A Study on the Influence of Caching: Sequences of Dense Linear Algebra Kernels

Elmar Peise and Paolo Bientinesi

AICES, RWTH Aachen, Germany
{peise,pauldj}@aices.rwth-aachen.de

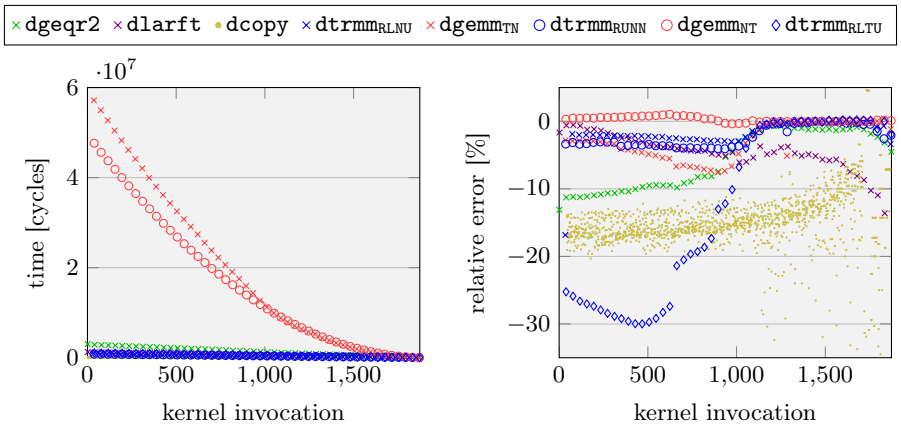
Abstract. It is universally known that caching is critical to attain high-performance implementations: In many situations, data locality (in space and time) plays a bigger role than optimizing the (number of) arithmetic floating point operations. In this paper, we show evidence that at least for linear algebra algorithms, caching is also a crucial factor for accurate performance modeling and performance prediction.

1 Introduction

In dense linear algebra (DLA), very basic yet highly tuned kernels — such as the Basic Linear Algebra Subprograms (BLAS) — are used as building blocks for high level algorithms — such as those included in the Linear Algebra PACKage (LAPACK). The objective of our research is to develop performance models for those building blocks, aiming at predicting the performance of high level algorithms avoiding entirely to execute them. In a recent article [1], we introduced a methodology for modeling and predicting performance, and showed its effectiveness in ranking different algorithmic variants performing the same target operation. However, to accurately tune algorithmic parameters such as the block-size, predictions of significantly higher precision are required. Intuitively, one would attempt to resolve this issue through more accurate performance models. Unfortunately, beyond a certain level, higher accuracy in the models of the building blocks does not translate into more precise predictions. In this paper we illustrate that such a mismatch is due to the influence of CPU caching on the performance of the compute kernels.

Several other works on the influence of caching on DLA performance exist; some notable examples are given in the following. Whaley empirically tunes the block-size for LAPACK routines and emphasizes its impact on performance [2]. Lam et al. study caching in the context of blocking within DLA kernels [3]. Iakymchuk et al. model the number of cache misses analytically based on a very detailed analysis of kernel implementations [4].

The rest of this paper is structured as follows. We introduce the considered problem and setup in Sec. 2 and establish bounds for the kernel execution times in Sec. 3. Then, we develop a cache prediction model in Sec. 4 and apply it to a broader range of scenarios in Sec. 5.



(a) in-algorithm (b) repeated execution

Fig. 1: In-algorithm timings and comparison with repeated execution. Along the x-axis, we enumerate the 1,873 kernel invocations within `dgeqrf`.

2 The Problem

In order to better understand the influence of caching on the performance of compute kernels, we focus on a specific, yet exemplary algorithm and setup: On one core of a quadcore Intel Harpertown E5450, we analyze the performance of LAPACK's QR decomposition (`dgeqrf`) linked to OpenBLAS v. 0.2.8 [5] on a square matrix of size[1] $n = 1{,}568$. With a size of 18 MB, this matrix exceeds this CPU's largest cache (L2), consisting of 6 MBs per 2 cores.

