```

   results in

   ```
   void f_prime(const double* const input, double* output, size_t width, size_t height, const double
    new_max, const int stride_input_0, const int stride_output_0)
   ```

4. permute and prescale `input`, vector postscale `output`:

   ```
   python main.py -t permute -f f -p f1 -o "input" in_file.c out_file1.c
   python main.py -t prescale -f f1 -p f2 -o "input" out_file1.c out_file2.c
   ```

```
python main.py -t postscale_vector -f f2 -p f_prime -o "output" out_file2.c out_file3.c
```

results in

```
void f_prime(const double* const input, double* output, size_t width, size_t height, const double
 new_max, const int* perm_input_0, const double prescale_input_0, const double* postscale_vec_output_0
)
```

# B Microsoft Compiler

The performance tests were additionally compiled with the Microsoft C/C++ Compiler Version 17.00.50727.1 for x64. The used compiler flags are /Ox, /arch:SSE2, /GA, /fp:fast, /Oa, /Ob2, /Oi and /Ot. For reference, the performance plot are give in this chapter. The biggest difference shows the blockstriding transformation, clearly running worse than with the Intel compiler.

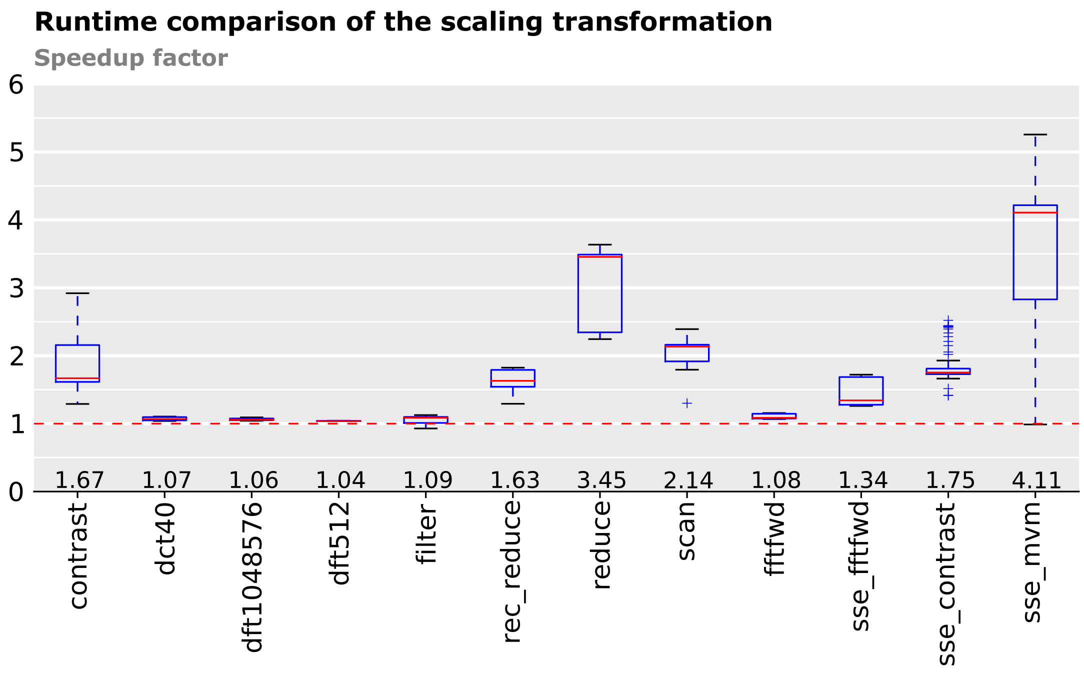

Figure 18: Speedup of scaling transformation

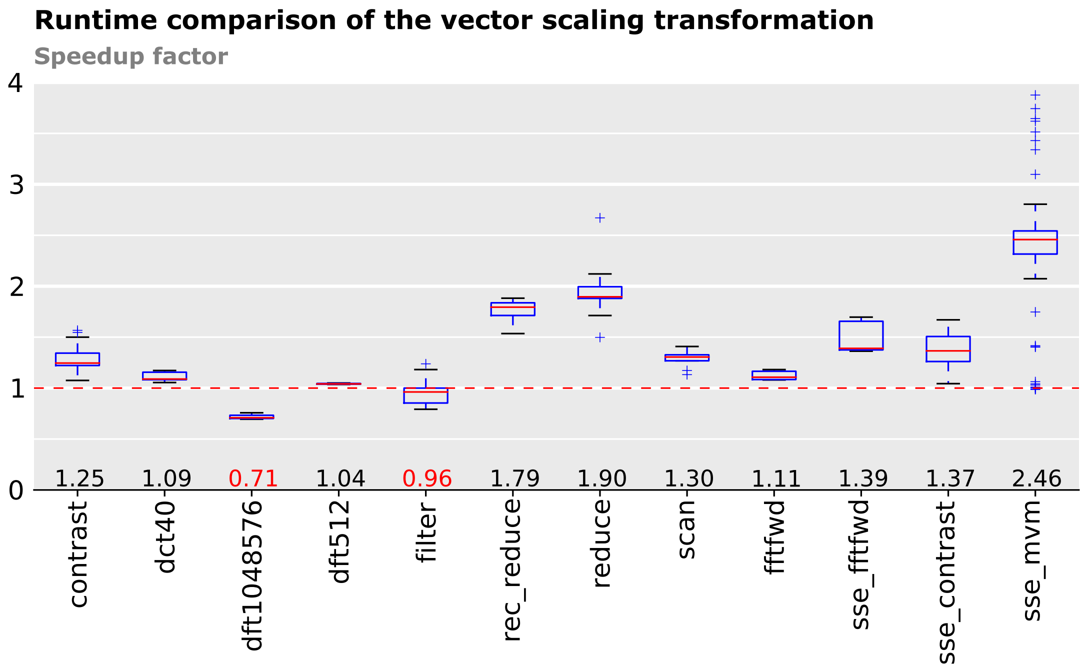

Figure 19: Speedup of vector scaling transformation

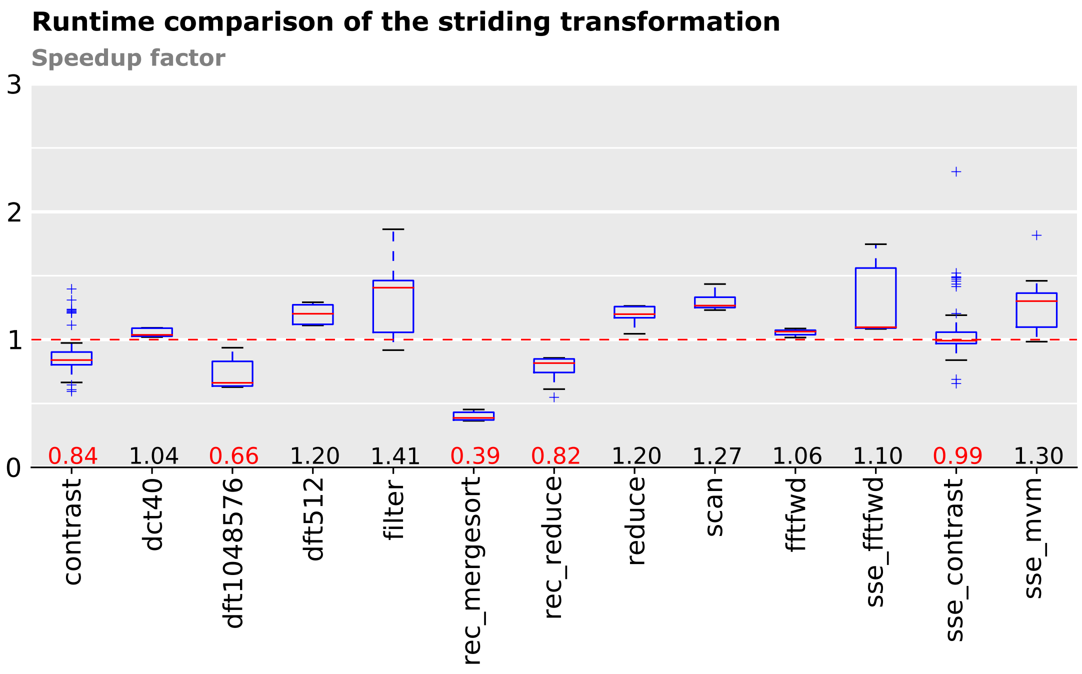

Figure 20: Speedup of striding

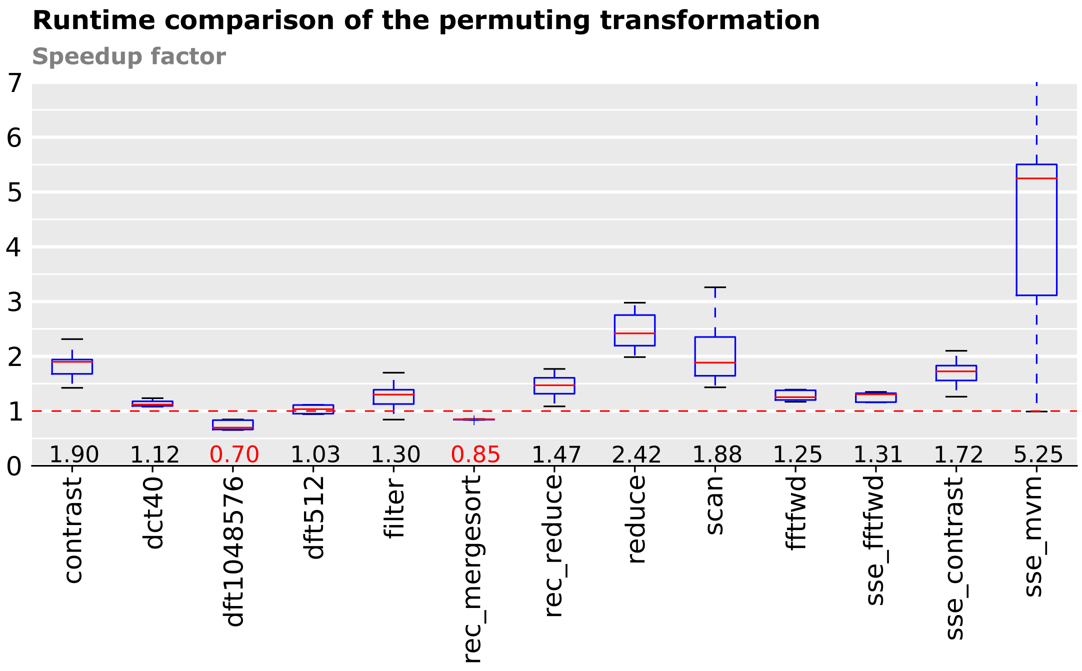

Figure 21: Speedup of permuting

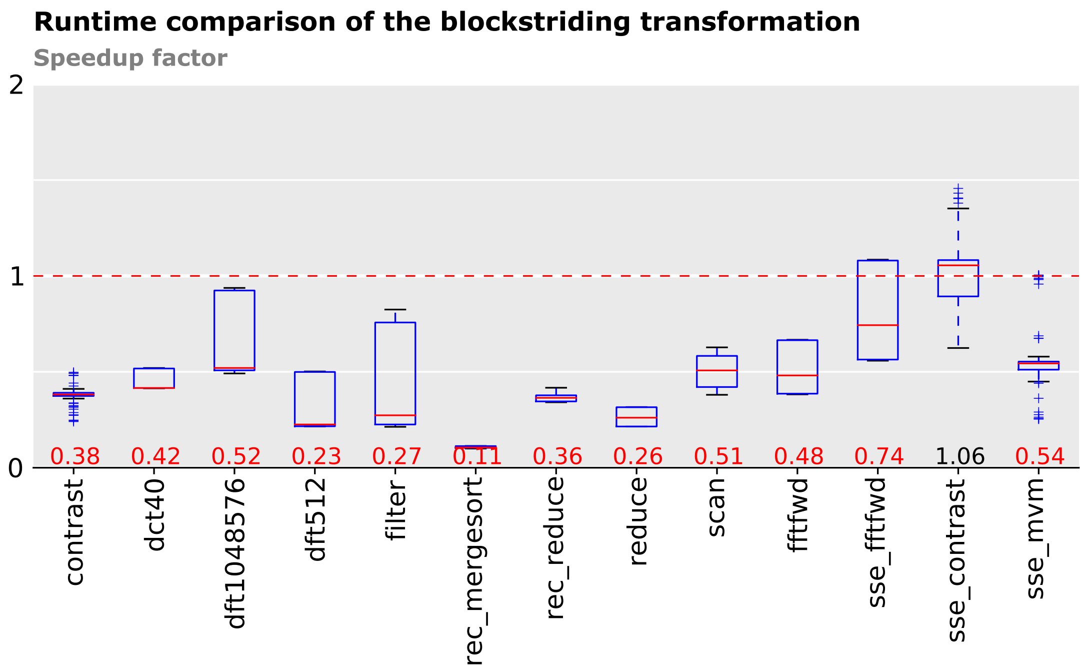

