Towards an Efficient Sequential Bulge-Chasing Kernel

Angelika Schwarz and Lars Karlsson

UMINF 17.20

Department of Computing Science

Towards an Efficient Sequential Bulge-Chasing Kernel

Angelika Schwarz[*]
angies@cs.umu.se

Lars Karlsson
larsk@cs.umu.se

ABSTRACT

The bulge-chasing kernel in the multi-shift QR algorithm is invoked repeatedly on the critical path. The bulge-chasing kernel operates on a computational window embedded in a bigger matrix. It requires multiple calls to the bulge-chasing kernel to chase all bulges; a highly optimised routine therefore reduces the length of the critical path and potentially the overall execution time. Off-diagonal blocks are updated with `DGEMM` operations. We provide an optimised sequential implementation that runs at 50% of the peak performance and is 5–15 times faster than `DLAQR5`, LAPACK's standard bulge-chasing routine.

1. INTRODUCTION

The multi-shift QR algorithm is the state-of-the-art method [1, 2] for computing all eigenvalues of a real-valued dense non-symmetric matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ and generalises the original double implicit shift QR algorithm by Francis. It computes a Schur decomposition $\mathbf{A} = \mathbf{Q}\mathbf{T}\mathbf{Q}^T$ where $\mathbf{Q} \in \mathbb{R}^{n\times n}$ is orthogonal and $\mathbf{T} \in \mathbb{R}^{n\times n}$ is block upper triangular with 1-by-1 or 2-by-2 blocks on the diagonal. The blocks correspond to real eigenvalues in the case of 1-by-1 blocks and to complex conjugate pairs of eigenvalues for 2-by-2 blocks.

The multi-shift QR algorithm initially reduces the matrix $\mathbf{A}$ to upper Hessenberg form $\mathbf{H}_0 \leftarrow \mathbf{Q}_0^T\mathbf{A}\mathbf{Q}_0$. Then a Schur decomposition is approximated iteratively. In each iteration, a polynomial p_k is computed in accordance with a shifting strategy. The shifts $\sigma_1, \ldots, \sigma_m$, $m \ll n$ are closed under complex conjugation in order to avoid complex arithmetic. A multi-shift QR step computes the QR factorisation

$$p_k(\mathbf{H}_k) = (\mathbf{H}_k - \sigma_1\mathbf{I})(\mathbf{H}_k - \sigma_2\mathbf{I})\cdots(\mathbf{H}_k - \sigma_m\mathbf{I}) = \mathbf{Q}_k\mathbf{R}_k$$

and updates $\mathbf{H}_{k+1} \leftarrow \mathbf{Q}_k^T\mathbf{H}_k\mathbf{Q}_k$. For a general matrix it is difficult to merge multiple shifts. However, the upper Hessenberg shape of $\mathbf{H}_k$ allows for implicit multi-shifts. A Householder matrix is computed such that it transforms $p_k(\mathbf{H}_k)\mathbf{e}_1$ into a multiple of $\mathbf{e}_1$. This reflection introduces an $(m+1)$-by-$(m+1)$ bulge in the top-left corner. The original Hessenberg shape is restored through the bulge-chasing procedure. For this purpose, a sequence of Householder matrices is constructed, which chases the bulge along the subdiagonal until it vanishes in the bottom right corner.

The choice of m is delicate. Increasing m improves the arithmetic intensity, but it also introduces shift blurring and negatively affects the convergence rate. Braman, Byers and Mathias [1] proposed a mathematically equivalent scheme, the so-called small-bulge multi-shift QR algorithm. Instead of introducing one big bulge, they pack multiple small bulges in a chain and thereby process several iterations simultaneously. Chasing a chain of bulges rather than single bulges increases the arithmetic intensity. For $m = 2$, their packing scheme introduces a tightly packed chain of 3-by-3 bulges in the top-left corner of the Hessenberg matrix. A good choice for the number of bulges is $n_\mathrm{b} = \lfloor n/6 \rfloor$, i.e., a half-way filled computational window [1, 7]. This approach is implemented in the LAPACK routine `DLAQR5`.

[*]Research supported by a DAAD (German Academic Exchange Service) scholarship.

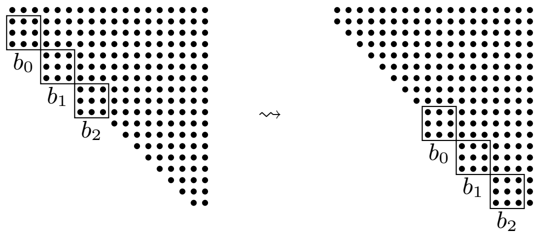


Figure 1: Input and output of the bulge-chasing kernel.

Karlsson, Kressner and Lang [7] presented an alternative packing scheme. Instead of packing bulges as a chain of 3-by-3 bulges as in Figure 1, delaying updates of the last row of bulges yields a tighter packing. The chain length can be reduced from $3n_\mathrm{b}$ to $2n_\mathrm{b} + 1$. This paper sticks to the original packing proposed by Braman, Byers, and Mathias because it is the packing used in `DLAQR5`, which we use as reference to compare with.

In a task-parallel implementation, the bulge-chasing kernel will be invoked repeatedly on a computational window embedded in a bigger matrix. The bulge-chasing kernel is on the critical path and limits the achievable speedup, especially near the limit of strong scalability. Then, in order to obtain further speedup, the length of the critical path has to be reduced. As the bulge-chasing kernel is hard to parallelise, we advocate optimizing the sequential bulge-chasing kernel. A faster implementation reduces the length of the critical path and, in turn, the overall execution time. This paper investigates how fast the sequential bulge-chasing kernel can become.

The rest of the paper is organised as follows. Section 2 introduces the single-step bulge-chasing mechanism and a

$$\begin{array}{|ccc|c} \multicolumn{1}{c}{h_{0,0}} & h_{0,1} & \multicolumn{1}{c}{h_{0,2}} & h_{0,3} \\ \hline h_{1,0} & h_{1,1} & h_{1,2} & h_{1,3} \\ h_{2,0} & h_{2,1} & h_{2,2} & h_{2,3} \\ h_{3,0} & h_{3,1} & h_{3,2} & h_{3,3} \\ \hline \multicolumn{1}{c}{0} & 0 & \multicolumn{1}{c}{0} & h_{4,3} \end{array} \rightsquigarrow \begin{array}{c|ccc|} h_{0,0} & h_{0,1} & h_{0,2} & \multicolumn{1}{c}{h_{0,3}} \\ \alpha & h_{1,1} & h_{1,2} & \multicolumn{1}{c}{h_{1,3}} \\ \cline{2-4} 0 & h_{2,1} & h_{2,2} & h_{2,3} \\ 0 & h_{3,1} & h_{3,2} & h_{3,3} \\ 0 & h_{4,3} & h_{4,3} & h_{4,3} \\ \cline{2-4} \end{array}$$

Figure 2: Moving a bulge one step.

reference code. Based on this reference code we identify a list of points that we discuss in Section 4 and Section 5 and finally lead to our optimised implementation presented in Section 6. We compare our optimised implementation to `DLAQR5` and motivate why our kernel is well-suited to be used within a task-based runtime system.

2. REFERENCE KERNEL

The input of the bulge-chasing kernel is an upper Hessenberg matrix $\mathbf{H} \in \mathbb{R}^{n\times n}$ perturbed with a chain of n_b tightly packed 3-by-3 bulges in the top left corner. The kernel chases the bulges along the diagonal to the bottom right corner. Figure 1 illustrates input and output. Note that we have excluded the corner cases of introducing and removing the bulges.

We exemplify the steps necessary to move a bulge with the situation of Figure 2 (note that we number rows and columns starting with 0 and not 1). The given bulge is moved one step along the diagonal through the following operations.

1. Construct a 3-by-3 Householder reflector $\mathbf{Q}_1 = \mathbf{I} - \tau\mathbf{v}\mathbf{v}^T$ and reduce the first column of the bulge to a multiple of the first unit vector:
$$\mathbf{Q}_1^T \begin{pmatrix} h_{1,0} \\ h_{2,0} \\ h_{3,0} \end{pmatrix} = \begin{pmatrix} \alpha \\ 0 \\ 0 \end{pmatrix}.$$
2. Compute the similarity transform $\mathbf{H} \leftarrow \mathbf{Q}_1^T\mathbf{H}\mathbf{Q}_1$. Apply $\mathbf{Q}_1$ to rows 1:3 from the left and to columns 1:3 from the right
$$\mathbf{Q}_1^T \begin{pmatrix} h_{1,1} & h_{1,2} & h_{1,3} \\ h_{2,1} & h_{2,2} & h_{2,3} \\ h_{3,1} & h_{3,2} & h_{3,3} \end{pmatrix} \text{ and } \begin{pmatrix} h_{0,1} & h_{0,2} & h_{0,3} \\ h_{1,1} & h_{1,2} & h_{1,3} \\ h_{2,1} & h_{2,2} & h_{2,3} \\ h_{3,1} & h_{3,2} & h_{3,3} \\ 0 & 0 & h_{4,3} \end{pmatrix} \mathbf{Q}_1.$$
3. Accumulate the reflector in the similarity transform matrix $\mathbf{Q}$.

For each move of a bulge a Householder reflector $\mathbf{Q}_j$ is created. The update $\mathbf{H} \leftarrow \mathbf{Q}_j^T\mathbf{H}\mathbf{Q}_j$ moves the bulge one step along the diagonal. Given a chain of bulges, the order of the bulges must be preserved and there may not be any collisions, i.e., bulges may never overlap. As the bulge-chasing kernel operates in a computational window, the accumulation of all reflectors $\mathbf{Q} = \mathbf{Q}_1\mathbf{Q}_2\mathbf{Q}_3...$ allows us to update off-diagonal blocks with an efficient `DGEMM` operation [1, 7].