The routine `dgeqrf` implements a blocked algorithm and traverses the input matrix diagonally from the top left to the bottom right corner in steps of a prescribed block-size b. We fix this block-size — this routine's only optimization parameter — at $b = 32$. Within each step of the blocked traversal, `dgeqrf` executes the following sequence of kernels on operands of decreasing size: `dgeqr2` (unblocked QR), `dlarft` (form triangular factor T for the compact representation of Q), b `dcopy`s (together transpose a matrix panel), $\texttt{dtrmm}_{\texttt{RLNU}}$ (triangular matrix-matrix product)[2], $\texttt{dgemm}_{\texttt{TN}}$ (matrix-matrix product), $\texttt{dtrmm}_{\texttt{RUNN}}$, $\texttt{dgemm}_{\texttt{NT}}$, and $\texttt{dtrmm}_{\texttt{RLTU}}$.

To measure the execution time of these kernels within `dgeqrf` (henceforth called *in-algorithm timings*), we manually instrument this routine, and collect timestamps[3] between kernel invocations. The in-algorithm timings computed from these timestamps are presented in Fig. 1a: Along the x-axis, we enumer-

[1] With $n = 1{,}568 = 2^5 \cdot 7^2$, we choose a matrix size that is not a power of 2 to avoid problem size specific performance artifacts.

[2] The subscripts `R` through `U` are the values of the flag arguments `side`, `uplo`, `trans`, and `diag`; they distinguish the form of the operation performed by the kernel.

[3] Read from the CPU's time stamp counter through the assembly instruction `rdtsc`.

ate the 1,873 kernel invocations; along the y-axis we present timings of each invocation grouped by the type of kernels. The figure shows that the execution time is dominated by the two `dgemm`s (× and ○); notably, although the size of their operands is the same, the corresponding timings differ significantly. Our ultimate goal is to develop performance models that accurately predict such differences and all other features of the in-algorithm timings.

To focus on the cache related performance features, we here attempt to reconstruct the in-algorithm timings with a very elementary timing setup: repeated execution of the kernels independent from each other. In these executions, we use the same flags and matrix sizes as within `dgeqrf` and a well separated memory location for each operand. The relative error in execution time of the median of 100 such independent repetitions compared to the in-algorithm timings is shown in Fig. 1b. While the relative error for `dcopy` (•) is rather large, the total contribution of its 1,536 invocations to the total runtime is below 1%. Not considering these `dcopy`s[4], the absolute errors of the instrumented timings relative to the in-algorithm timings averaged across kernel invocations (in the following simply referred to as error) is 4.48%.

For most routines and especially for $\texttt{dtrmm}_{\texttt{RLTU}}$ (◇) and `dgeqr2` (×), the repeated execution underestimates the in-algorithm timings for the first 1,000 kernel invocations. More surprisingly however, $\texttt{dgemm}_{\texttt{NT}}$ is even overestimated — it is faster within `dgeqrf`.

