Loop Transformations Leveraging Hardware Prefetching

Savvas Sioutas
Eindhoven University of Technology
s.sioutas@tue.nl

Sander Stuijk
Eindhoven University of Technology
s.stuijk@tue.nl

Henk Corporaal
Eindhoven University of Technology
h.corporaal@tue.nl

Twan Basten
Eindhoven University of Technology
TNO ESI
a.a.basten@tue.nl

Lou Somers
Océ Technologies
Eindhoven University of Technology
l.j.a.m.somers@tue.nl

Abstract

Memory-bound applications heavily depend on the bandwidth of the system in order to achieve high performance. Improving temporal and/or spatial locality through loop transformations is a common way of mitigating this dependency. However, choosing the right combination of optimizations that should be applied on the target loop nest is not a trivial task, due to the fact that most of these transformations alter the memory access pattern of the application and as a result interfere with the efficiency of the sophisticated hardware prefetching mechanisms present in most modern architectures. We propose an optimization algorithm that analytically classifies an algorithmic description of a loop nest in order to decide whether it should be optimized stressing its temporal or spatial locality. We detect specific patterns in its definition while also taking the hardware prefetching units of modern processors into account. We implement our proposed technique as a tool to be used with the Halide compiler and test it on a variety of benchmarks. We find an average performance improvement of over 40% compared to previous analytical models targeting the Halide language and compiler.

CCS Concepts • **Software and its engineering → Compilers**; *Domain specific languages*;

Keywords loop optimizations, compiler optimizations, Halide

ACM Reference Format:
Savvas Sioutas, Sander Stuijk, Henk Corporaal, Twan Basten, and Lou Somers. 2018. Loop Transformations Leveraging Hardware Prefetching. In *Proceedings of 2018 IEEE/ACM International Symposium on Code Generation and Optimization, Vienna, Austria, February 24–28, 2018 (CGO'18),* 11 pages.
https://doi.org/10.1145/3168823

CGO'18, February 24–28, 2018, Vienna, Austria
2018. ACM ISBN 978-1-4503-5617-6/18/02…$15.00
https://doi.org/10.1145/3168823

1 Introduction

The ever-growing gap between processor and memory speed in modern architectures is currently a severe drawback in the efficiency of applications in domains where high performance is necessary. Memory-intensive (or memory-bound) applications are affected the most by this problem, since they usually contain loop nests with a large number of memory accesses and relatively few computations. As a result, they are bound by the memory bandwidth of the system [29].

Developers often employ various optimization methods and techniques in order to mitigate the effects of this memory bandwidth problem and increase the performance of their implementations. Loop tiling [1, 4–6, 11, 13, 22, 23, 26, 27] is a common loop transformation that aims to improve temporal locality thereby enabling data reuse. Tiling paired with vectorization and parallelization can have a huge impact on the performance of an application. However, picking the proper tile dimensions that will minimize external memory accesses without interfering with the SIMD unit or the hardware prefetching mechanism present in most modern architectures is not a trivial task. Due to these reasons, manually optimizing a target algorithm is a time-consuming and error-prone process, where numerous architecture and application-specific parameters need to be considered.

In the past, most approaches to automatic tile size selection have mainly focused on analytical models [3, 12, 15, 18] that only consider loop nests that fit into specific patterns while relying on the compiler to decide on the optimal loop ordering. These methods may quickly generate efficient code when the loop nest fits into the expected pattern but may produce sub-optimal results in other case. Furthermore, they usually ignore the hardware prefetching mechanisms found in modern architectures and as a result, the proposed optimizations may actually lead to a deterioration in the performance of the final implementation.

Other approaches employ dynamic auto-tuning frameworks [2, 25] that exhaustively search the optimization space in order to optimize the target application. In general, these frameworks are able to produce near-optimal results. However the time needed to converge to that solution is usually

unknown, thus making them inadequate for fast design space exploration and debugging. Furthermore, the fact that they need to run on the target platform can also be a limitation for some architectures.

In this work, we propose an optimization algorithm and analytical model for memory-bound applications that aims to minimize external memory accesses, while taking necessary architecture and application-specific parameters into account. The model first classifies the application by detecting patterns in the definitions which are derived by the statements in the innermost level of the loop nest. We use these patterns to determine whether the applications should be optimized with emphasis on spatial or temporal locality in order to better exploit the hardware prefetching mechanisms, as well as to determine which other optimizations (i.e. vectorization, non-temporal instructions, multi-threading) may improve the performance of the final implementation. The algorithm then invokes an analytical model that based on the previous classification decides which levels of the cache hierarchy to optimize for and then chooses the tile dimensions as well as the final loop nest order.

We implement our algorithm as a tool to be used along with the Halide language and compiler [20] in order to automatically generate optimization schedules for Halide functions often within milliseconds. We extend the Halide compiler with the ability to generate non-temporal stores by adding a new scheduling directive to the language's front end. We test our method on various target applications and compare its results with previous analytical as well as dynamic empirical (auto-tuning) models. We find that our method achieves an average performance improvement of 40% compared to the aforementioned analytical models targeting the Halide DSL. Performance is also better in terms of quality to the exhaustively auto-tuned implementations, where the results using our approach are usually achieved in a matter of milliseconds instead of hours (in terms of optimization run-time) when using the Halide autotuner.

The remainder of the paper is organized as follows: Section 2 discusses related work. Section 3 presents the proposed model and analysis technique. Section 4 shows the implemented Halide tool, while Section 5 demonstrates experimental results and a comparison with similar frameworks. Concluding remarks are discussed in Section 6.

2 Related Work

The problem of optimizing memory-intensive applications has been considered many times in the past. Most of that work has focused on tile size selection algorithms. These algorithms usually employ analytical models that aim to determine the optimal tile dimensions in order to exploit temporal locality. The authors in [6] take cache parameters into account when generating tile sizes, but are only considering one level of cache hierarchy and no interaction with other optimizations or cache associativity. In [17] the authors propose the block data layout as a solution to the problem and also provide a corresponding analysis. However specific hardware and software support is needed in order take advantage of their approach, which limits the application scope. [28] proposes a combination between machine learning techniques and synthetic kernels to calculate the tile size for a specific class of applications, but limits their search to cubic tiles and only takes one level of cache hierarchy into account. The authors in [18] propose an analytical model to optimize nested loops with a combination of tiling and interchange. However, their work is focused on embedded accelerators and thus all interaction with the cache is ignored, leading to suboptimal results of the model in cache-based systems.