The state-of-the-art small-bulge multi-shift bulge-chasing routine is `DLAQR5` in LAPACK. It is a generic routine, which can be configured in several ways. As we intend to call the bulge-chasing kernel as tasks processing a computational window,

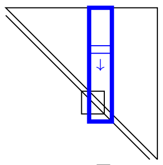
Algorithm 1 Bulge Chasing Reference Kernel

```
Require: number of bulges $n_\mathrm{b}$, Hessenberg matrix perturbed with a chain of tightly packed bulges $\mathbf{H} \in \mathbb{R}^{n\times n}$
Ensure: $\mathbf{H}$ with the chain of bulges in the bottom right corner, transformation matrix $\mathbf{Q} \in \mathbb{R}^{n\times n}$
  function ChaseBulges($n_\mathrm{b}$, $\mathbf{H} \in \mathbb{R}^{n\times n}$, $\mathbf{Q} \in \mathbb{R}^{n\times n}$)
    for bulge $b \leftarrow 0, ..., n_\mathrm{b} - 1$ do
      $b.m_Q \leftarrow 3$
      $b.n_H \leftarrow n - 3b - 1$
      $b.m_H \leftarrow 3b + 5$
    $\mathbf{Q} \leftarrow \mathbf{I}$
    for bulge $b \leftarrow n_\mathrm{b} - 1, ..., 0$ do
      for move $i \leftarrow 0, ..., n - 5 - 3(n_\mathrm{b} - 1)$ do
        $t \leftarrow 3b + i$
        // Reduce first column of bulge $b$.
        $\tau, \mathbf{v}$ = Dlarfg($\mathbf{H}_{t+1:t+3,t}$)
        // Update $\mathbf{H}$.
        LeftUpdate($\mathbf{H}_{t+1:t+3,t+1:t+b.m_H}$, $\tau$, $\mathbf{v}$)
        RightUpdate($\mathbf{H}_{0:b.n_H-1,t+1:t+3}$, $\tau$, $\mathbf{v}$)
        // Accumulate reflectors in $\mathbf{Q}$.
        RightUpdate($\mathbf{Q}_{3b+1:3b+b.m_Q,t+1:t+3}$, $\tau$, $\mathbf{v}$)
        // Update row and column counters.
        $b.n_H \leftarrow b.n_H - 1$
        $b.m_H \leftarrow b.m_H + 1$
        if $n_\mathrm{b} - b - 1 > i$ then
          $b.m_Q \leftarrow b.m_Q + 3$
        else
          $b.m_Q \leftarrow b.m_Q + 1$
```

we want to accumulate the reflectors instead of applying them directly. Moreover, `DLAQR5` contains vigilant deflation checks, which affect the runtime unpredictably. In order to have a predictable kernel behaviour, we define a reference kernel. We eliminate the vigilant deflation checks and replace the generic Householder reflector `DLARFG` with our own routine, which is cheap to compute due to the absence of exception checks and overflow protection. Otherwise, the reference kernel follows the compute pattern of `DLAQR5`; details and differences are discussed below. We leave it to the compiler to optimise the reference kernel and solely provide compiler hints. The reference kernel by itself is more than a factor of two faster than `DLAQR5`.

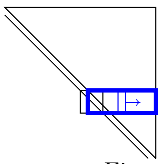
The purpose of this paper is to examine the speedup achievable through manual optimisations. Hence, we incrementally enhance the reference kernel with hand-tuned routines. We refer to this manually tuned routine as optimized version. In summary, we have three versions:

- **LAPACK**. The standard bulge-chasing routine `DLAQR5` that relies on generic helper routines and exhibits hard-to-predict behaviour.
- **Reference kernel**. A compiler-optimised kernel with predictable behaviour that resembles `DLAQR5` without vigilant deflation checks using specialised helper routines.
- **Optimised kernel**. A manually optimised version of the reference kernel.



```
function RightUpdate(
$\mathbf{H} \in \mathbb{R}^{n\times 3}$, $\tau \in \mathbb{R}$, $\mathbf{v} \in \mathbb{R}^{3\times 1}$)
    for $i \leftarrow 0, \dots, m-1$ do
        $s \leftarrow \tau(\mathbf{H}_{i,:}\mathbf{v})$
        $\mathbf{H}_{i,:} \leftarrow \mathbf{H}_{i,:} - s\mathbf{v}^T$
```

Figure 3: LAPACK-style right update.



```
function LeftUpdate(
$\mathbf{H} \in \mathbb{R}^{3\times n}$, $\tau \in \mathbb{R}$, $\mathbf{v} \in \mathbb{R}^{3\times 1}$)
    for $j \leftarrow 0, \dots, n-1$ do
        $s \leftarrow \tau(\mathbf{v}^T\mathbf{H}_{:,j})$
        $\mathbf{H}_{:,j} \leftarrow \mathbf{H}_{:,j} - s\mathbf{v}$
```

Figure 4: LAPACK-style left update.

Algorithm 1 shows the reference kernel. Reflectors are computed one at a time and directly applied to the computational window as well as the transformation matrix. In order to do so, the reference implementation relies on the compiler to optimise the dot-product-based matrix updates illustrated in Figure 3 and Figure 4. We harness the sparsity patterns to apply the updates to the minimum number of rows and columns. For this purpose, we introduce three counters n_H, m_H and m_Q for each bulge. More precisely, n_H stores the row count for the next right update of $\mathbf{H}$; m_H tracks the column count for the next left update of $\mathbf{H}$. Finally, m_Q stores the column count for the next right update of $\mathbf{Q}$. After each move we update the counters accordingly. Although it is possible to compute the exact counters based on the bulge number and the move index, we prefer to maintain three counters since this allows us to move bulges more freely.

Our implementation and the subsequent analysis are subject to two assumptions. First, matrices are stored in column major order. The storage scheme has profound impact on the cost of the matrix updates. We choose column major order to be compatible with LAPACK. Second, the computational window has a compact memory representation. This assumption is feasible because in a bigger bulge-chasing kernel it is reasonable to copy the computational window to a more compact memory representation to avoid huge column strides.

In contrast to our implementation, `DLAQR5` first computes the reflector vectors of all bulges. Afterwards, all reflectors are applied to the Hessenberg matrix first from the left and then from the right. In other words, the left update is completed before the right update is started. The reference kernel, by contrast, executes the left and the right update immediately when a reflector is available. In other words, the left and the right update are interleaved.

For a first analysis, we profile the reference implementation on a half-way filled 120-by-120 window. The matrix updates dominate the runtime; 60% of the time is spent in LeftUpdate, 21% in RightUpdate and 17% in the accumulation of the $\mathbf{Q}$ matrix. The performance gap between the two former matrix updates is striking. Although the computational cost is roughly the same, the left update takes three times longer than the right update.

We base the optimisations conducted in Section 4 and Section 5 on the following observations.

1. **Fixed number of moves.** Bulges may not overtake one another. As a consequence, a constant number of $n - 4 - 3(n_b - 1)$ moves is applied to each bulge.

2. **Non-overlapping updates.** The matrix updates conducted in RightUpdate and LeftUpdate affect non-overlapping entries in $\mathbf{H}$ for distinct bulges. Multiple function calls to LeftUpdate can therefore be executed at the same time, followed by multiple calls to RightUpdate. This argument also holds for updates to $\mathbf{Q}$. Note that this – even when driven to the extreme – does not coincide with the update pattern in `DLAQR5` because we literally permit simultaneous updates instead of applying the batch of reflectors successively.

3. **Independent loops.** The loop over the moves and the loop over the bulges are independent and can be exchanged. Visually this corresponds to either chasing all bulges simultaneously or chasing the bulges one after another. The two choices differ with respect to cache utilization.

4. **Local accumulation of Q.** Updates to the $\mathbf{Q}$ matrix $\mathbf{Q} = \mathbf{Q}_1\mathbf{Q}_2\mathbf{Q}_3\mathbf{Q}_4\dots$ can be bracketed and computed as $\mathbf{Q} = (\mathbf{Q}_1\mathbf{Q}_2)(\mathbf{Q}_3\mathbf{Q}_4)\dots$, which is discussed in Section 5.3. Thereby multiple updates can be applied jointly at the expense of building the reflector matrix explicitly.

5. **Alignment.** Vector loads and stores across cache line boundaries impose performance penalties. Provided that the Hessenberg matrix is aligned and its leading dimension is a multiple of the vector width, RightUpdate can benefit from aligned instructions.

6. **Reflector type.** The reflector $\mathbf{I} - \tau\mathbf{v}\mathbf{v}^T$ incurs additional multiplications with the scalar τ when executing the left and right update. Alternatively, a uniform reflector, i.e., $\tau = 1$, is more expensive to compute, but saves multiplications when updating the matrices. We examine the choice of the reflector in Section 4.

7. **Vectorisation of the left update.** The left update, $\mathbf{H} \leftarrow \mathbf{Q}_j^T\mathbf{H}$, operates on a 3-by-$n$ submatrix. Since we store the matrices in column major order, the columns of the submatrix are not consecutive in memory. As a result, the compiler cannot use vector load and store instructions. While we have to accept the strided memory access pattern, the insertion of artificial zeros allows us to manually vectorise this update, see Section 5.2.

8. **Grouped bulge chasing.** Chasing all bulges simultaneously or one after another are just two extreme choices. Another choice is moving the bulges in groups to their final position. This promises computational benefits while preserving good cache utilisation. The group size is a tuning parameter.

3. TEST ENVIRONMENT

We employ three test platforms. First, we use an Intel Xeon E5-2690 v3 ("Haswell") machine operating at 2.6 GHz. One core exhibits two 256-bit wide SIMD units processing four double-precision operands each. It processes two fused-multiply-add instructions per cycle. The peak performance of one core therefore calculates $2.6 \cdot 2 \cdot 4 \cdot 2 = 41.6$ [Gflops/s]. Second, we use an Intel Xeon E5-2690 v4 ("Broadwell") with a base frequency of 2.6 GHz. Third, we use an Intel Core i5-6600 ("Skylake") with a base frequency of 3.3 GHz. The latter two CPUs exhibit the same hardware characteristics as the Haswell platform and have dynamic frequency scaling enabled. The nominal peak performance therefore amounts to 41.6 Gflops/s and 52.8 Gflops/s, respectively. We abbreviate the three test platforms in accordance to their microarchitecture with hsw, bdw, and skl. Note that the real peak performance deviates from the nominal peak performance because (a) the AVX core frequency is lower than the base frequency and (b) the clock speed may be adapted so that the processor meets power limits [5].

On all three platforms cache lines have a size of 64 bytes [4]. In order to avoid performance penalties from loads and stores that cut cache lines, we align the base pointers of all matrices and introduce zero-padding to properly align all columns. We inform the compiler about alignment wherever appropriate.