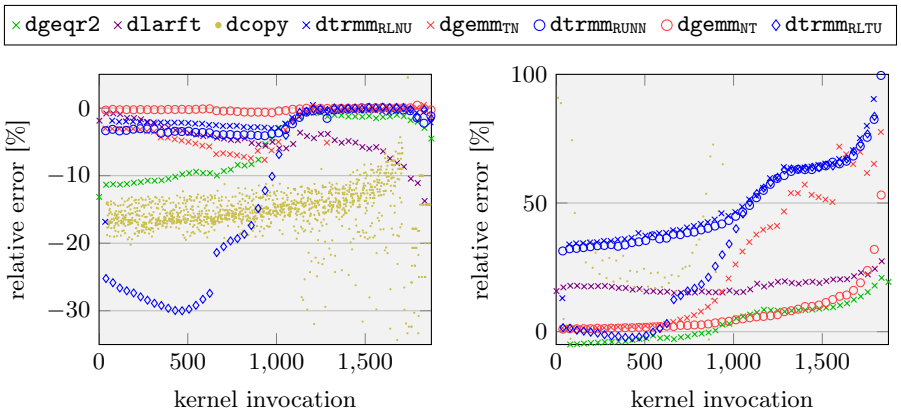
3 Cache-Aware Timings

The change in behavior noticeable around the 1,00th kernel invocation (see Fig. 1b) is directly linked to the size of the cache. While traversing the matrix, `dgeqrf` only operates on its bottom right quadrant, which becomes smaller at each iteration. Beyond the 1,00th invocation, the quadrant is small enough to fit in the L2 cache. As a result, the subsequent runtime measurements of repeated executions show only minimal differences with respect to the in-algorithm timings. This confirms the cache as the cause of the discrepancies.

To better understand the scope of this influence we now manipulate the cache locality of the kernel's operands in our independent executions. To do so, we assume a simplified cache replacement policy: a fully associative Least Recently Used (LRU) algorithm. We consider the two extreme scenarios in which the operands immediately required by the kernels are either entirely within the L2 cache or not at all. These in- and out-of-cache scenarios serve, respectively, as lower and upper bounds on the in-algorithm timings.

For kernels with operands whose size is smaller than 6MBs, repeated execution suffices to guarantee that the operands are in cache prior to execution. By contrast, when the aggregate size of all kernel operands exceeds 6MB (as for $\texttt{dgemm}_{\texttt{NT}}$ (○)), different kernel implementations (i.e. different libraries) may initially access different regions of the operands. An ideal in-cache setup would

[4] The system fluctuations cause variations of the `dgeqrf` timings of 0.057% on average. With the exception of the tiny `dcopy`s, these fluctuations are not significant.



(a) in-cache (b) out-of-cache

Fig. 2: In-cache and out-of-cache compared to in-algorithm timings. In (b), the error for `dcopy` (•) is around 1,000%.

place exactly the immediately accessed regions in cache. However, since we do not assume knowledge about kernel implementation, we restrict our in-cache setup to fulfill the reasonable assumption that input-only operands are accessed before input/output and output-oly operands. In order to accordingly prepare the cache, we touch[5] all input operands just before the kernel invocation. This timing setup yields the runtime estimates shown in Fig. 2a. Here, the estimates are in all cases equal to or underestimating the in-algorithm timings. The error is 4.51%.

Under the assumption of a fully associative LRU cache, to ensure that the operands are not in the cache, it suffices to touch a section of the main memory larger than the cache size. This approach yields the runtime estimates presented in Fig. 2b. Now, almost all estimates are equal to or overestimating the in-algorithm timings. The error is 29.1%.

Not only do the established in-cache and out-of-cache timings indeed serve as lower and upper bounds on the in-algorithm timings, for most kernel invocations one of these two bounds is actually attained (see Fig. 2). Based on this observation, the next section introduces a cache model to use these in-core and out-of-core timings to estimate the in-algorithm timings.

4 Modeling the Cache

In order to predict the state of the cache throughout the execution of `dgeqrf`, we consider which parts of its operands are accessed by its kernel invocations. `dgeqrf` itself receives three operands: the input matrix $A \in \mathbb{R}^{n \times n}$, an output

[5] By "touching", we mean a simple read+write access to the data, e.g. $x := x + \varepsilon$.