In [14] the authors consider multi-level cache hierarchies in order to exploit reuse in both L1 and L2 cache levels while taking associativity into account. We use a similar analysis for our temporal locality optimizer that exploits reuse in L1 and L2 cache, but extends it in order to also take the hardware prefetching mechanisms and multi-core aspects of current architectures into account and to generate the loop nest permutation that takes advantage of those features. In [15], interaction with the hardware prefetching mechanisms is considered in order to achieve reuse in the L3 cache. However, both techniques rely on the compiler in the back-end to find the optimal loop order before performing any analysis. To this end, we propose a combined approach that considers loop tiling and loop ordering at the same time. Furthermore, they only consider tiling for applications with some form of temporal locality, which may lead to suboptimal results in situations where tiling should be focused only on self-spatial reuse.

Other approaches have focused on empirical autotuning methods that exhaustively try to optimize an application [2, 7, 24, 25]. In [25] an example of such a method is presented, that generates an optimized BLAS library for a target platform on the Pluto framework. However, such approaches usually require a large amount of time in order to explore the entire design space and converge to an efficient solution and furthermore cannot be used without access to the target architecture.

Hybrid methods have also been presented where both analytical models and exhaustive searches are used [8, 10, 19, 21]. For example, in [21] an analysis is conducted to obtain bounds on the search space that should be explored. The authors consider data reuse in multiple levels of the cache hierarchy but ignore cache associativity.

Halide [20] is a relatively new programming language that enables the separation of a target algorithm from its optimization schedule. For this reason, it is a good target environment for testing our optimization algorithm. Similar to our approach, the Halide Auto-Scheduler [16], attempts to automatically generate an optimization schedule for a

given function by using a heuristic based optimization algorithm. However, the authors' approach focuses on finding the best loop fusion options in image processing pipelines with numerous stages and thus the cache and tiling analysis it employs is limited (considering only one level of cache hierarchy). This leads to suboptimal results in small memory intensive applications. Moreover, it uses the bounds inference information provided by the back-end compiler regarding memory accesses and footprint and is thus unable to discern patterns in the source code. The Halide autotuner implemented in the autotuning framework Opentuner [2], is another method that automatically generates optimization schedules by iteratively running an application using different optimization configurations. The autotuner needs a large amount of time to search the design space, while providing no guarantee regarding the quality of the final solution. Furthermore, part of the design space is sometimes actually excluded from the search space, and thus the framework may be incapable of finding the optimal solution altogether.

3 Proposed Method

In this section we present the general optimization flow of our proposed method and demonstrate the analysis involved. Figure 1 shows the generic procedure that takes an algorithmic description of a loop nest as input, classifies it in order to decide whether to apply loop transformations as well as which combination of them. It finally performs parallelization and vectorization (if supported by the target architecture) in order to produce an optimization schedule for it. Furthermore, if the optimizer detects that the output data is not used in future loop iterations, then non-temporal store instructions are used in order to bypass the cache and reduce cache pollution. Non-temporal stores can help assure that data fetched into the cache by the hardware prefetchers do not get evicted before they can actually be used.

Table 1 lists all the application and architecture-specific parameters that are required throughout the optimization process. In the former category, we consider the problem size (loop bounds in each dimension) as well as the size of the data type as the main parameters of interest. For architecture-specific parameters, most of the information we need refers to the memory hierarchy of the system. Other parameters may include the native vector width of the architecture, and the number of processing units.

3.1 Classification

The first step in the optimization flow is the classification of the algorithm definition. The main purpose of this step is to decide whether transformations should be applied on the target loop nest, and whether they should focus on optimizing for temporal or spatial locality. The reason behind this distinction is twofold. Firstly, tiling (and therefore altering the stride of most load operations of a program) a loop nest with only contiguous memory accesses or no temporal locality may interfere with the efficiency of the streaming hardware prefetching unit and lead to suboptimal results. Secondly, tiling for (self-) spatial locality requires a different analysis since in these applications the only notion of reuse would refer to cache line reuse, or more specifically to data that belong to the same cache line.

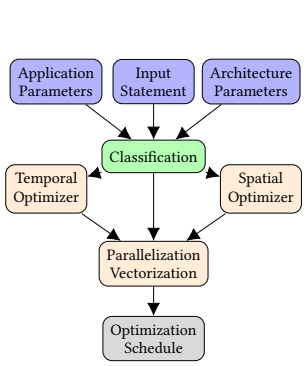


Figure 1. Optimization Flow

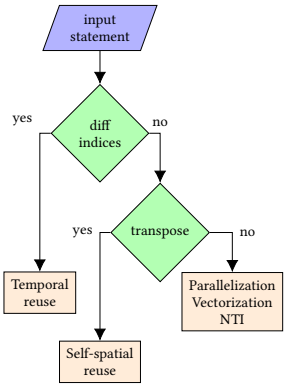


Figure 2. Classification Process

The classification process that specifies whether to transform the loop nest and therefore optimize for temporal or spatial locality can be seen in Figure 2.

Table 1. Architecture and application Parameters

Li_{CLS}	Li cache line size
Li_{way}	Li cache associativity
Li_{CS}	Li cache size
B_i	Problem size in ith Dimension
D_{TS}	Data Type Size
N_{Cores}	Number of Cores
$N_{threads}$	Threads per Core

We first check if the unique indices in the input arrays of the algorithm description are different from the ones in the output array. In that case we have multiple cache line references in our algorithm with temporal reuse possibilities. If we do not detect such a pattern, then it either means that only self-spatial reuse may be exploited, or that the algorithm contains only contiguous memory accesses and applying any loop transformation may alter the stride of the load operations and therefore interfere with the efficiency of the prefetching mechanisms. To optimize for spatial reuse we check whether any arrays appear transposed in the statement. In this case, we transform the loop in order to ensure that useful data fetched to the L1 cache due to prefetching will not get evicted before they can be used. If none of the above patterns exist in the statement, then no further analysis is needed and no loop transformations are deemed beneficial. This decision is also supported by the work in [9], which explains that tiling may not be effective for stencil computations (even though they might reference multiple cache lines and therefore have some form of temporal reuse)