All runtime results report the wall-clock time for double-precision arithmetic. Throughout this report, we employ the Intel compiler version 17.0.1 with the optimisation flags `-malign-double -qopt-prefetch -O3 -unroll-aggressive -xHost -ipo`. Results reported for the GNU compiler refer to version 5.4.0. We consistently use the compile flags `-Ofast -march=native -funroll-loops -fprefetch-loop-arrays -malign-double -LNO:prefetch` and activate link time optimisation `-flto -O3`. We make extensive use of the `restrict` qualifier to allow for more aggressive compiler optimisation. We observe that both compilers reliably vectorise a good proportion of the loops, unroll small-sized loops and make heavy use of inlining.

All our measurements have been executed with exclusive node access. The runtime exhibits very little variation on the Haswell cluster. Results are also reliably reproducible on the Skylake platform. We observe runtime differences of up to 30% on the Broadwell cluster when run on distinct nodes. The ratio between the code versions, however, scales accordingly. We attribute this to different clock frequency scaling behaviour of the nodes. We have ensured that all comparisons have been executed on the same node to avoid different node behaviours.

4. HOUSEHOLDER REFLECTORS

In order to move a bulge, Householder reflectors are constructed. Given a 3×3 bulge, the Householder reflector transforms the first column of the bulge to a multiple of e_1. We use two reflector types. First, we use the standard reflector $\mathbf{I} - \tau \mathbf{v}\mathbf{v}^T$, which corresponds to LAPACK's `DLARFG`. Second, we define $w := \sqrt{v}$ and compute the uniform reflector $\mathbf{I} - \mathbf{w}\mathbf{w}^T$. This saves the multiplication with the scalar τ when updating the Hessenberg matrix and the similarity transform matrix.

	hsw		**bdw**		**skl**
Reflector	**GNU**	**Intel**	**GNU**	**Intel**	**GNU**
			scalar instructions		
standard	99.11	50.02	85.17	83.57	66.47
uniform	194.61	194.69	98.33	83.61	66.34
			vector instructions		
standard	62.52	51.53	90.11	88.56	47.25
uniform	119.55	115.43	113.58	110.35	55.57

Table 1: Median of 9 runs of the cumulated runtime to compute 10M reflectors. Timings are given in ms.

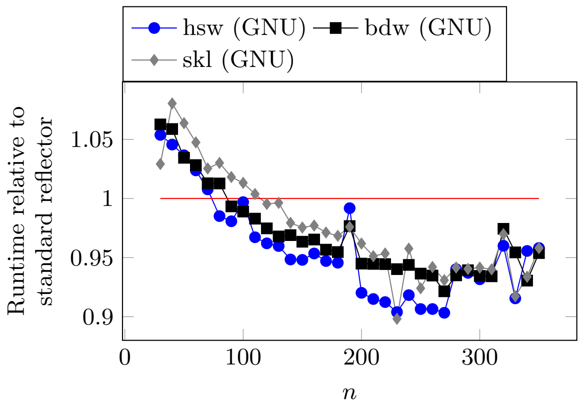


Figure 5: Impact of the reflector choice when chasing $\lfloor n/6 \rfloor$ tightly packed bulges as in Algorithm 1. The graph shows the fraction runtime with uniform reflector / runtime using standard reflector.

In contrast to the generic LAPACK implementation `DLARFG`, we hardcode the constant size of the reflector vectors in our implementation. The generic LAPACK implementation works in-place and uses the memory locations of the input vector to return the reflector vector. We, by contrast, explicitly overwrite the input vector and require additional memory to return the reflector vector. Moreover, we omit error checking because the first column of the bulge is known to be a non-zero column. As a consequence, the implementation is shorter and inlined by the compiler, but also less robust.

The compiler chooses scalar instructions to implement either reflector. As we can reduce the first column of multiple bulges at the same time, we vectorise the computation of both reflector types by using AVX instructions. The usage of AVX allows the computation of up to four reflectors at a time. In summary, we compare the following four implementations: (1) standard reflector with scalar instructions, (2) uniform reflector with scalar instructions, (3) standard reflector with vector instructions and (4) uniform reflector with vector instructions.

Table 1 shows a runtime comparison of the four implementations. The benefit of using AVX instructions depends on the platform. On Haswell and Skylake the vectorised versions are equally good or faster; on Broadwell the scalar implementation is consistently faster. The runtime results can be traced to the throughput respectively the latency of the packed respectively scalar division and squareroot instruction [4].

The choice of the reflector is apparently a hardware-dependent one. Regarding the bulge-chasing kernel, this results in two decisions. When should the standard reflector be used and when the uniform reflector? In what cases does the vectorised implementation give better results?

There is a trade-off between (1) the more expensive uniform reflector and cheaper matrix updates and (2) the cheaper standard reflector and more expensive matrix updates. The results of this trade-off are shown in Figure 5. The plot takes the runtime using the standard reflector as reference. For small matrices up to $n = 60$, the standard reflector is faster than the uniform reflector. For bigger matrices, it pays off to save multiplications and the uniform reflector is faster. Since we consider 100-by-100 the minimum size of the computational window, we conclude that the uniform reflector is the preferred choice on all of our test platforms.

The decision whether the scalar or the vectorised uniform reflector is the better choice depends on the underlying hardware. On Haswell and Skylake it is advantageous to use the vectorised version. On Haswell, the performance difference even suggests to use the vectorised implementation when only a batch of three reflectors (with the fourth SIMD entry left unused) is computed. On Broadwell, by contrast, the scalar implementation is consistently the best option.

5. OPTIMISING THE MATRIX UPDATES

When a bulge is moved one step along the diagonal, an update $\mathbf{H} \leftarrow \mathbf{Q}_j^T \mathbf{H} \mathbf{Q}_j$ is executed in the form of two updates. We call $\mathbf{H} \leftarrow \mathbf{H}\mathbf{Q}_j$ the right update and $\mathbf{H} \leftarrow \mathbf{Q}_j^T \mathbf{H}$ the left update. Further, we accumulate the reflectors into the similarity transform matrix $\mathbf{Q} \leftarrow \mathbf{Q}\mathbf{Q}_j$. The runtime of the bulge-chasing kernel is dominated by these matrix updates. We therefore compare different choices to select the fastest possible implementation.

We define two test problems on a 100-by-100 computational window with leading dimension 104, which we consider a lower bound for the size of the computational window in a bigger bulge-chasing kernel. Test 1 chases one bulge from the top-left corner to the bottom right corner. Test 2 chases four tightly packed bulges. We expect this test to give meaningful results to evaluate the effectiveness of our optimisations because (a) the computational intensity increases when we chase a chain of bulges and (b) our optimisations will be increasingly effective for bigger window sizes until a memory requirement slightly greater than the L2 cache capacity is exceeded. Our kernel is cache-friendly because it exhibits temporal locality due to overlapping updates, spatial locality for the right update and a predictable memory access pattern for the left update the stride prefetcher can cope with. However, the benefit of overlapping updates diminishes with larger computational windows; large updates no longer benefit from temporal locality.

Inspired by work of Goto and van de Geijn [3] for `DGEMM`, we distinguish between three layers of abstraction. We refer to the three matrix updates as *macro kernels*. We split the macro operations up into smaller structures, so-called *micro kernels*, which are chosen in accordance with the underlying architecture. The *instruction level* chooses the hardware instructions necessary to realise the micro kernel.

Tag	**Implementation**	**Reflector**	**Ratio**
A	LAPACK-style	standard	13
B	Figure 6	standard	52
C	Figure 6, unrolled twice	standard	104
D	LAPACK-style	uniform	12
E	Figure 6	uniform	48
F	Figure 6, unrolled twice	uniform	96

Table 2: Implementations of $\mathbf{H} \leftarrow \mathbf{H}\mathbf{Q}_j$. The right column gives flops/iteration and indicates the degree of instruction-level parallelism.

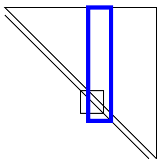
5.1 Right Update of the Hessenberg Matrix

Each right update, $\mathbf{H} \leftarrow \mathbf{H}\mathbf{Q}_j$, operates on a tall-and-skinny n-by-3 submatrix. The implementations can benefit from consecutive memory accesses and alignment. A full list of all tested implementations is given in Table 2.

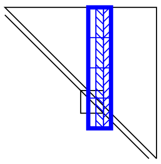
The compiler-optimised LAPACK-style update, denoted by variants A and D, serves as our reference implementation and is illustrated in Figure 3. As the row count is a runtime parameter and can attain any value, the compiler provides two versions. The first version unrolls the dot products using scalar fused-multiply-add instructions and directly resembles the row-wise dot products. The second version vectorises over the rows and uses packed fused-multiply-add instructions. The routine is inlined across all test platforms.

Our implementations are very close to the compiler-generated instruction sequence. We harness problem-specific knowledge to improve them further. Figure 6 explains our implementation. Aiming for an in-place update, the submatrix is split into micro kernels, here 4-by-3 blocks. The block height is chosen in accordance with the vector width, which is 4 on our test platforms. At instruction level, the columns of each 4-by-3 block are scaled with the appropriate broadcast entry of $\mathbf{v}$. A register is used to accumulate the intermediate result $\mathbf{Hv}$. This register is multiplied with τ if the standard reflector is used. The multiplication with $\mathbf{v}^T$ remains. We reuse the broadcast entries of $\mathbf{v}$, multiply with the register holding $\tau\mathbf{Hv}$ and update the respective columns in the 4-by-3 block. Note that the bottommost block in the panel may not coincide with the row count. This is the reason why the compiler is forced to provide two implementations, one with scalar instructions and one with vector instructions. We, by contrast, avoid this case distinction and instead round up the row count to the next multiple of 4. Recall that the computational window is copied to a compact memory representation. As a consequence, the rounding is a safe step to take because the window is easily sufficiently padded and the additional flops are guaranteed to be zero flops. We abbreviate these implementations as variants B and E, respectively.

The performance of this implementation is hampered by limited instruction-level parallelism. We apply loop unrolling in order to increase the performance. The first approach extends the micro kernel and process two consecutive 4-by-3

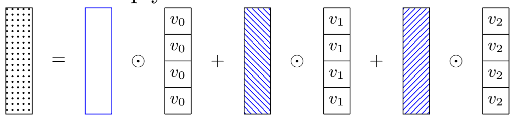


(a) Macro kernel.