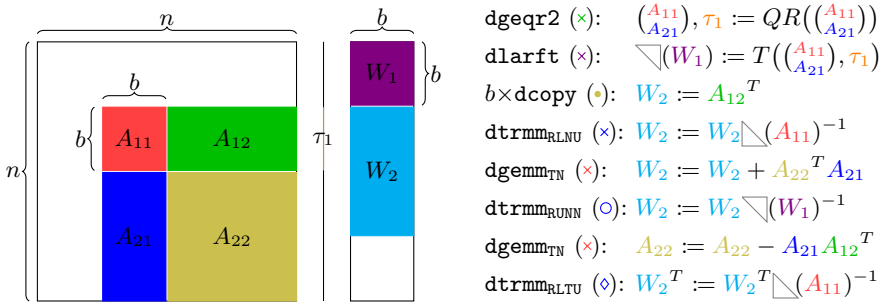


Fig. 3: Memory accesses by the kernels within one step of the blocked algorithm `dgeqrf`. The three shapes on the left represent `dgeqrf`'s operands A, τ, and W.

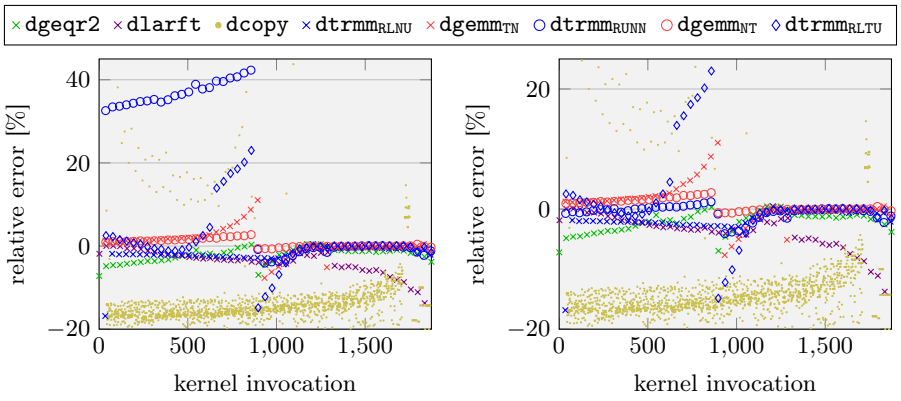
vector $\tau \in \mathbb{R}^n$, and auxiliary work space $W \in \mathbb{R}^{n \times b}$. Fig. 3 shows where within these three memory regions the operands of the kernels invoked in one step of `dgeqrf`'s blocked algorithm lie. Since we do not consider details of the kernel implementations, we do not make any assumptions on the patterns in which the kernels access their operands.

For the assumed fully associative LRU cache replacement policy, identifying if a memory region is available in cache reduces to the task of counting how many other data elements were accessed since its last use. To determine this count (henceforth referred to as *access distance*), we scan the sequence of kernel invocations and keep a history of the memory regions they access[6]. We consider the cache line as the smallest accessible memory unit: An access to a single data element means an access to the entire surrounding cache line. For each operand of a kernel invocation, we go backward through the access history until (and including) we find its last access; thereby summing the sizes of the accessed memory regions yields the operand's access distance. (If the access history does not reveal a previous access, the access distance is set to ∞.)

By comparing the obtained access distances to the cache size, we determine whether the corresponding operand is expected in the cache or not. Given these expectations, we separately sum the sizes of the in-cache and out-of-cache kernel operands. These sums are then used to weight the runtime of the corresponding timings to yield initial estimates of the instrumentation timings, shown in Fig. 4a. Comparing to Fig. 2, our mechanism chooses (or weights) the in-cache and out-of-cache timings correctly for most kernels. However, the error is still 4.65%, because for `dtrmm`$_{\texttt{RUNN}}$ (○) out-of-cache is erroneously favored over in-cache.

The reason for this flaw is that (see Fig. 3) `dtrmm`$_{\texttt{RUNN}}$ (○) is preceded by the large `dgemm`$_{\texttt{TN}}$ (×): This `dgemm`'s operands, which are together larger than cache, are accumulated into `dtrmm`$_{\texttt{RUNN}}$'s right-hand-side operand's access distance. However, since `dtrmm`$_{\texttt{RUNN}}$'s right-hand-side happens to be the output operand of the very matrix-times-vector-shaped `dgemm`$_{\texttt{TN}}$, it appears to be left in cache. We use

[6] The length of the list can safely be restricted to to the number of kernel calls per iteration of the blocked algorithm.