Table 2. Basic notation

l_c	Amount of data that fits in one cache line
N_{sets}	Number of sets in cache
T_{width}	Tile width
B_c	Loop Bounds in the leading (column) dimension
max_{Ti}	Maximum tile size in ith dimension
T_{dims}	Number of tile dimensions
T_i	Tile size in ith dimension
ws_{Li}	Li cache working set
a_i	Li access time cost
C_{Li}	Estimated misses in Li cache
C_{order}	Loopnest permutation cost
$L2_{pref}$	L2 cache prefetches per access
$L2_{maxpref}$	Maximum prefetch distance
Lie_{way}	Effective associativity of Li cache

```
for ii=0;ii<Bi;ii+=Ti
 for kk=0;kk<Bk;kk+=Tk
  for jj=0;jj<Bj;jj+=Tj
   for i=ii;i<ii+Ti;i++
    for k=kk;k<kk+Tk;k++
     for j=jj;j<jj+Tj;j++ // vector loop
      C[i][j]=C[i][j]+A[i][k]*B[k][j]
```

Listing 1. Tiled matrix multiplication

due to uniform access patterns that can be easily exploited by the hardware prefetchers in modern architectures which can achieve the same level of reuse without the loop overhead of tiling.

3.2 Optimizations for Temporal Reuse

This section presents the analytical model, as well as the procedure that is followed in order to determine both the dimensions of the tile and the final loop permutation.

In general, our goal is to exploit reuse in both L1 and L2 cache in order to minimize the overall number of cache misses. More specifically, we pick tile dimensions such that L1 reuse is achieved in the outermost intra-tile loop and L2 reuse in the innermost inter-tile loop. The shared cache (L3) is also implicitly considered during the optimization procedure; modern hardware prefetching units are also capable of detecting non-unit strides in load operations in which case they fetch the expected data to the last-level-cache (and usually to the L2 as well). To better exploit this feature, we also aim to minimize the inter/intra-tile distance of each loop, therefore minimizing the stride of the equivalent load operations as well.

As an example, consider the code in Listing 1 which shows a simple C implementation of tiled matrix multiplication. In this case the classifier will recognize that different indices appear in the left and right side of the statement and thus we should exploit temporal reuse. We want to achieve L1 cache reuse at the outermost intra-tile loop level (i).

For the loop nest of Listing 1, an iteration of the i loop accesses/loads a row of width T_j from array C, a row of width T_k from array A and a tile of size $T_k * T_j$ from array B. Thus the working set for the L1 cache in this case will be :

$$ws_{L1} = T_j + T_k + T_jT_k \tag{1}$$

The total estimated cold misses in the L1 cache for one iteration of the i loop will be:

$$\frac{T_j}{l_c} + \frac{T_k}{l_c} + \frac{T_jT_k}{l_c} \tag{2}$$

However, due to the streaming prefetchers present in the L1 and L2 cache which fetch the next cache line after every reference, the estimated cold misses will be:

$$1 + 1 + T_k \tag{3}$$

Furthermore, for Ti iterations of the i loop, Equation 3) becomes:

$$T_i + T_i + T_k \tag{4}$$

Finally, the total number of estimated misses in the L1 cache after taking the inter-tile loop nest iterations into account will be:

$$C_{L1} = (T_i + T_i + T_k)(\frac{B_iB_jB_k}{T_iT_jT_k}) \tag{5}$$

Similarly, we want to achieve L2 reuse at the innermost inter-tile loop level (jj). One iteration of the jj loop will access a whole tile of arrays A, B and C. In this case the working set for the L2 cache will be:

$$ws_{L2} = T_jT_i + T_kT_i + T_jT_k \tag{6}$$

Moreover, just like for the L1 cache, the estimated number of cold misses for one iteration of the jj loop will be:

$$\frac{T_jT_i}{l_c} + \frac{T_kT_i}{l_c} + \frac{T_jT_k}{l_c} \tag{7}$$

Which after eliminating the prefetched references becomes:

$$T_i + T_i + T_k \tag{8}$$

Which in turn for $\frac{B_j}{T_j}$ iterations of the jj loop :

$$T_i\frac{B_j}{T_j} + T_i + T_k\frac{B_j}{T_j} \tag{9}$$

And finally after taking the other two inter-tile loops (kk,ii) into account we can compute the total estimated cost for the L2 cache:

$$C_{L2} = (T_i\frac{B_j}{T_j} + T_i + T_k\frac{B_j}{T_j})\frac{B_i}{T_i}\frac{B_k}{T_k} \tag{10}$$

After computing both (5) and (10) we can compute the final cost function:

$$C_{total} = a_2C_{L1} + a_3C_{L2} \tag{11}$$

We use a weighted cost function where the a_2 and a_3 are the relative access times of L2 and L3 cache respectively. We assume that the hardware prefetching unit can follow the strides of the memory references and therefore fetch the equivalent data into the L2 and L3 cache.

C_{order} is the cost function that describes the total distance in terms of iterations between the equivalent intra and inter tile loops. In detail the partial costs for Listing 1 are: T_iT_k,

$\frac{B_j}{T_j}T_i$ and $\frac{B_j}{T_j}\frac{B_k}{T_k}$ for the j, k, i original loops respectively. The total loop permutation cost would be:

$$C_{order} = (\frac{B_jB_k}{T_jT_k} + \frac{B_jT_i}{T_j} + T_iT_k) \quad (12)$$

It is obvious that for a different inter-tile or intra-tile permutation, a different loop would be at the outermost intra-tile loop level (or innermost inter-tile), which in turn would lead to a different L1/L2 working set and a different number of estimated misses. This explains why we evaluate all possible permutations.

Algorithm 1 is used to acquire an upper bound on the dimensions of the tile, such that no interference misses occur. In detail, it emulates the behavior of the cache, by fetching tile rows into the array emu_{cache} and testing whether the set that the new data will be mapped to is already full, at which point the interference flag ($intr_{flag}$) becomes true and the upper bound max_{Ti} is returned. Furthermore, the algorithm keeps track of the prefetched data that might cause interference misses in the following way: If we are optimizing for the L1 cache, then we also need to consider the fact that for every cache line that is fetched, the next one is also brought into the cache by the hardware prefetcher. When optimizing for the L2 cache, we need to take into account that more than one prefetching requests may be issued at once, usually with a maximum distance between the actual reference and the prefetched data (usually 20 for Intel processors). For this reason we also fill the array with these extra lines in order to detect situations where the prefetched data might cause useful data to be evicted. To accomplish this, we track the total number of prefetched lines (s), as well as the distance between the actual reference and the prefetched line ($s - p$). Finally, in the case of L2, we limit the effective number of sets to half the original size. In other words, we reduce the effective cache size by half (and thus size of ws_{L2}) to account for the data that are fetched by the constant stride prefetchers. As the experiments show in Section 5, this leads to efficient results, especially in the case of processors without L3 cache where data is only brought to L2. All the relevant notation can be found in Table 2.