(b) Decomposition into micro kernels.

```
function RightUpdate($\mathbf{H} \in \mathbb{R}^{n\times 3}$, $\tau \in \mathbb{R}$, $\mathbf{v} \in \mathbb{R}^{3\times 1}$)
    Broadcast entries of $\mathbf{v} \equiv (v_0, v_1, v_2)$ and store them in registers
    Broadcast $\tau$ and store in register
    for each block ▨ in H do
        Load columns into registers
        Fused multiply-add
```



```
        Scale with $\tau$
        Update block in-place with fused multiply-add
```

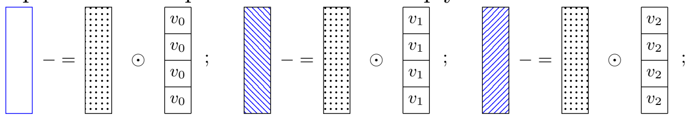


Figure 6: Layered view of the implementation of $\mathbf{H} \leftarrow \mathbf{H}(\mathbf{I} - \tau\mathbf{v}\mathbf{v}^T)$.

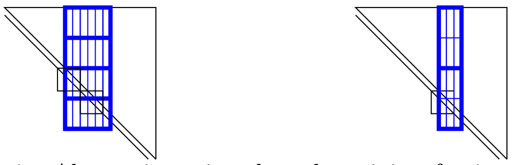


Figure 7: Alternative micro kernels striving for improved instruction-level parallelism.

blocks at a time (variants C, F), see Figure 7 (right). Although this increases the number of zero flops (up to 7 rows can be unnecessary computation), we observe speedup. This approach requires additional zero-padding of the computational window in order to safely process the final move of the bottommost bulge. We pad the computational window with four zero rows. Again, this step is feasible in a bigger bulge-chasing kernel because we copy the computational window to a more compact memory representation to avoid huge column strides.

The second approach to increase the instruction count per iteration is to unroll panel-wise. Instead of one panel, multiple panels can be updated simultaneously, see Figure 7 (left). We abandon this idea because the performance gain from grouped bulge-chasing outweighs the speedup obtainable through panel-wise loop unrolling for simultaneous bulge-chasing.

Figure 8 and Table 3 display the performance results using Test 1. Note the amount of zero flops, which is introduced due to rounding up to the next multiple of four. In general it is beneficial to use the uniform reflector. Furthermore, we observe speedup of our manually optimised implementations over the compiler-optimised implementation across all platforms. The version with manual loop unrolling performs best in all cases. It is especially effective on the Skylake platform with a speedup of 1.48 (standard reflector) respectively 1.67 (uniform reflector). On Haswell and Broadwell we achieve a performance improvement between 1.10 and 1.29. We attribute the observed speedup to the cases where we trade two or three scalar instructions for one vector instruction (with zero flops), sufficient instruction-level parallelism and less branching. While in general the uniform reflector

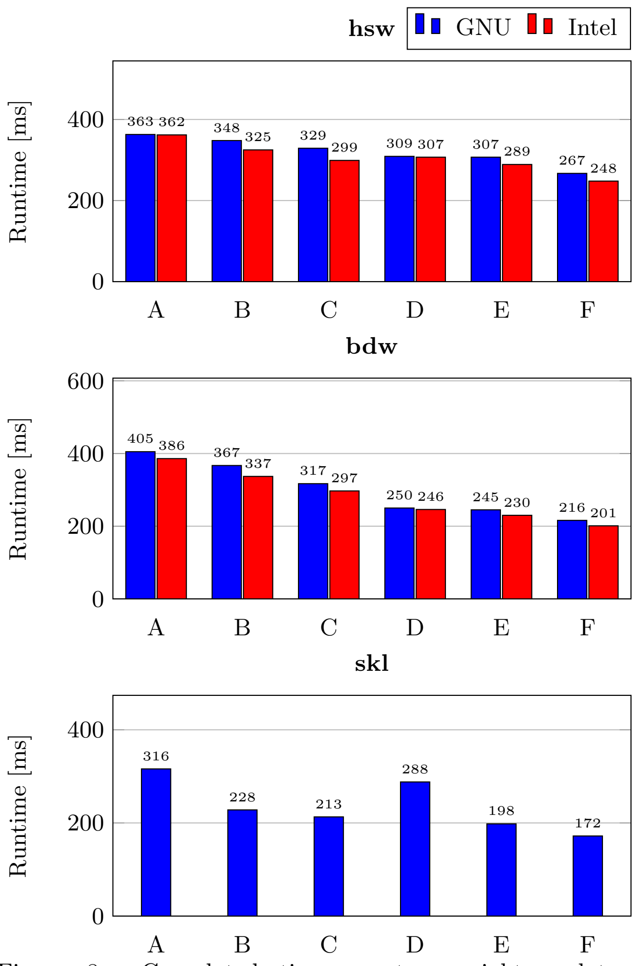
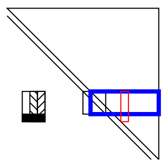


Figure 8: Cumulated time spent on right updates, $\mathbf{H} \leftarrow \mathbf{H}\mathbf{Q}_j$, needed to chase 1 bulge 100000 times in a 100×100 window.

outperforms the standard reflector, we observe a significant performance discrepancy on Broadwell. The additional parameter τ present only in the standard reflector seems to incur a performance penalty.

		hsw		bdw		skl
Tag	Flops	GNU	Intel	GNU	Intel	GNU
A	64220 (64220)	17.69	17.74	15.85	16.64	20.32
B	66092 (64220)	18.99	20.34	18.01	19.61	28.99
C	68536 (64220)	20.83	22.92	21.62	23.08	32.18
D	59280 (59280)	19.18	19.31	23.71	24.10	20.58
E	61008 (59280)	19.87	21.11	24.90	26.53	30.812
F	63264 (59280)	23.69	25.51	29.29	31.47	36.78

Table 3: Performance results for $\mathbf{H} \leftarrow \mathbf{H}\mathbf{Q}_j$ in Gflops/s as measured for Figure 8. The flop count reports the number of total flops; the number in brackets counts useful flops excluding zero flops.



```
function LeftUpdate($\mathbf{H} \in \mathbb{R}^{3\times n}$, $\tau \in \mathbb{R}$, $\mathbf{v} \in \mathbb{R}^{3\times 1}$)
    Construct reflector matrix ▨ $\equiv \tau \mathbf{v}\mathbf{v}^T$ and store columns in registers
    for each column ▯ $\equiv (H_{0j}, H_{1j}, H_{2j}, *)^T$ in $\mathbf{H}$ do
        Broadcast column entries $H_{0j}$, $H_{1j}$, $H_{2j}$
        Update column in-place with fused multiply-add
```

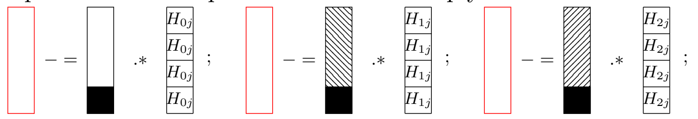


Figure 9: Implementation of $\mathbf{H} \leftarrow (\mathbf{I} - \tau \mathbf{v}\mathbf{v}^T) \cdot \mathbf{H}$

5.2 Left Update of the Hessenberg Matrix

The left update, $\mathbf{H} \leftarrow \mathbf{Q}_j^T \mathbf{H}$, applies the reflector to a 3-by-n submatrix. The compiler-optimised LAPACK-style update is based on dot products, as illustrated in Figure 4. Left updates face the following performance penalties.

- **Non-SIMD width row count.** As we use column major memory layout, the number of consecutive memory entries is limited to 3.
- **Strides.** The memory locations that have to be updated are far away from each other.
- **Misaligned memory accesses.** The base pointer of the 3-by-n submatrix may or may not be aligned. Columns can be expected to be split across cache lines.

The performance of the left update on the test problems is typically more than a factor of 2 worse than the right update. On Broadwell, the performance discrepancy is even close to a factor of 3. The compiler fails to vectorise the LAPACK-style update.

As stated in [3], the key to fast updates is harnessing consecutive memory accesses. Due to the column major memory layout, the left update has strided accesses and, as a consequence, we cannot expect the left update to reach the same performance level as the right update. We can, however, implement optimisations that are beyond the scope of the compiler.

We manually vectorise the left update. Figure 9 explains our implementation. Although only a 3-by-n submatrix is updated, a 4-by-n band is read and written to. The fourth entry of each SIMD instruction yields a zero flop. For this purpose, we explicitly construct the matrix $\mathbf{v}\mathbf{v}^T$. We pad the 3-by-3 matrix with zero entries and store the columns in registers

Tag	Implementation	Reflector	Ratio
A	LAPACK-style	standard	13
B	Figure 9	standard	24
C	LAPACK-style	uniform	12
D	Figure 9	uniform	24
E	Figure 9	uniform	48
F	Figure 9	uniform	72
G	Figure 9	uniform	96

Table 4: Implementations of single-column left updates. The right column gives flops/iteration and indicates the degree of instruction-level parallelism.

$$\begin{pmatrix} v_0v_0 & v_1v_0 & v_2v_0 \\ v_0v_1 & v_1v_1 & v_2v_1 \\ v_0v_2 & v_1v_2 & v_2v_2 \\ 0 & 0 & 0 \end{pmatrix}.$$

The explicit zero-padding allows us to avoid mask-based instructions with corresponding latencies for vector loads and stores. In the case of the standard reflector, all matrix entries are additionally scaled with τ.

We process the matrix update columnwise in the form of matrix-vector multiplies $\mathbf{H}_{:,j} - (\mathbf{v}\mathbf{v}^T)\mathbf{H}_{:,j}$. We load the 4-by-1 vector $\mathbf{H}_{:,j}$ into a register and additionally broadcast its first three entries into registers. We multiply the i-th column of the reflector matrix with the broadcast i-th entry of $\mathbf{H}_{:,j}$ and update the register holding $\mathbf{H}_{:,j}$. The zero-padded structure of the reflector matrix guarantees that the fourth vector entry of $\mathbf{H}_{:,j}$ is not changed. Consequently, 1/4th of the flops are zero flops.