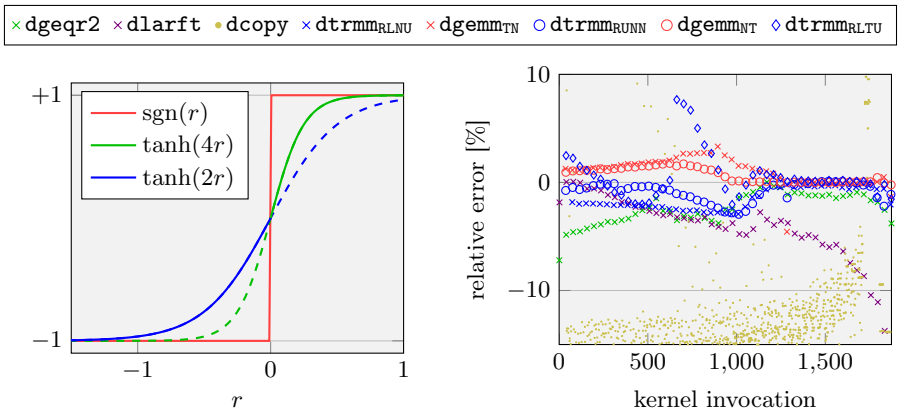
(a) basic estimates

(b) kernel-splitting estimates

Fig. 4: Basic and kernel-splitting estimates compared to in-algorithm timings

this insight to extend our cache model with a crucial assumption: After a kernel, whose (input-)output operand is significantly smaller than its input-only operands, we expect the (input-)output operand to be in cache. This assumption is implemented by splitting the memory accesses of such a kernel into two parts: The first access contains the large input-only operand(s), while the second only involves the small (input-)output operand. Therefore, the backward traversal of the access history will encounter the latter separately and, in case it is the sought operand, terminates before processing the cache-exceeding accesses. The timing estimates from this modifications (called *splitting estimates*) are shown in Fig. 4b. Here, *all* kernels are chosen correctly from the in-cache and out-of-cache timings. As a result, the error is reduced to 2.27%.

The only remaining deficiency of our estimates is in the form of severe spikes around the transition from out-of-cache to in-cache, around the 900th kernel invocation. To avoid such spikes, we apply smoothing of the association of operands with in-cache and out-of-cache. To determine if an operator was in-cache (+1) or out-of-cache (−1), we previously used a step function. In terms of the relative access distance $r = \frac{\text{(cache size)}-\text{(access distance)}}{\text{cache size}}$, this function was $\operatorname{sgn}(r)$. We now replace it with $f(r) = \begin{cases} \tanh(\alpha r), \text{ for } r \geq 0 \\ \tanh(\beta r), \text{ for } r < 0 \end{cases}$, where α and β are smoothing coefficients. As shown in Fig. 5a, $f(r)$ converges toward $\operatorname{sgn}(r)$ for both large and small values of r while showing a smooth transition of the origin. When applied to our estimates with empirical values of $\alpha = 4$ and $\beta = 2$, we obtain the smoothed estimates shown in Fig. 5b. With all estimates very close to the instrumentation timings, the error further decreases to 1.84%.