Algorithm 2 shows the procedure that is followed in order to optimize a loop nest for temporal locality. The first step is to obtain the proper tile dimensions that minimize misses in the L1 and L2 cache. To achieve this we evaluate all possible tile sizes, as constrained by the bounds returned by the cache emulation algorithm (Algorithm 1) (for the first three dimensions) and problem size (for loop nests with four or more levels) for all valid intra-tile and inter-tile permutations. Invalid permutations are considered those where the loops that correspond to column indices are outermost. For each possible tile we calculate the size of the working set in the L1 and L2 cache to ensure that the tile fits in the cache (in order to minimize capacity misses) and finally if the dimension that corresponds to the outermost intertile

Algorithm 1 Cache emulation Algorithm (emu)

Input: $L1_{CLS}$, Li_{CS}, D_{TS}, T_{i-1}, Li_{way}, B_i, $N_{threads}$, $addr$, $L2_{pref}$, $L2_{maxpref}$
Output: max_{Ti}

```
Initializations :
l_c = ⌊L1_CLS / D_TS⌋
N_sets = ⌊Lics / (Liway * D_TS)⌋
Lie_way = Li_way / N_threads
max_Ti ← 0, s ← 0
intr_flag ← False
if optimizing for L2 then
    T_{i-1} = ⌈max(T_{i-1}, l_c) / l_c⌉
    N_sets = N_sets / 2
else
    T_{i-1} = ⌈max(T_{i-1} + l_c, 2 * l_c) / l_c⌉
end if
emu_cache[N_sets] = 0
repeat
    set ← ⌈(addr + max_Ti * B_i) / l_c⌉
    for i = 0 to T_{i-1} do
        if emu_cache[(set + i)] = Lie_way then
            intr_flag ← True
        else
            emu_cache[(set + i)] + +
            s + +
        end if
        if s − i <= L2_maxpref then
            for p = 0 to L2_pref do
                if emu_cache[(set + i + p)] = Lie_way then
                    intr_flag ← True
                end if
            end for
        end if
    end for
    if intr_flag = False then
        max_Ti + +
    end if
until intr_flag = True OR max_Ti = B_i
Return max_Ti
```

loop (the one that we plan to parallelize over cores/threads) fulfills the following constraint:

$$B_{outer}T_{outer} \geq N_{threads/core} * N_{cores} \quad (13)$$

This constraint ensures that each core/thread can execute at least one iteration of the inter-tile loop nest in order to better distribute the computation load among the processing units. The tile dimensions that correspond to the minimum total cost are chosen for the final tile size. To better utilize the prefetching units, we introduce a second step in our procedure where we try to minimize the distance between the inter and intra-tile loops that correspond to the same original loop in the original nest. This way we minimize the reuse distance of the equivalent data, as well as the stride of the load operation that will occur on the next inter-tile reference. Finally, after tiling and reordering the loop nest, we fuse the outer inter-tile loops when possible to reduce loop overhead and further exploit parallelism.

Algorithm 2 Temporal Reuse Optimizer

Input: $L1_{CLS}$, $L2_{CLS}$, $L1_{CS}$, $L2_{CS}$, $L1_{way}$, $L2_{way}$, N_{cores}, B_0, .., B_n, D_{TS}
Output: *Tile size, Loop order*

```
Step 1: Loop Tiling:
i ← 0
for Every inter-tile loop permutation do
  for Every intra-tile permutation do
    if Column index is outermost then
      Skip to next permutation
    end if
    repeat
      PickT_i ≤ B_c
      i ← i + 1
      max_Ti = emu(L1_CLS, L1_CS, D_TS, L1_way,
      B_c, N_threads, addr, 0, 0)
      Pick T_i ≤ max_Ti
      if (T_dims > 2) then
        i ← i + 1
        max_Ti = emu(L2_CLS, L2_CS, D_TS, L2_way, B_i,
        N_threads, addr, L2_pref, L2_maxpref)
        Pick T_i ≤ max_Ti
        if (T_dims > 3) then
          for i=3 to T_dims do
            Pick T_i ≤ B_i
          end for
        end if
        Calculate ws_L2, Estimate C_MissL2
      end if
      Calculate ws_L1, Estimate C_L1
      if (ws_L1, ws_L2 fit in cache and iterations per thread ≥ 1)
      then
        CostFunction=(a2C_L1 + a3C_L2)
      end if
    until all valid tile sizes evaluated
  end for
end for
Step 2: Reorder Loop:
for Every valid inter-tile loop permutation do
  for Every valid intra-tile permutation do
    Calculate C_order
  end for
end for
```

3.3 Optimizations for Spatial Reuse

Optimizing for spatial locality is important in applications with no temporal reuse possibilities. This section presents the analytical model and the procedure to obtain tile dimensions that take advantage of the streaming hardware prefetching units in applications with complex strides like transposed arrays.

As an example, consider the C code in Listing 2 which shows a tiled implementation of a transposition and masking algorithm.

```
for yy=0;yy<By;yy+=Ty
 for xx=0;xx<Bx;xx+=Tx
  for y=yy;y<yy+Ty;y++
   for x=xx;x<xx+Tx;x++
    out[y][x]=A[x][y]&B[y][x]
```

Listing 2. Tiled Transposition and Masking

In this case the classifier detects that the indices are the same in the input and output arrays, and that one array appears transposed in the statement. As a result the algorithm is optimized targeting spatial locality, using Algorithm 3.