We have compared the following variants for the left update, see Table 4. Variants A and C are the compiler-optimised

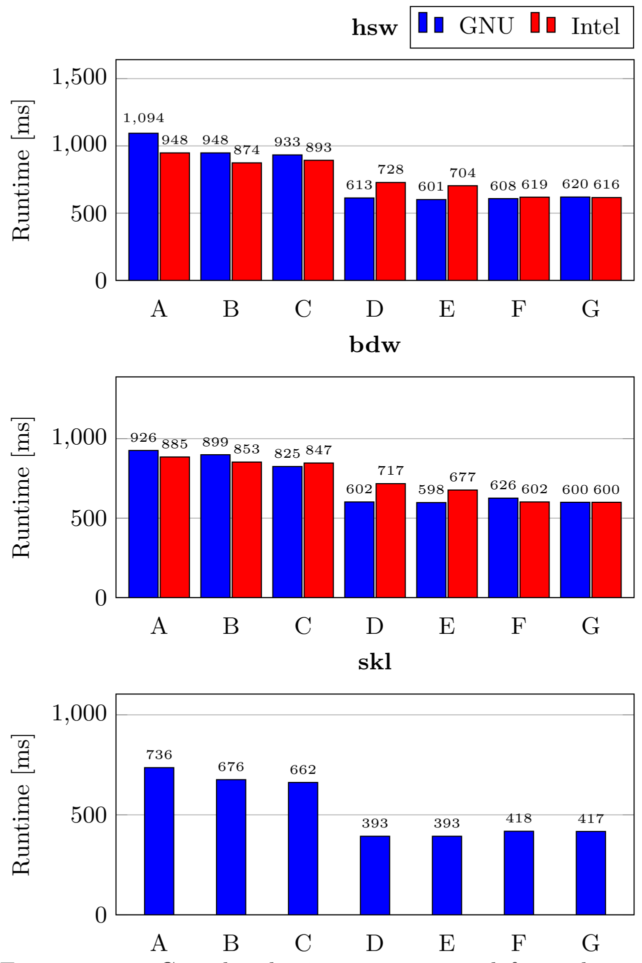


Figure 10: Cumulated time spent on left updates, $\mathbf{H} \leftarrow \mathbf{Q}_j^T\mathbf{H}$, needed to chase 1 bulge 100000 times in a 100×100 window.

LAPACK-style updates; the optimised variants B and D implement the approach of Figure 9 with intrinsics for either reflector.

Analogously to the right update, we apply manual loop unrolling aiming for improved instruction level parallelism. We restrict our analysis to the uniform reflector. Variant E, F, and G process 2, 3 or 4 columns per loop iteration.

Figure 10 lists the runtime results. The vectorised version (D) is between 1.37 and 1.72 times faster than the LAPACK-style updates (C). The performance more than doubles, see Table 5. We do not observe any noteworthy benefit from additional loop unrolling.

The combination of using the uniform reflector and manual vectorisation (D) yields a performance improvement in the range of 1.54 to 1.87 over the standard LAPACK-style update (A). As such, the performance gap between left updates and right updates persists.

Motivated by the idea of grouped bulge chasing mentioned in Section 2, we test a specialised routine for a joint update of four bulges. Instead of doing four calls to LeftUpdate

	hsw		**bdw**		**skl**
Tag	**GNU**	**Intel**	**GNU**	**Intel**	**GNU**
A	5.87	6.77	6.94	7.26	8.72
B	12.75	13.83	13.45	14.17	17.89
C	6.35	6.64	7.19	7.00	8.96
D	19.51	16.45	19.90	16.70	30.45
E	19.91	17.00	20.00	17.69	30.42
F	19.70	19.33	19.12	19.88	28.66
G	19.31	19.43	19.96	19.94	28.68

Table 5: Performance results in Gflops/s as measured for Figure 10.

	hsw		**bdw**		**skl**
Tag	**GNU**	**Intel**	**GNU**	**Intel**	**GNU**
4 * D	84.08	95.13	73.22	83.99	55.96
joint	80.85	78.94	68.70	68.50	53.8209

Table 6: Cumulated runtime in ms spent on left updates needed to chase four bulges 4000 times in a 100×100 window.

one after another, we update a 12-by-n subwindow. Analogously to Figure 9, we load and store 4-by-1 vectors rather than 3-by-1 vectors. Consequently, again 1/4th of the flops are zero flops. In the 12-by-n window there are always two non-overlapping 4-by-1 vectors available, whose processing order does not matter. We harness this and attempt to maximise the instruction-level parallelism while preserving the correct read-and-write order of overlapping entries. We notice that the efficiency of our implementation is highly susceptible to the order of the instructions and, to a lesser degree, to the choice of the compiler. Table 6 lists our runtime measurements. The joint updates achieve a small performance improvement over four successive calls to version D, the best implementation of LeftUpdate. More importantly, we mitigate the performance gap between the Intel and the GNU compiler and now have an implementation which consistently performs well on all of our test platforms independent of the choice of the compiler.

5.3 Updates to the Q matrix

The $\mathbf{Q}$ matrix accumulates all reflectors used within the computational window. This allows us to use $\mathbf{Q}$ to later on update off-diagonal windows with an efficient `DGEMM` operation. In our kernels we initialise $\mathbf{Q}$ with the identity matrix and successively apply all reflector vectors in the order they arise. Due to the favourable memory access pattern, we follow the accumulation scheme of `DLAQR5` and multiply all reflectors from the right $\mathbf{Q} = \mathbf{Q}_1\mathbf{Q}_2\mathbf{Q}_3 \ldots$.

While the reference bulge-chasing routine uses the identical LAPACK-style right update, $\mathbf{H} \leftarrow \mathbf{H}\mathbf{Q}_j$, discussed in Subsection 5.1, we cannot reuse the intrinsics implementation. As we initialise $\mathbf{Q}$ with the identity matrix and obtain fill-in through successive application of the reflector vectors, the update range may start at any, possibly unaligned address. Consequently, our implementation has to rely on unaligned loads and stores. Otherwise we use the same set of techniques discussed previously. In particular we trade zero flops for SIMD instructions and apply manual loop unrolling. Again, the manually unrolled version performs best. We observe, however, that the efficacy of these techniques falls behind the one of $\mathbf{H} \leftarrow \mathbf{H}\mathbf{Q}_j$. We attribute this to the performance penalty arising from unaligned instructions and the fact that due to the sparsity pattern in general less flops are necessary.

As an alternative, the one-sided accumulation order allows for various accumulation orders as long as the order of the reflectors is preserved. We bracket $\mathbf{Q} = (\mathbf{Q}_1\mathbf{Q}_2)(\mathbf{Q}_3\mathbf{Q}_4)\dots$ and construct 4-by-4 reflector matrices explicitly. Given two reflector vectors $\mathbf{v}_1^T = (*, *, *, 0)$ and $\mathbf{v}_2^T = (0, *, *, *)$, we obtain

$$\begin{aligned}\mathbf{Q}_1\mathbf{Q}_2 &= (\mathbf{I} - \mathbf{v}_1\mathbf{v}_1^T)(\mathbf{I} - \mathbf{v}_2\mathbf{v}_2^T) \\ &= \mathbf{I} - \mathbf{v}_1\mathbf{v}_1^T - \mathbf{v}_2\mathbf{v}_2^T + \mathbf{v}_1\mathbf{v}_1^T\mathbf{v}_2\mathbf{v}_2^T.\end{aligned}$$

This scheme allows us to merge two successive moves of one bulge. In our performance measurements we observe that the bracketed scheme yields an increase in flops, which goes along with an increased runtime. For our optimised routine we therefore opt for the unrolled single-bulge update.

Consider, for example, a 4-by-4 matrix $\mathbf{Q}$. If $\mathbf{Q}$ is updated with two 3-by-3 reflectors, i.e., $\mathbf{Q} \leftarrow (\mathbf{Q}\mathbf{Q}_1)\mathbf{Q}_2$, 96 flops are necessary. Here, we use the uniform reflector and implement a row update with two dot products requiring 12 flops per row (6 additions, 6 multiplications). The first reflector $\mathbf{Q}_1$ modifies the three leftmost columns of $\mathbf{Q}$ and yields 4 rows · 12 flops/row = 48 flops. Then $\mathbf{Q}_2$ requires an equal amount of 48 flops to modify the three rightmost columns of $\mathbf{Q}$. Together, the flops add up to 96 flops. If, by contrast, $\mathbf{Q}$ is updated as in $\mathbf{Q} \leftarrow \mathbf{Q}(\mathbf{Q}_1\mathbf{Q}_2)$, then 128 flops are needed. Neglecting the cost of constructing the 4-by-4 reflector, 32 flops are executed per row (two dot products, each with 8 multiplications and 8 additions). With 4 rows, the total amount of flops equals 128 flops.

5.4 Benefit of Compile-time Parameters

Parameters such as the matrix size, the leading dimension or the number of bulges can be regarded as compile-time constants. The advantages are twofold. First, it allows the compiler to propagate constants, which is especially effective because we activate link-time optimisation (GNU) respectively interprocedural optimisation (Intel). Second, the compiler can deduce that our column updates operate on non-overlapping memory locations. As a result, further optimisations become available, or, at the very least, less case distinctions have to be made.

We have experimentally passed n and n_b as template parameters to our bulge-chasing routine. Further, we declared the leading dimension as a compile-time constant. In other words, all three parameters were known at compile-time. We have actively used the compile-time constants to compute the iteration counters and initialise the update range counters n_H and m_H. Furthermore, the implementation of RightUpdate can benefit from the additional knowledge that aliasing cannot occur. This update affects three consecutive columns. Consider for example the LAPACK-style update. The evaluation of the scalar $s = \tau(H_{i,j}v_0 + H_{i,j+1}v_1 + H_{i,j+2}v_2)$ computes the memory locations as `H[i+j*ldH]`, `H[i+(j+1)*ldH]` and `H[i+(j+2)*ldH]`. If the leading dimension is not known at compile-time, virtually any memory access pattern can occur. Even if we exclude negative values, an update could address overlapping memory locations if `ldH` $< n$. Informing the compiler about the leading dimension is therefore another way to pass the knowledge that the memory locations to be updated logically correspond to disjoint columns.