(a) smoothing functions

(b) smoothed estimates

Fig. 5: Smoothing functions and resulting estimates compared to in-algorithm timings

Table 1: Estimation improvements through cache-modeling for various scenarios

algorithm	#cores	BLAS	n	b	repeated execution	smoothed estimates	improvement
`dgeqrf`	1	OpenBLAS	1,568	32	4.48%	1.84%	×2.44
`dgeqrf`	1	OpenBLAS	1,568	**64**	3.15%	1.64%	×1.92
`dgeqrf`	1	OpenBLAS	1,568	**128**	2.68%	2.13%	×1.26
`dgeqrf`	1	OpenBLAS	**2,080**	32	5.11%	1.84%	×2.78
`dgeqrf`	1	OpenBLAS	**2,400**	32	5.23%	1.75%	×2.99
`dgeqrf`	1	**ATLAS**	1,568	32	3.55%	1.98%	×1.79
`dgeqrf`	1	**MKL**	1,568	32	8.58%	4.40%	×1.95
`dgeqrf`	**2**	OpenBLAS	1,568	32	9.58%	4.63%	×2.07
`dgeqrf`	**4**	OpenBLAS	1,568	32	22.71%	19.75%	×1.15
$\texttt{dtrtri}_{\texttt{LN}}$	1	OpenBLAS	2,400	32	6.70%	3.37%	×1.99
$\texttt{dpotrf}_{\texttt{U}}$	1	OpenBLAS	2,400	32	11.18%	7.56%	×1.48

5 Results

In the previous sections we focused on a very specific setup (see Sec. 2). To demonstrate that our observations and models are more broadly applicable, we now vary this setup and present the obtained accuracy improvements of our smoothed estimates over the repeated execution timings in Table 1.

Although the error of our estimates remains above 1.5%, it is in many cases an improvement of about a factor of 2. For both increasing block-size b and matrix size n, with a varying error for repeated executions timings, our estimates reliably yield an error of around 2%. When OpenBLAS is replaced with ATLAS [6], the picture is very much the same; for Intel's MKL instead the error in both the

repeated execution timings and in our estimates increases significantly[7]; however, the latter is still an improvement over the former by a factor of 2. The same can be observed when doubling the number of cores to 2. When we use all 4 cores of our CPU however, the error increases drastically; this is because every two cores share a separate L2 cache, while our model is designed for a single large cache. Finally, our approach also applies to other LAPACK algorithms: For $\texttt{dtrtri}_{\texttt{LN}}$ (inversion of a lower triangular matrix) and $\texttt{dpotrf}_{\texttt{U}}$ (Cholesky decomposition of an upper triangular matrix), it yields considerable improvements in accuracy.

6 Conclusion

In this paper, we studied the influence of caching on the execution time of sequences of dense linear algebra kernels within blocked algorithms. We established in-cache and out-of-cache timings as lower and upper bounds on the kernel execution times within the algorithm. We then developed a cache tracking model that, based on a sequence of kernel invocations, identifies which memory regions are available in cache and which are not. With the help of this model, we were able to combine the in-cache and out-of-cache timings into highly accurate estimates for the actual kernel execution times. This methodology was shown to noticeably reduce the average error for our estimates. The insights and results presented in this paper constitute an important step towards our ultimate goal of selecting and optimally configuring dense linear algebra algorithms through performance models of the computational kernels, without ever executing the algorithms themselves.

References

1. Peise, E., Bientinesi, P.: Performance Modeling for Dense Linear Algebra. In: Proceedings of the 3rd International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS12). (November 2012)
2. Whaley, R.: Empirically tuning lapack's blocking factor for increased performance. In: Computer Science and Information Technology, 2008. IMCSIT 2008. International Multiconference on. (October 2008) 303–310
3. Lam, M.D., Rothberg, E.E., Wolf, M.E.: The cache performance and optimizations of blocked algorithms. In: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS IV, New York, NY, USA, ACM (1991) 63–74
4. Iakymchuk, R., Bientinesi, P.: Modeling Performance through Memory-Stalls. ACM SIGMETRICS Performance Evaluation Review **40**(2) (2012)
5. OpenBLAS: http://www.openblas.net/
6. Whaley, R.C., Dongarra, J.: Automatically Tuned Linear Algebra Software. Technical Report UT-CS-97-366, University of Tennessee (December 1997)

[7] For MKL, we removed the step of splitting (input-)output from input-only operands in the access history; this BLAS library does not leave the output operand in cache.