We again assume the presence of a streaming prefetcher in both levels of the cache hierarchy, which means that the processor will fetch the next cache line for memory references in A and B. Just like in the previous section, the cost of accessing one tile of the transposed array A will be equal to T_x which after taking the inter-tile loops into account becomes:

$$T_x \frac{B_x B_y}{T_x T_y} = \frac{B_x B_y}{T_y} \tag{14}$$

The total cost for array A will be:

$$C_{partial} = (\frac{B_x B_y}{T_y}) \frac{T_x}{l_c} \tag{15}$$

where we refer to the factor $\frac{T_x}{l_c}$ as the prefetching efficiency for array A which represents the efficiency of the constant stride prefetching unit in the L2 cache. This factor gets minimized for $T_x = l_c$ (assuming that all tiles have a minimum of l_c size in every dimension). In other words, the transposed array favors tiles that have the maximum height and the minimum width. Similarly, for array B the cost of accessing one tile be equal to T_y which after taking the inter-tile loops into account becomes:

$$T_y \frac{B_x B_y}{T_x T_y} = \frac{B_x B_y}{T_x} \tag{16}$$

The total cost for array B will be:

$$C_{partial} = (\frac{B_x B_y}{T_x}) \frac{T_x}{l_c} \tag{17}$$

Finally the working sets for the two levels of cache:

$$ws_{L1} = l_c T_x + T_x \tag{18}$$

$$ws_{L2} = 2 T_x T_y \tag{19}$$

Algorithm 3 shows the pseudocode for the spatial locality optimizer. Just like in the previous section, we use Algorithm 1 to obtain an upper bound for the tile dimensions (tile height for 2 dimensional arrays). We calculate the working sets and if the tile height also fulfills equation (6), then for each input array we calculate the partial cost ($C_{partial}$) as explained in the previous example (equations (16),(17)). The final cost function C_{Total} is equal to the sum of all $C_{partial}$ costs. We evaluate all valid tile sizes as constrained by the bounds returned from Algorithm 1 and the problem size in the leading (column) dimension, and the tile that corresponds to the minimum C_{total} is chosen as the final tile size.

3.4 Parallelization, Vectorization, NTI

Standard optimizations include performance optimizations that can be applied after properly transforming the loop nest. These optimizations usually include vectorization and parallelization. Another possibility is the usage of non-temporal

Algorithm 3 Spatial Locality Optimizer

Input: $L1_{CLS}$, $L2_{CLS}$, $L1_{CS}$, $L2_{CS}$, $L1_{way}$, $L2_{way}$, B_0, .., B_n, D_{TS}
Output: *Tile size*

```
Initializations :
l_c = ⌊ L1_CLS / D_TS ⌋,
repeat
    C_Total ← 0
    Pick T_width ≤ B_c
    i ← i + 1
    max_Ti = emu(L2_CLS, L2_CS, D_TS, L2_way, B_i,
    N_threads, addr, L2_pref, L2_maxpref)
    Pick T_i ≤ max_Ti
    Calculate ws_L2
    Calculate ws_L1
    for Every input array do
        if (ws_L1, ws_L2 fit in cache and iterations per thread ≥ 1) then
            Calculate C_partial
            C_Total+ = C_partial
        end if
    end for
until all valid tile sizes evaluated
```

stores in applications with no temporal reuse in the output data. For applications with contiguous memory accesses no further loop optimization is usually needed since the streaming hardware prefetching units are capable of fetching the next cache line and thus the data that will be needed in the near future, and this is why in such cases we bypass all loop transformations during the optimization flow.

4 Experimental Framework

In this section, we present the experimental framework that was developed for the Halide DSL and compiler [20]. The red box in Figure 3 highlights the optimizer that is described in this work and which is implemented as a tool to be used with Halide. The Halide language separates the algorithm description from its schedule and therefore enables fast design space exploration with minimal effort. Listing 3 gives an example implementation of a matrix multiplication implementation in the Halide language, along with an optimization schedule. We should emphasize that our proposed optimization flow can be used with any other compiler/back-end but the Halide DSL was chosen in order to make use of the scheduling directives that enable quick application of various loop transformations and optimizations as seen in Listing 3. Such a framework is especially useful in many applications in the image processing domain, where most parameters are fixed and known at compile-time.

As already mentioned in Section 2, there are currently two ways to generate optimization schedules for Halide functions: the Halide Auto-Scheduler [16] uses a heuristics-based algorithm to decide on the tile size and final loop permutation, while the autotuner [2] iteratively searches the design space with various schedule configurations in order to minimize the execution time of the final application. We use those two approaches as references for comparison with our framework.

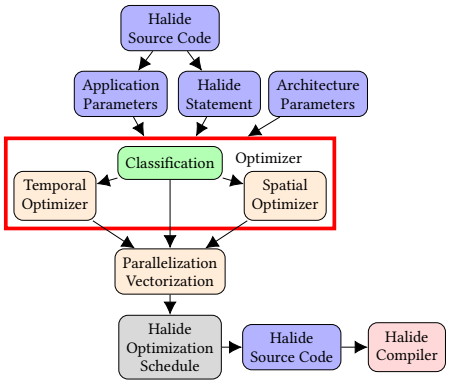


Figure 3. Experimental Halide Optimization Flow

```
//Algorithm Definition
C(j,i)=0;
C(j,i)=C(j,i)+A(k,i)*B(j,k);
//Optimization Schedule
C.update().split(j,j_o,j_i,512)
  .split(i,i_o,i_i,32)
  .reorder(j_i,i_i,j_o,i_o)
  .vectorize(j_i,8)
  .parallel(i_o);
```

Listing 3. Matrix Multiplication in Halide

Our framework requires the definition of a Halide function, along with the application and architecture specific parameter as input to the optimizer. The Halide statement is then processed during the classification step, and depending on the information that is derived and the patterns that can be recognized, a different optimization technique is used, as explained previously in Section 3.

Furthermore, since the Halide compiler cannot generate non-temporal instructions, we extend it with a new scheduling directive that produces non-temporal stores in the generated code when used in the optimization schedule of a function. To this end, we introduce a new optimization to the Halide front-end that allows the compiler to mark a function and the subsequent Halide Intermediate Representation (IR) store nodes as non-temporal in order to internally use that information to generate specific instructions (both scalar and vector variants) with non-temporal hints during the LLVM code generation pass in the back-end. Examples of such instructions (and the ones generated by the compiler in the following experiments) in Intel platforms with SSE/AVX support are the vector operations *(v)movntdq, (v)movntps* for integer and single precision floating point data types respectively.

5 Experimental Results

5.1 Comparison to Halide approaches

This section presents the results that were obtained for a variety of benchmarks. All experiments were conducted multiple times measuring the average execution time of 100 runs

for each benchmark. The run-time between different runs of the same experiment was less than 1%.