The usage of template parameters has boosted the performance of our kernel by 2 to 4 Gflops/s. On Haswell, for example, we increased the maximum performance from 17.03 Gflops/s to 18.86 Gflops/s. On Skylake, the gain is bigger with an increase from 26.5 Gflops/s to 30.19 Gflops/s. For the time being we refrain from compile-time computations because it makes the kernel inflexible for using in a bigger bulge-chasing kernel. We think, however, that it is possible to benefit from templated functions in a task-based context when the partitioning of a big matrix can be computed at compile time. The performance results we present in the next section do not make use of template parameters.

5.5 Kernel Performance Comparisons

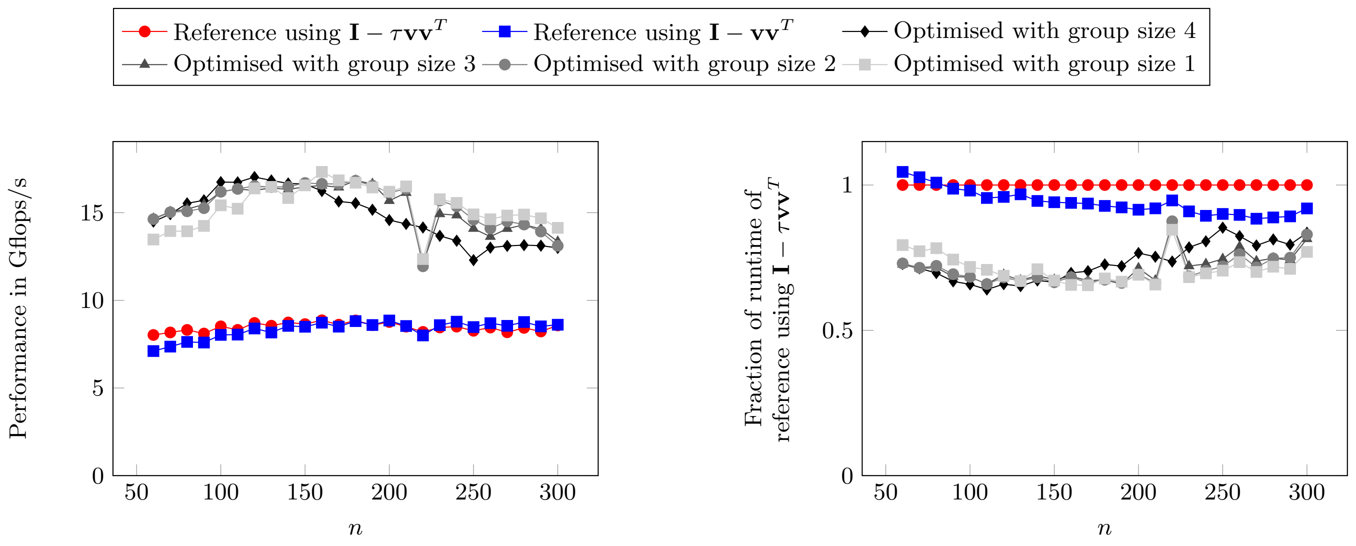
Given several optimised routines for the computation of the reflector and the matrix updates, we merge these routines now to obtain an optimised bulge-chasing mechanism within the computational window.

We compare a total of six bulge-chasing kernels. On the one hand, we have one reference version for each reflector type. On the other hand, we use optimised routines and experiment with different group sizes. We configure the optimised kernels as follows.

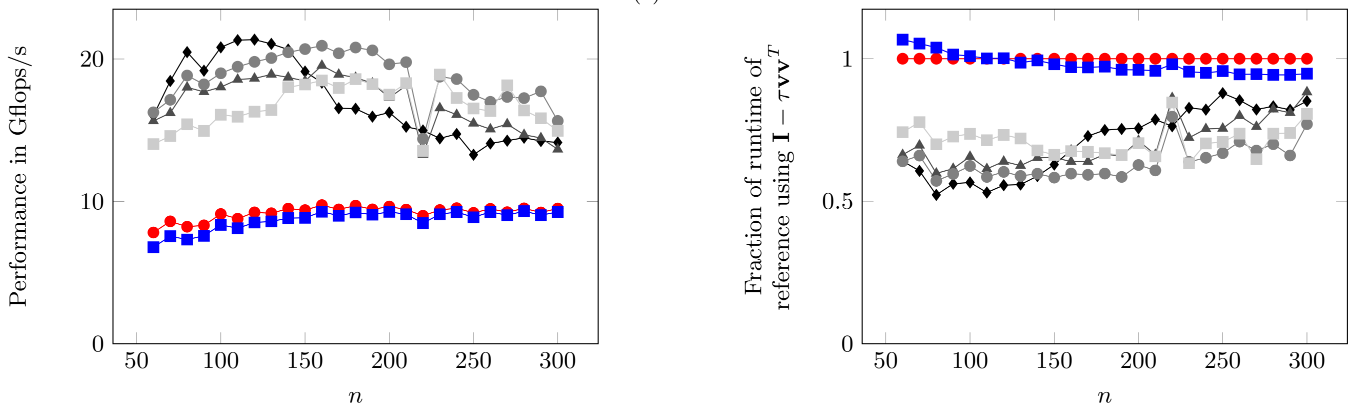
- Dlarfg. We choose the uniform reflector across all platforms. On Broadwell, we solely use the scalar implementation; on Haswell and Skylake we prefer the SIMD implementation for a group size of four and otherwise use the scalar version.
- UpdateFromRight. We choose variant F, which is consistently the fastest implementation.
- UpdateFromLeft. The choice of the implementation is delicate because results suggest that it should be a compiler-dependent one. We base this case study on the GNU compiler and therefore opt for variant D. For a group size of 4 we make use of the specialised routine for jointly executing 4 updates.

We choose the leading dimension as $(4 \cdot \lceil n/4 \rceil) + 4$, where n is the matrix size. In other words, we round up to the nearest multiple of four and add an additional padding of four. If this number is a power of two, we revise the leading dimension by an additional pad of 4. This choice satisfies the desired alignment properties and the demand for the additional zero-padding.

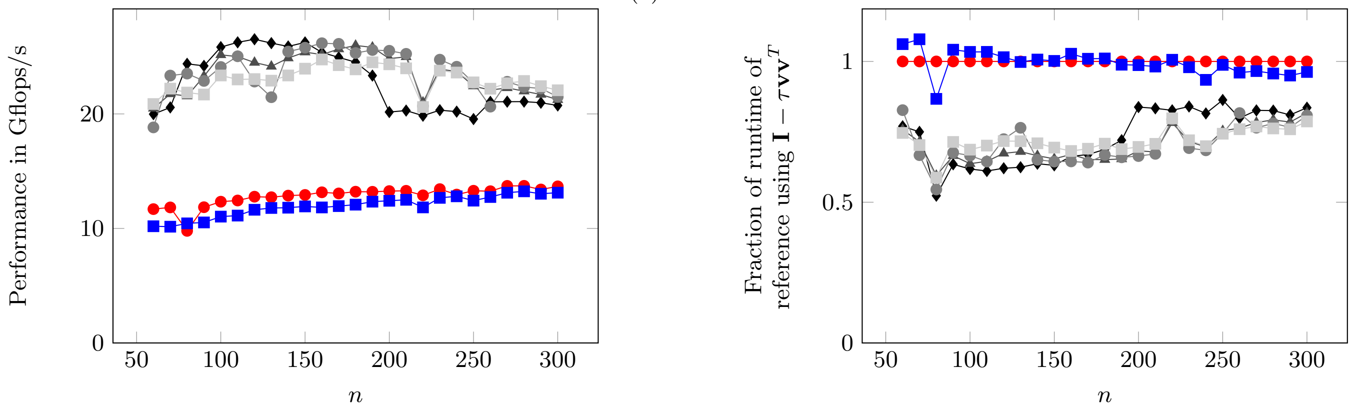
Figure 12 shows the results. On all test platforms the optimised bulge-chasing kernel reaches approximately 50% of the peak performance for matrix sizes between 80 and 200 and more than doubles the performance of the reference code. The increase in performance is also visible in the runtime. We obtain a speedup of a factor of 1.7 to 1.9 over the fastest



(a) hsw



(b) bdw



(c) skl

Figure 12: Performance and time-to-solution measurements for the unblocked bulge-chasing routine.

reference implementation. Changing the leading dimension from 224 to 228 eliminates the performance drop at $n = 220$, which affects three of the optimised implementations and both reference versions.

The impact of the group size depends on the matrix size. For matrices $n < 150$ chasing bulges in groups is advantageous. A group size of 4 reduces the time-to-solution by up to 15% compared to a group size of 1. Increasing the group size to up to 8 improves the performance further for small-sized matrices with $n < 80$. For matrices greater than 150, it becomes advantageous to reduce the group size to 2 or 1. We attribute this to cache effects. For a group size of 4, we observe a drop in performance when the L2 cache size is hit. For larger matrices, the bulge-chasing kernel in groups of size 1 or 2 likely benefits from better temporal locality due to overlapping rows and columns in successive moves. This effect diminishes for bigger matrix sizes and the performance and runtime results of all variants converge. The runtime improvement over the reference code shrinks accordingly.

In conclusion, the choice of the group size allows us to configure the bulge-chasing kernel to sustain close to 50% of the peak performance for $n \in [80, 200]$. The runtime improvement ranges in 1.5 to 1.9. Though the choice is best left to an offline-tuning procedure, we provide the following default values. For $n \leq 150$, we configure the bulge-chasing kernel with $gs = 4$. For $n \in (150, 200]$ we use $gs = 2$. For $n > 200$ we observe a performance drop, which we address next. We identify the following hot spots.

- **Uneven load**. The left update is disproportionally expensive. About 60% of the runtime is spent in left updates, 19% in right updates and 18% in the accumulation of the $\mathbf{Q}$ matrix. The performance penalties persist and, if anything, worsen for bigger matrix sizes because the stride grows.
- **Memory moves**. The positive effect of interleaving and overlapping updates diminishes with increasing matrix size. We observe an increased amount of memory moves, which we attribute to an increased amount of cache misses.
- **False sharing.** Due to the memory access pattern for left updates, the performance suffers from a lot of false sharing. This exacerbates the memory traffic problem.

We implement a classic blocking scheme to sustain the performance for bigger matrix sizes and to reduce the impact of the left update on the overall performance.