Figure 22: Speedup of blockstriding

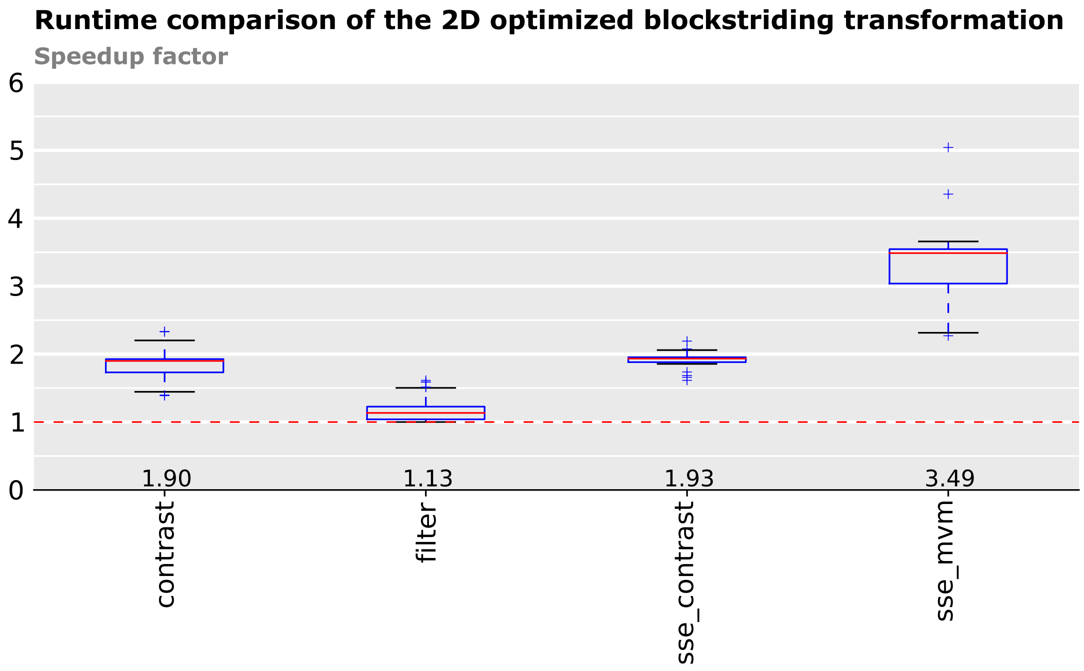

Figure 23: Speedup of 2D optimized blockstriding

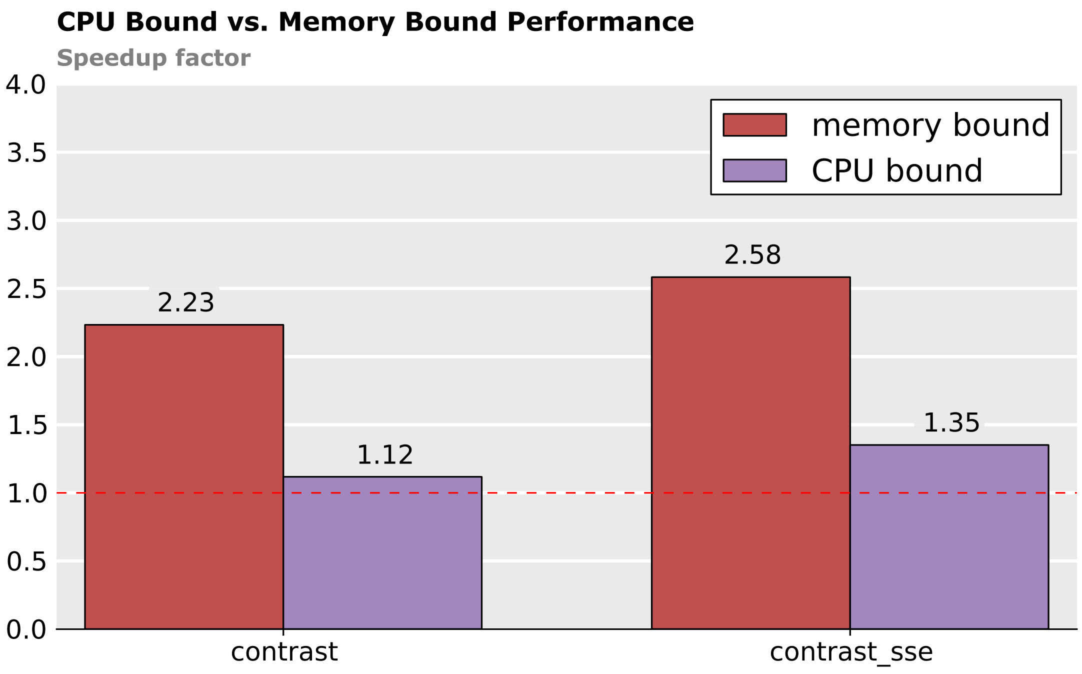

Figure 24: Effects on the runtime of CPU bound functions

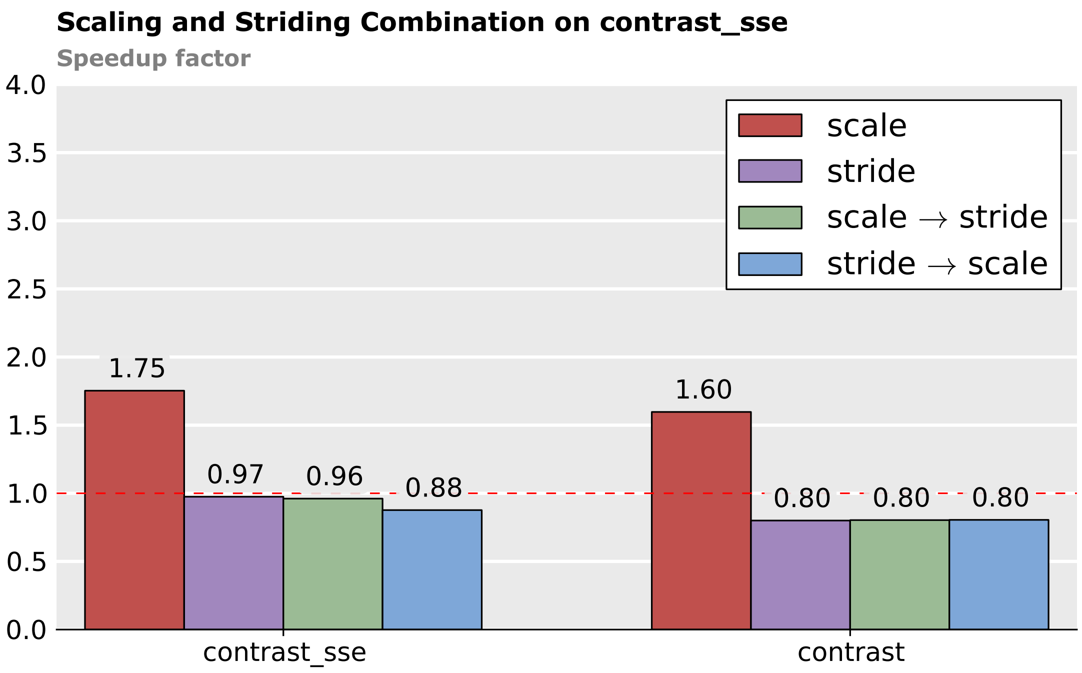

Figure 25: The order of scaling and striding affects the possible performance gain.