We compare our results with the equivalent that the Halide Auto-Scheduler and autotuner generate on the same platform. Table 3 shows the hardware specifications of the target architectures that were used throughout the experimental process, while Table 4 lists the benchmarks, the problem size used in each of them along with the average execution time of the best implementation for each benchmark to be used as reference for the following graphs. We chose three different architectures to showcase the flexibility of our approach in platforms with different architectural parameters. Specifically, the two Intel platforms differ in the number of cores and therefore may lead to different tile sizes (Equation 13), while the ARM architecture operates on a completely different memory hierarchy and utilizes one thread per core. Finally, Table 5 shows the runtime of our framework for each benchmark. In most cases, the tool is able to provide solutions within milliseconds, with the only exception being the convolution layer benchmark due to the large number of nested loops present in the tiled version of the algorithm and therefore the large number of possible loop permutations.

Table 3. Experimental Platforms

	Intel i7 5930k	Intel i7 6700	ARM Cortex A15
L_{CLS}	64B	64B	64B
$L1_{way}$	8	8	2
$L1_{CS}$	32KB	32KB	32KB
$L2_{way}$	8	8	16
$L2_{CS}$	256KB	256KB	512KB
N_{Cores}	6	4	4
$N_{threads}$	2	2	1

Figure 4 shows the throughput (1/s) relative to the fastest implementation for the two Intel platforms. The autotuner bar refers to the schedule that the Halide autotuner converges to after one hour of runtime. The Baseline bar corresponds to the most basic optimization a developer may perform, which usually includes parallelization of the outer loop and vectorization of the inner one. Finally, in order to make the comparison clearer, and since neither the autotuning nor the autoscheduling methods are able to generate non-temporal instructions, we separate the results where the classifier decides to use streaming stores. The first nine benchmarks (convolution layer, doitgen, matmul, 3mm, trmm, gemm, syrk, syr2k) have been optimized for temporal reuse, while the transposition, transposition and masking, copy and mask kernels have been optimized for spatial reuse. Non-temporal instructions can also be used for the four last algorithms.

The autotuning framework generates relatively poor schedules for most benchmarks either because it excludes schedules with tiling in all dimensions, or because it needs even more time to converge to a better solution. Due to this reason, we performed an extra experiment for the matrix multiplication, doitgen, convolution layer and transposition and masking benchmarks, where we compare our solution to the schedule generated by the autotuner after one day of runtime. It should also be noted that most of the applications had to be rewritten in a Halide-specific way that uses helper functions (e.g. sum) and bypasses the initial definition of the algorithms (e.g. initialize sum to zero) in order to be optimized by the autotuner. These functions rely on the compiler to perform some optimizations instead of actual loop transformations specified by the developer. Without this alternate definition, the autotuner would only attempt to optimize the initialization step and not the actual computation. The syrk and syr2k benchmarks could not be rewritten in such a way and thus the autotuned implementations are excluded.

Figure 5 demonstrates the performance of the solutions generated after one day of autotuning along with the results of our framework. We chose algorithms with different loop dimensions (2, 3, 4, 5 dimensions for the transpose and masking, matrix multiplication, doitgen and convolution layer respectively) in order to compare our analysis for transforming N-dimensional loops with stochastic autotuned methods. These results are similar to the ones presented in Figure 4 and therefore strengthen our decision to tile each dimension of the input loop nest, as opposed to the autotuner schedules that only attempt tiling in the dimensions of the output array.

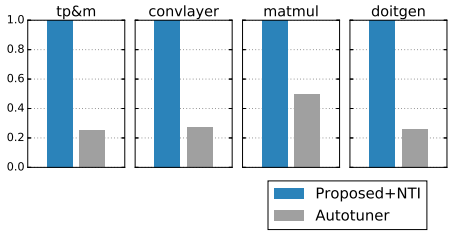


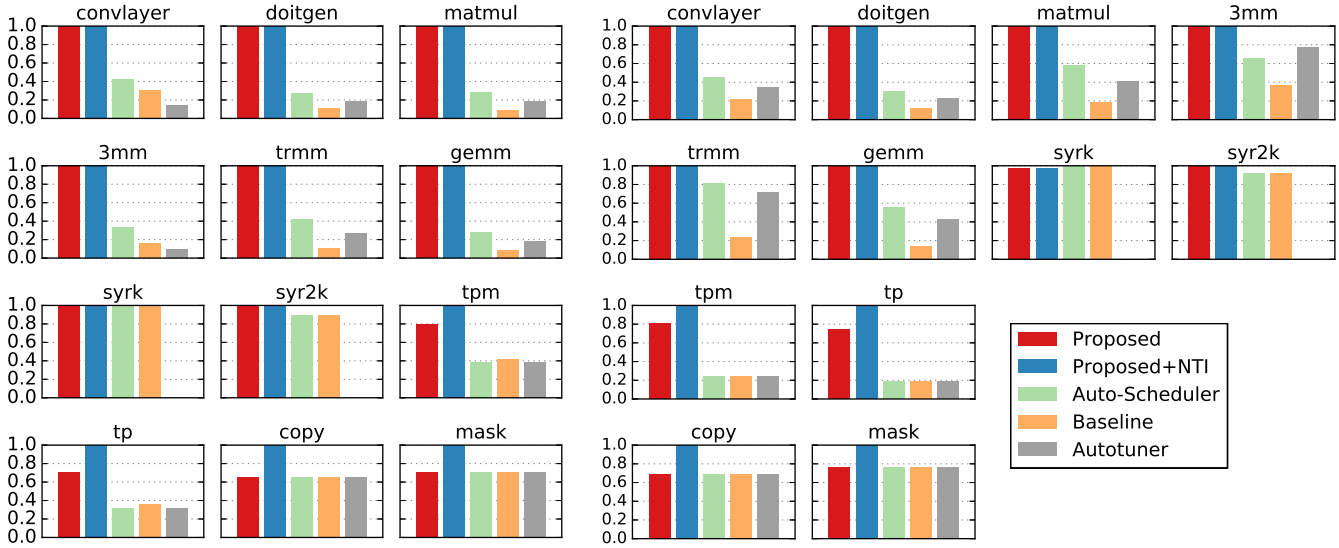
Figure 5. Throughput (1/s) relative to fastest implementation; autotuner ran for 1 day on Intel 5930K

The schedules provided by the Auto-Scheduler offer a significant speed-up compared to both the baseline schedules and the autotuned ones. However, our schedules still perform significantly better for most benchmarks. The syrk and syr2k benchmarks are the only exceptions where our approach performs similar to the baseline schedule due to the fact that the algorithms contain references along the cache line, and therefore do not significantly benefit from tiling. However, as expected, after repeating the experiments for larger problem sizes, the tiled version performed around 25% better than the baseline schedule.