6. FULLY BLOCKED BULGE CHASING

For bigger matrices, we consider the setup of Figure 13. The bulge-chasing kernel is applied to a computational window; the reflectors are accumulated in the similarity transform matrix $\mathbf{Q}$. The matrix panels outside of the window are updated with matrix-matrix multiplies $\mathbf{H}_t \leftarrow \mathbf{H}_t\mathbf{Q}$ and $\mathbf{H}_r \leftarrow \mathbf{Q}^T\mathbf{H}_r$.

The performance of the bulge-chasing kernel is tolerant towards the choice of the window size in $\{80, 81, \ldots, 200\}$.

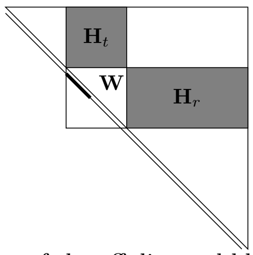


Figure 13: Update of the off-diagonal blocks with DGEMM.

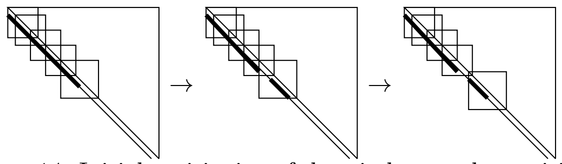


Figure 14: Initial positioning of the windows and repositioning of the bottommost window after the first move.

Blocking allows us to provide an implementation which sustains performance for any matrix size. The focus of the off-diagonal updates is therefore on (a) finding an advantageous window positioning, (b) harnessing the structure of the similarity transform matrix in the matrix-matrix multiplies and (c) tuning the kernel w.r.t. the window size.

6.1 Window Positioning

We define the computational windows subject to two constraints. First, the base pointer of each window is aligned to a 32-byte aligned boundary. Second, the window size ensures that after each bulge-chase in a window the next window meets the alignment constraint as well. This allows us to avoid performance penalties from loads and stores that cut across cache lines boundaries and reduce false sharing.

Assume that a n-by-n window is positioned at an aligned address (p, p). The base address of the window in the next iteration is given by the position of the top bulge after the execution of the bulge-chasing kernel. As each of the n_b bulges is moved by $n - 4 - 3(n_b - 1)$ positions, the address $p + n - 3n_b - 1$ must be aligned.

Consider for example a window with $n_b = 20$ bulges spanning 60 entries. Let the window be positioned at an aligned address (p, p). We seek a window size n such that the window in the next iteration is aligned as well. As the bulges move along the *sub*diagonal, the new window is moved by $n - 60 - 1$ entries and starts at $(p + n - 61, p + n - 61)$. This becomes an aligned address for $n = 121$, which also satisfies $\lfloor n/6 \rfloor \approx n_b$.

Our implementation positions the windows from bottom to top. Figure 14 illustrates the initial window positioning. Beginning with the bottommost window, we position a n-by-n candidate window such that it is filled half-way. The base pointer of this window may or may not be aligned. In the latter case we extend the window to the top by adding bulges. Due to the 3-by-3 structure of the bulges, it takes at most 3 bulges until an aligned address has been reached. Next, we increment the window size until the window size guarantees that the windows of all future iterations are aligned. As we align to 32-byte boundaries, at most 3 increments are nec-

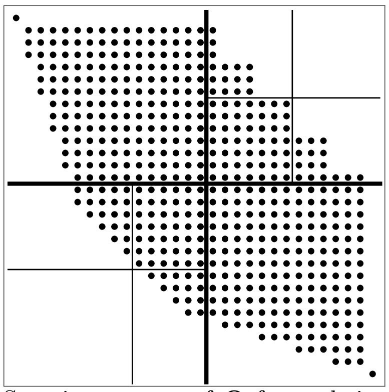


Figure 15: Sparsity pattern of $\mathbf{Q}$ for a chain of 5 tightly packed bulges in a 30-by-30 computational window.

essary. The bottommost window is therefore typically more than half-way filled.

Reconsider the 121-by-121 example window from above. Assume (p, p) is not an aligned address. Then adding bulges to the top changes the start address to any of $(p-3, p-3)$, $(p-6, p-6)$, $(p-9, p-9)$. It is guaranteed that one these choices is an aligned address. Instead of 20 bulges, the window contains 21, 22 or 23 bulges. As a consequence, the window exhibits a window size of 124, 127 or 130 and is overfilled.

We proceed with the tessellation with n-by-n windows and adapt the start address and the window size analogously to the bottommost window. The topmost window is automatically aligned because it coincides with the matrix boundaries. Its window size is extended analogously with the other windows. Our implementation always chooses the window size as a multiple of 6 in order to avoid additional adjustments needed to never cut through bulges.

Due to the uneven degree of filling, the windows are chased one after another rather than simultaneously. The base window moves the bulges by $n-4-3(n_b-1)$ positions; an overfilled bottommost window moves them by less positions. Simultaneous moves would lead to windows overtaking the bottommost window. Figure 14 illustrates this situation. The overfilled bottommost window after repositioning overlaps with the window second from bottom. This window must not be processed until the bottommost window has been moved because bulges may never overlap.

6.2 Off-diagonal Updates

As the bulge-chasing kernel initialises the transformation matrix with the identity matrix, the resulting matrix exhibits a certain sparsity pattern. An example is depicted in Figure 15. Braman, Byers and Mathias [1] discuss how the sparsity pattern of $\mathbf{Q}$ can be harnessed to update off-diagonal blocks. The LAPACK routine `DLAQR5` implements two of their proposals. The first option disregards the sparsity pattern and updates off-diagonal blocks with one call to `DGEMM`. For a half-way filled Hessenberg matrix this yields approximately 38% zero flops. The second option partitions

$$\mathbf{Q} = \begin{pmatrix} \mathbf{Q}_{11} & \mathbf{Q}_{12} \\ \mathbf{Q}_{21} & \mathbf{Q}_{22} \end{pmatrix}$$

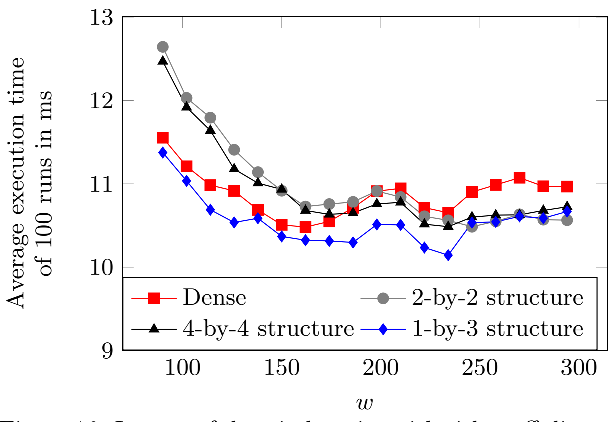


Figure 16: Impact of the window size with either off-diagonal update option for a 5000-by-5000 matrix using MKL on bdw.

where $\mathbf{Q}_{12}$ and $\mathbf{Q}_{21}$ are triangular matrices. Off-diagonal blocks are updated with calls to `DGEMM` and `DTRMM`, respectively. Reference [6] remarks that the second option may not pay off if the implementation of `DTRMM` is not well-optimised. Our experiments confirm this insight for small-sized computational windows. We therefore propose two different partitioning schemes that solely rely on `DGEMM`. The first scheme partitions the triangular blocks further and obtains the 4-by-4 block structure

$$\mathbf{Q} = \left(\begin{array}{cc|cc} \multicolumn{2}{c|}{\mathbf{Q}_{11}} & \mathbf{Q}_{13} & 0 \\ & & \mathbf{Q}_{23} & \mathbf{Q}_{24} \\ \hline \mathbf{Q}_{31} & \mathbf{Q}_{32} & \multicolumn{2}{c}{\mathbf{Q}_{44}} \\ 0 & \mathbf{Q}_{42} & & \end{array}\right).$$

Despite the triangular shape of some submatrices (see Figure 15), we apply a total of eight `DGEMM` operations. As we target computational windows ranging from 100 to 300, the `DGEMM` operations can potentially be small. As larger matrix multiplies generally perform better, the second scheme partitions horizontally into non-evenly spaced panels

$$\mathbf{Q} = \left(\begin{array}{c|cc|c} \star & \star & \star & 0 \\ \star & \star & \star & \star \\ 0 & \star & \star & \star \end{array}\right).$$

so that each `DGEMM` operates on a larger submatrix. We align the panels of this 1-by-3 partitioning with the minor grid of the 4-by-4 partitioning. Both schemes therefore save an equal amount of approximately 12.5% zero flops compared to one call to `DGEMM`.

Figure 16 compares the execution time for all four update options subject to the window size $w \in \{90, 102, \cdots, 294\}$ in steps of 12. The choice of the window size affects the size of the off-diagonal updates and, in turn, the potential saving through avoiding zero flops. A certain window size, $w \geq 150$, is necessary to achieve good performance. For small window sizes a single dense `DGEMM` is a competitive option. Splitting into smaller matrices as done in the 4-by-4 partitioning does not pay off because `DGEMM` requires a certain size in order to be an efficient operation. The 2-by-2 update falls behind, which we attribute to the performance of `DTRMM`. Bigger window sizes increase the gain from saving flops through exploiting the sparsity structure. Then the performance of the sparsity-exploiting updates is very similar. The 1-by-3 partitioning is the fastest option, but underperforms given a saving of approximately 12.5% zero flops. The speedup over a single `DGEMM` is less or equal to 5.0%. Increasing the matrix size increase the potential gain from sparsity-exploiting updates. We therefore consider the 1-by-3 update an option worth being considered for bigger bulge-chasing kernels.

For our 5000-by-5000 test matrix, we observe the best performance with $w \in \{150, 162, \cdots, 186\} \cup \{222, 234\}$. Tests with different matrix sizes and other platforms suggest that the choice of the window size requires fine tuning. An intuitive explanation arises from the window positioning scheme. The window size can be chosen such that all windows are more or less evenly filled (and not end up with the topmost window filled with only a few bulges). A half-way filled computational window minimises the total amount of flops needed to chase all bulges. We therefore add the window size and the choice of the off-diagonal update to the list of tuning parameters that should be configured in an offline-tuning procedure.