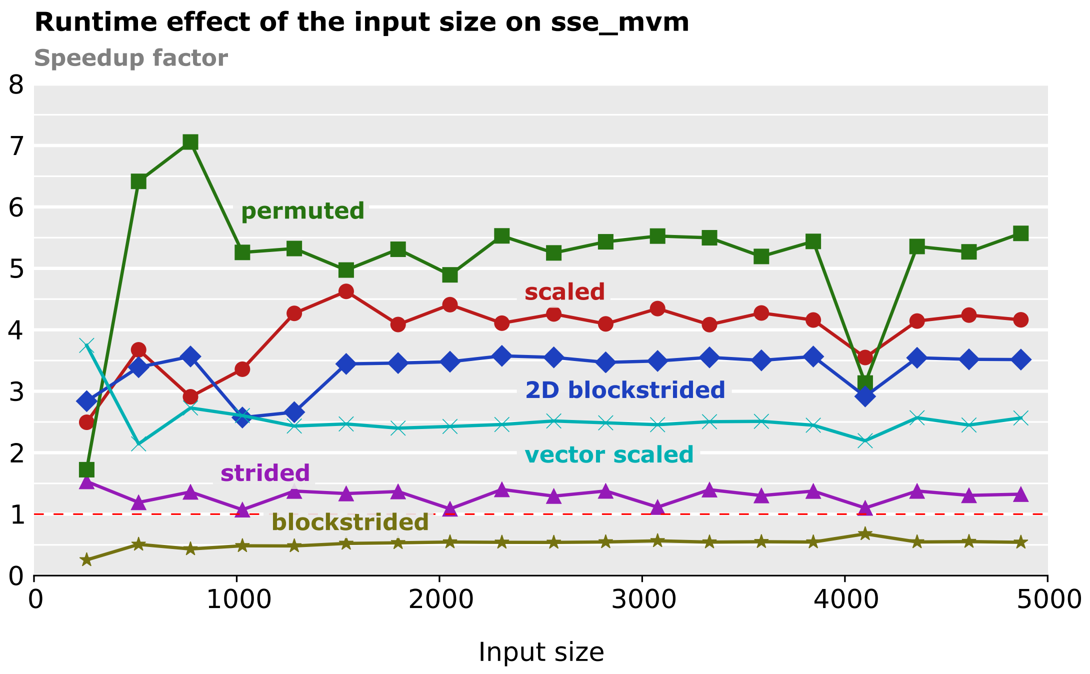

Figure 26: Effects of input size for *sse_mvm*

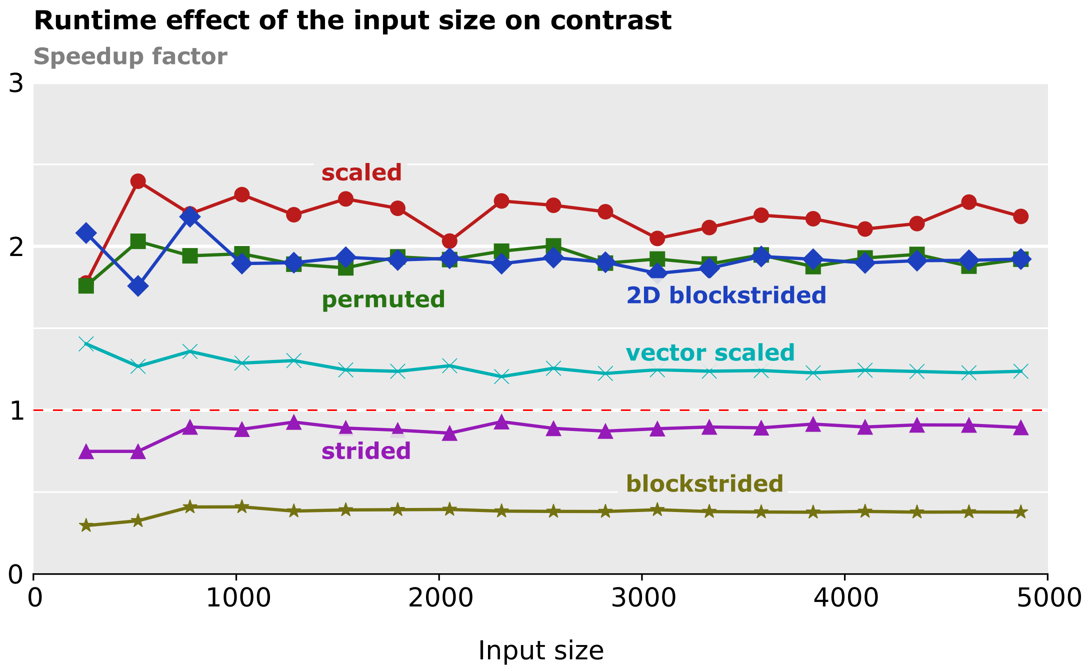

Figure 27: Effects of input size for *contrast*