Figure 6 shows the effect of non-temporal store instructions in the applications where the classifier does not detect output data reuse for the Intel i7-5930K platform. As seen in the graph, this optimization can significantly improve the performance in applications with no temporal reuse on the

Table 4. Benchmarks

			Average execution time (ms) - Best implementation		
Benchmark	**Description**	**Problem Size**	**Intel i7 6700**	**Intel 5930K**	**ARM A15**
convlayer	3x3x64x64 Convolution Layer	256x256x64x16	887.12	503.80	8897.29
doitgen	Multiresolution Analysis Kernel	256x256x256	233.29	143.77	2824.87
matmul	Matrix Multiplication	2048x2048	298.97	182.24	2080.58
3mm	Linear Algebra Kernel - three matrix multiplications	2048x2048	310.97	178.90	1564.15
gemm	Generalized Matrix Matrix Multiplication	2048x2048	286.12	183.00	1503.06
trmm	In-place Triangular Matrix Matrix Multiplication	2048x2048	199.44	131.76	1295.14
syrk	Symmetric rank k update	2048x2048	742.57	364.80	3575.62
syr2k	Symmetric rank 2k update	2048x2048	1442.41	992.61	7269.75
tpm	Matrix Transposition and Masking	4096x4096	10.02	6.00	41.87
tp	Matrix Transposition	4096x4096	7.23	4.5	39.00
copy	Array Copy	4096x4096	5.49	3.18	-
mask	Array Mask	4096x4096	8.32	4.67	-



(a) Intel i7 6700 **(b)** Intel i7 5930K

Figure 4. Intel platforms - Throughput (1/s) relative to fastest implementation (see Table 4)

output data due to a reduction of the total number of cache misses.

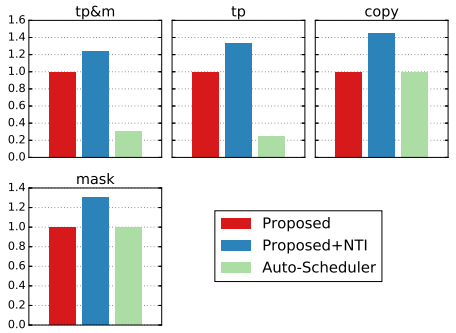


Figure 6. Throughput (1/s) relative to Proposed Non-NTI implementation (Intel 5930K)

Figure 7 demonstrates the results for the ARM Cortex A15 architecture. This architecture does not have an L3 cache and the L2 one is shared among the four cores of the platform. Due to this reason, a minor change to the model was required before conducting the experiments: The calculation of $L2_{way}$ in Algorithm 2 should be updated to $\frac{L2_{way}}{N_{cores}}$ instead of $\frac{L2_{way}}{N_{threads}}$ to account for this fact. Furthermore, since the ARM architecture does not support vector stores with non-temporal hints and all other implementations in Figure 7 are vectorized for increased performance, the mask and copy algorithms are not included in this graph (their performance is identical in all three implementations). Our proposed algorithm outperforms the Auto-Scheduler and baseline on this architecture as well.

5.2 Comparison to other tiling approaches

In this section, we compare our approach to previous state of the art analytical models for automatic tile size selection. Namely we pick the TSS method proposed in [14] as well as the TTS method which was introduced in [15]. We pick these

Table 5. Optimization runtime

Benchmark	convlayer	doitgen	matmul	3mm	gemm	trmm	syrk	syr2k	tp&m	tp	copy	mask
Runtime	7.604s	0.153s	0.006s	0.006s	0.006s	0.005s	0.009s	0.012s	0.002s	0.002s	0.002s	0.002s

Table 6. Average execution time (ms) - Intel 5930K

Problem Size	400			800			1024			1600		
Technique	TTS	TSS	Proposed	TTS	TSS	Proposed	TTS	TSS	Proposed	TTS	TSS	Proposed
matmul	1.76	**1.54**	1.65	12.08	15.06	**9.35**	23.56	71.97	**20.46**	98.65	104.18	**71.62**
tmm	1.01	1.19	**0.93**	5.93	22.42	**5.02**	**10.01**	35.46	10.57	58.47	137.83	**36.21**
syrk	14.80	7.20	**5.80**	96.24	115.07	**69.04**	224.88	228.83	**159.47**	242.11	294.84	**213.32**
syr2k	30.57	13.22	**11.29**	58.88	86.99	**51.21**	228.33	248.60	**97.14**	451.45	536.22	**314.38**

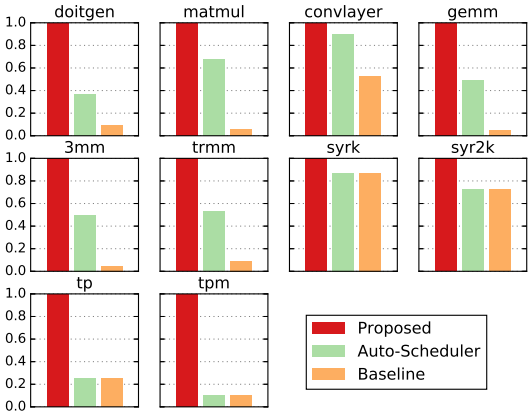


Figure 7. ARM platform - Throughput (1/s) relative to fastest implementation

techniques as they have similarities with our approach: The TSS method considers reuse in the L1 and L2 cache without taking prefetching into account, while the TTS technique optimizes for L2 and L3 cache while taking advantage of hardware prefetching. However, prefetching is not considered in the analytical model and prefetched references are not taken into account while estimating the number of cold misses in every iteration. As a result, the proposed tile sizes for both TTS and TSS are different than the ones picked by our approach.