6.3 Tuning

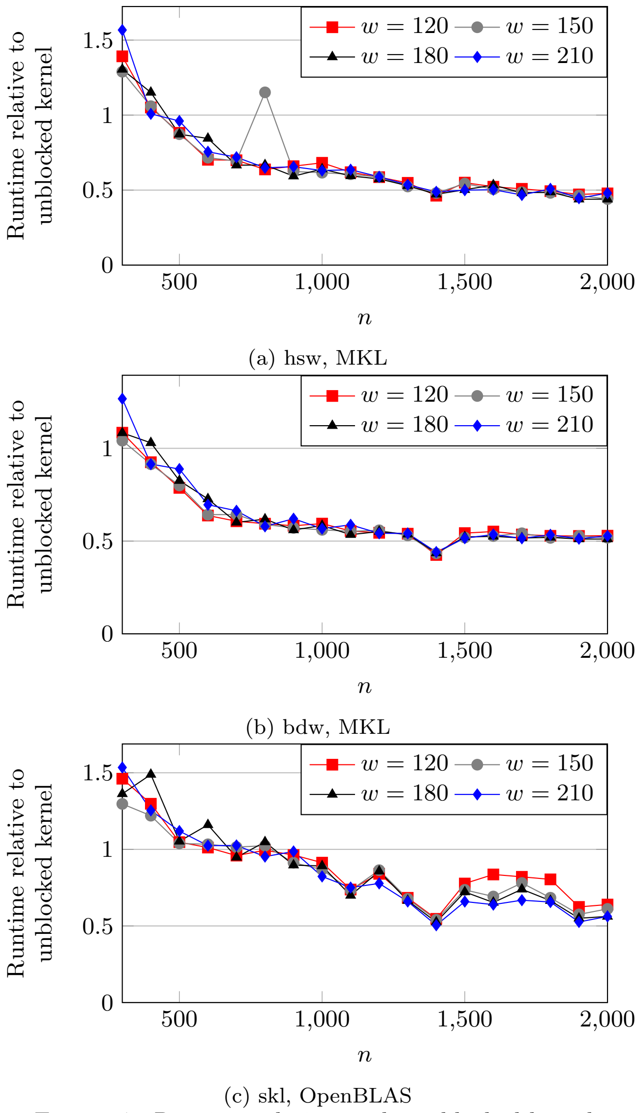
In Section 5 we have identified the window size as the key parameter to fine-tune the bulge-chasing kernel. We choose parameters such as the reflector type, the group size and routines for the matrix updates subject to the window size. Next we strive for a fully optimised bulge-chasing kernel. We restrict our analysis to small- and medium-sized matrices because for bigger matrices `DGEMM` dominates the runtime.

We compare the performance of the blocked and the unblocked scheme to determine the intersection point. Figure 17 displays the results for four different window sizes ($w \in \{120, 150, 180, 210\}$) and matrices in the range of 300 to 2000. A comparison in terms of execution time is given in Figure 18. It requires a certain matrix size to make the blocked bulge-chasing kernel pay off. The turning point depends on the platform and ranges from 300 on Broadwell to 800 on Haswell. The blocked bulge-chasing kernel levels off at 0.5 of the runtime of the unblocked kernel. The bump at $n = 1400$ results from an outlier of the unblocked bulge-chasing kernel. This wiggle can again be fixed by choosing a different leading dimension.

The impact of the window size on the performance is inconclusive. On Haswell and Broadwell the choice of the window size has little effect on the runtime, which coincides with the results from Figure 16. On Skylake, bigger window sizes generally perform better.

In summary, an offline-tuning procedure has to address the following parameters.

- **Group size.** The performance of the small-sized computational window depends on how many bulges are chased simultaneously. For different matrix sizes distinct group sizes are advantageous. The number of case distinctions, the group sizes and the determination of the intersection points are subject to tuning.
- **Window size.** The choice of the window size influences the performance within the computational window, the amount of iterations (windows) needed to chase all bulges, and the performance of the off-diagonal updates.
- **Off-diagonal update scheme.** We accumulate the Householder reflectors in the similarity transform matrix to update the matrix panels outside of the computational window. There are four schemes to execute the update.



(a) hsw, MKL

(b) bdw, MKL

(c) skl, OpenBLAS

Figure 17: Runtime relative to the unblocked kernel.

6.4 Comparison with LAPACK

Finally we compare our optimised bulge-chasing kernel with the LAPACK implementation. The routine `DLAQR5` creates and chases a chain of bulges in the multi-shift QR algorithm. As a consequence, a direct comparison with our implementation is impossible. Instead, we fall back to a derivative of `DLAQR5` in ScaLAPACK, `DLAQR6`. The latter routine has an additional *job* argument, which permits us to only chase (not create and chase) a chain of bulges. After the case distinction due to the *job* argument, the reference code of `DLAQR5` and of `DLAQR6` are identical. Note that `DLAQR6` is

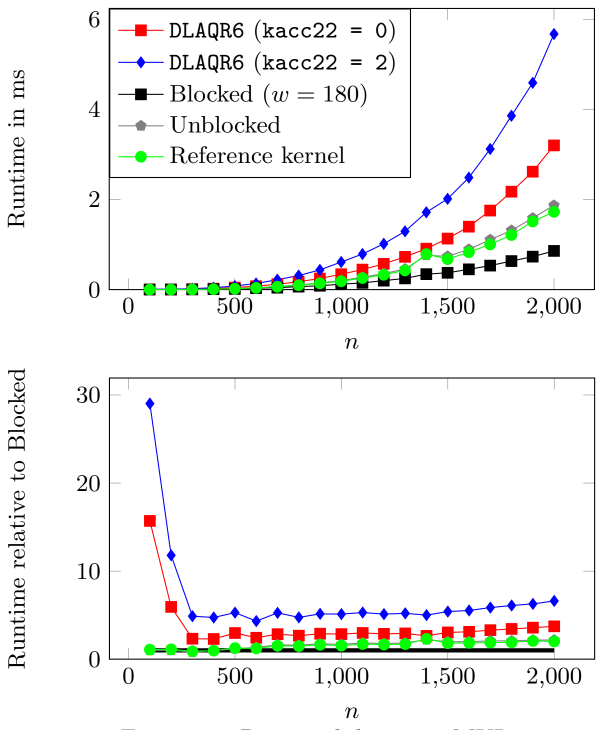


Figure 18: Race on bdw using MKL.

not an MPI code and therefore not placed at a disadvantage when run sequentially.

We configure the call to `DLAQR6` to mimic the behaviour of our bulge-chasing kernel. The number of shifts is set to $2n_b$. Furthermore, we accumulate the reflectors and use them to update far-from-diagonal blocks. Interestingly, exploiting the 2-by-2 structure of the $\mathbf{Q}$ matrix (`kacc22` $= 2$) is slightly faster than the single-step `DGEMM` update (`kacc22` $= 1$), which contradicts the preferable choice for our bulge-chasing kernel. The kernel can also be configured to not accumulate any reflectors and instead directly apply reflectors to the entire matrix (`kacc22` $= 0$). It is surprising that direct application halves the runtime. Even if the similarity transform matrix is not assumed to be the identity matrix initially, the accumulation should take at most 1/3 of the time given that three matrix updates are necessary.

Figure 18 displays the runtime of two configuration of `DLAQR6`, our final blocked implementation and, for comparison, the unblocked implementation for $n \in \{100, 200, ..., 2000\}$. Our blocked kernel is between 5 and 15 times faster than the MKL implementation of `DLAQR6` for realistic computational window sizes. When the runtime is dominated by `DGEMM` updates, the performance gap becomes narrower. Recall that `DLAQR6` includes vigilant deflation checks whose impact on the runtime is unknown.

In Section 5.5 we showed that the innermost bulge-chasing kernel can be configured in such a way that it runs at close to 50% of the peak performance (including zero flops). Since our blocked implementation relies on reasonably sized `DGEMM` updates, we claim that our kernel can sustain this performance result for any greater matrix size. It thereby qualifies as a kernel to be used in a runtime system to solve a big bulge-chasing instance in a task-based fashion.

7. CONCLUSION

We have optimised the bulge-chasing kernel for a chain of tightly packed bulges, as it arises in the multi-shift QR algorithm. Our serial implementation sustains around 50% of the peak performance for any reasonable problem size. Our kernel can be used as a building block in a task-based bulge-chasing routine. In a parallel setup the bulge-chasing kernel is on the critical path and can be regarded a sequential bottleneck. A well-tuned kernel therefore has the potential to reduce the length of the critical path. The performance of our implementation is insensitive to the matrix size. We consider this predictable runtime behaviour advantageous because it allows tuning of a parallel implementation to concentrate on other parameters.

8. ACKNOWLEDGEMENTS

We credit inspiring discussions on the topic of this paper to Carl Christian Kjelgaard Mikkelsen and Björn Adlerborn. Furthermore, we credit support for this research to Michael Bader and acknowledge the Leibniz Supercomputing Centre (LRZ) for providing compute time on the mpp2-serial Haswell cluster. Computing resources on the Broadwell nodes of Kebnekaise at High Performance Computing Center North (HPC2N) have been provided under the grant SNIC 2016/1-536.

9. REFERENCES

[1] K. Braman, R. Byers, and R. Mathias. The multishift QR algorithm. Part I: Maintaining well-focused shifts and level 3 performance. *SIAM Journal on Matrix Analysis and Applications*, 23(4):929–947, 2002.

[2] K. Braman, R. Byers, and R. Mathias. The multishift QR algorithm. Part II: Aggressive early deflation. *SIAM Journal on Matrix Analysis and Applications*, 23(4):948–973, 2002.

[3] K. Goto and R. A. van de Geijn. Anatomy of High-Performance Matrix Multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):12, 2008.

[4] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, June 2016.

[5] Intel Corporation. *Intel® Processor E5 v3 Product Family*, September 2016.

[6] L. Karlsson, B. Kågström, and E. Wadbro. Fine-Grained Bulge-Chasing Kernels for Strongly Scalable Parallel QR Algorithms. *Parallel Computing*, 40(7):271–288, 2014.

[7] L. Karlsson, D. Kressner, and B. Lang. Optimally packed chains of bulges in multishift QR algorithms. *ACM Transactions on Mathematical Software (TOMS)*, 40(2):12, 2014.

[8] D. Kressner. *Numerical Methods for General and Structured Eigenvalue Problems.* Lecture Notes in Computational Science and Engineering. Springer Berlin Heidelberg, 2006.

[9] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4):65–76, Apr. 2009.