Table 6 shows the average execution time (ms) on the Intel i7-5930K platform for the three methods. Since both TTS and TSS use a different framework and back-end compiler, we are not able to reproduce their optimization flow. For this reason, we choose this specific platform for our experiments, since it has similar cache hierarchy (L_{CLS}, $L1_{way}$, $L1_{CS}$, $L2_{way}$, $L2_{CS}$) as the one used in [15] in order to use the tile dimensions picked by TTS and TSS as listed in [15], with a difference on the size of the L3 cache and the number of cores. However, since the size of the L3 cache in both platforms is large enough, and the effective size per core is the same, we do not expect a big impact on the final tile dimesions. Furthermore, since nor TTS, nor TSS consider loop interchange, we try every possible loop permutation for each benchmark and pick the one that results in the best performance to include in our experiments. We compare the three techniques for the four benchmarks that are common between the ones used in [15] and the ones we used in Section 5.1 and four different problem sizes.

The results presented in Table 6 indicate that our method outperforms the other two techniques by up to two times on the syr2k benchmark. Furthermore, the experiments show that our proposed approach generates results which are on average 26% and 41% faster than the solutions provided by TTS and TSS respectively.

6 Conclusion

In this work we propose an optimization framework for memory bound applications that considers architecture and application specific parameters while taking advantage of the hardware prefetching mechanisms in modern platforms. We implement it as a tool to be used with the Halide DSL and compiler and compare it to both other analytical as well as empirical methods. Experimental results indicate a significant improvement in performance compared to previous models, while providing solutions usually within milliseconds. These results show that interaction between loop transformations and the sophisticated hardware prefetching mechanisms in modern architectures is of utmost importance when optimizing memory intensive applications.

Acknowledgments

This research was performed within the framework of the IMPULS-II program between TU Eindhoven and Océ Technologies.

References

[1] Jaume Abella. Near-optimal loop tiling by means of cache miss equations and genetic algorithms. In *Proceedings of the 2002 International Conference on Parallel Processing Workshops*, ICPPW '02, pages 568–, Washington, DC, USA, 2002. IEEE Computer Society.

[2] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. M. O'Reilly, and S. Amarasinghe. Opentuner: An extensible framework for program autotuning. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 303–315, Aug 2014.

[3] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling stencil computations to maximize parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking,*

Storage and Analysis, SC '12, pages 40:1–40:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[4] B. Bao and C. Ding. Defensive loop tiling for shared cache. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11, Feb 2013.

[5] Jacqueline Chame and Sungdo Moon. A tile selection algorithm for data locality and cache interference. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 492–499, New York, NY, USA, 1999. ACM.

[6] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. *SIGPLAN Not.*, 30(6):279–290, June 1995.

[7] Jim Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Rich Vuduc, R. Clint Whaley, and Katherine Yelick. Self adapting linear algebra algorithms and software. In *Proceedings of the IEEE*, page 2005, 2005.

[8] B. B. Fraguela, M. G. Carmueja, D. Andrade, G. R. Joubert, W. E. Nagel, F. J. Peters, O. Plata, P. Tirado, E. Zapata, Basilio B. Fraguela A, MartÃŋn G. Carmueja A, and Diego Andrade A. Optimal tile size selection guided by analytical models. In *In PARCO*, pages 565–572, 2005.

[9] Shoaib Kamil, Parry Husbands, Leonid Oliker, John Shalf, and Katherine Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *Proceedings of the 2005 Workshop on Memory System Performance*, MSP '05, pages 36–43, New York, NY, USA, 2005. ACM.

[10] P. M. W. Knijnenburg, T. Kisuki, K. Gallivan, and M. F. P. O'Boyle. The effect of cache models on iterative compilation for combined tiling and unrolling: Research articles. *Concurr. Comput. : Pract. Exper.*, 16(2-3):247–270, January 2004.

[11] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. *SIGPLAN Not.*, 26(4):63–74, April 1991.

[12] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Orti. Analytical modeling is enough for high-performance blis. *ACM Trans. Math. Softw.*, 43(2):12:1–12:18, August 2016.

[13] Qingda Lu, Sriram Krishnamoorthy, and P. Sadayappan. Combining analytical and empirical approaches in tuning matrix transposition. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT '06, pages 233–242, New York, NY, USA, 2006. ACM.

[14] Sanyam Mehta, Gautham Beeraka, and Pen-Chung Yew. Tile size selection revisited. *ACM Trans. Archit. Code Optim.*, 10(4):35:1–35:27, December 2013.

[15] Sanyam Mehta, Rajat Garg, Nishad Trivedi, and Pen Chung Yew. *TurboTiling: Leveraging prefetching to boost performance of tiled codes*, volume 01-03-June-2016. Association for Computing Machinery, 6 2016.

[16] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.*, 35(4):83:1–83:11, July 2016.

[17] Neungsoo Park, Bo Hong, and V. K. Prasanna. Analysis of memory hierarchy performance of block data layout. In *Proceedings International Conference on Parallel Processing*, pages 35–44, 2002.

[18] Maurice Peemen, Bart Mesman, and Henk Corporaal. Inter-tile reuse optimization applied to bandwidth constrained embedded accelerators. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 169–174, San Jose, CA, USA, 2015. EDA Consortium.

[19] Apan Qasem and Ken Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, pages 249–258, New York, NY, USA, 2006. ACM.

[20] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[21] Jun Shirako, Kamal Sharma, Naznin Fauzia, Louis-Noël Pouchet, J. Ramanujam, P. Sadayappan, and Vivek Sarkar. Analytical bounds for optimal tile size selection. In *Proceedings of the 21st International Conference on Compiler Construction*, CC'12, pages 101–121, Berlin, Heidelberg, 2012. Springer-Verlag.

[22] S. Tavarageri, L. N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. Dynamic selection of tile sizes. In *2011 18th International Conference on High Performance Computing*, pages 1–10, Dec 2011.

[23] O. Temam, C. Fricker, and W. Jalby. Cache awareness in blocking techniques. In *in Journal of Programming Languages*, 1998.

[24] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[25] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the atlas project. *PARALLEL COMPUTING*, 27:2001, 2000.

[26] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. pages 30–44, 1991.

[27] K. Yotov, Xiaoming Li, Gang Ren, M. J. S. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance blas? *Proceedings of the IEEE*, 93(2):358–386, Feb 2005.

[28] Tomofumi Yuki, Lakshminarayanan Renganarayanan, Sanjay Rajopadhye, Charles Anderson, Alexandre E. Eichenberger, and Kevin O'Brien. Automatic creation of tile size selection models. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 190–199, New York, NY, USA, 2010. ACM.

[29] Lixin Zhang, J. B. Carter, W. C. Hsieh, and S. A. McKee. Memory system support for image processing. In *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00425)*, pages 98–107, 1999.