Optimized Batched Linear Algebra for Modern Architectures

DOI:
10.1007/978-3-319-64203-1_37

Document Version
Accepted author manuscript

Published in:
Euro-Par 2017: Parallel Processing

Optimized Batched Linear Algebra for Modern Architectures

Jack Dongarra[1,2], Sven Hammarling[2], Nicholas J. Higham[2], Samuel D. Relton[2], and Mawussi Zounon[2]

[1] University of Tennessee, Oak Ridge National Laboratory, TN, USA.
dongarra@icl.utk.edu
[2] School of Mathematics, The University of Manchester, Manchester, UK.
sven.hammarling@btinternet.com, {nick.higham, samuel.relton, mawussi.zounon}@manchester.ac.uk

Abstract. Solving large numbers of small linear algebra problems simultaneously is becoming increasingly important in many application areas. Whilst many researchers have investigated the design of efficient batch linear algebra kernels for GPU architectures, the common approach for many/multi-core CPUs is to use one core per subproblem in the batch. When solving batches of very small matrices, 2×2 for example, this design exhibits two main issues: it fails to fully utilize the vector units and the cache of modern architectures, since the matrices are too small. Our approach to resolve this is as follows: given a batch of small matrices spread throughout the primary memory, we first reorganize the elements of the matrices into a contiguous array, using a block interleaved memory format, which allows us to process the small independent problems as a single large matrix problem and enables cross-matrix vectorization. The large problem is solved using blocking strategies that attempt to optimize the use of the cache. The solution is then converted back to the original storage format. To explain our approach we focus on two BLAS routines: general matrix-matrix multiplication (GEMM) and the triangular solve (TRSM). We extend this idea to LAPACK routines using the Cholesky factorization and solve (POSV). Our focus is primarily on very small matrices ranging in size from 2×2 to 32×32. Compared to both MKL and OpenMP implementations, our approach can be up to 4 times faster for GEMM, up to 14 times faster for TRSM, and up to 40 times faster for POSV on the new Intel Xeon Phi processor, code-named Knights Landing (KNL). Furthermore, we discuss strategies to avoid data movement between sockets when using our interleaved approach on a NUMA node.

1 Introduction

Over the last decade, the high-performance computing (HPC) community has made significant strides in solving large-scale matrix problems efficiently. Another major challenge is to achieve good performance when computing a large batch of small matrix problems: this situation occurs commonly in applications

including deep learning libraries [1, 3], multifrontal solvers for sparse linear systems [5], and radar signal processing [4] etc. In deep learning applications, for example, many applications require the solution of thousands of independent (and very small) general matrix-matrix multiplication (GEMM) in Equation (1), where *batch_count* is number of independent problems in the batch.

$$C^{(i)} \leftarrow \alpha^{(i)} A^{(i)} B^{(i)} + \beta^{(i)} C^{(i)}, \qquad i = 1 : batch\_count. \tag{1}$$

The challenge is to make more efficient use of computational cores than a simple `for loop` around a single call to a vendor optimized GEMM kernel, where there may not be enough work to keep the cores running at full efficiency. Note that, depending on the application, the batch can contain matrices of different sizes and $\alpha^{(i)}$ and $\beta^{(i)}$ can have different values. But in this work, we focus on the "fixed batch" case, which is more common in applications. In a fixed batch the values of α and β are the same for all the problems in the batch and the matrices have a constant size, i.e. the dimensions of $A^{(i)}$ are the same for all i in $[1, batch\_count]$ and similarly for the matrices $B^{(i)}$ and $C^{(i)}$.

To address the need for efficient libraries to perform batches of small linear algebra operations in parallel, new APIs have been investigated and a comparative study of these APIs is given in [11]. While most research focuses on providing high-performance batch linear algebra implementation for GPU architectures, there is—at the time of writing—no better solution than using one core per problem when it comes to many/multi-core architectures. When solving batches of very small matrices, 2×2 for example, this design exhibits two main problems. Due to the small size of the matrices we fail to fully utilize the vector units and the cache of modern architectures.

In this work, we focus on level 3 BLAS routines because they are the critical building blocks of many high-performance software. Our motivation to focus on GEMM and triangular solve (TRSM) is that all of level 3 BLAS routines except TRSM can be viewed as a specialized GEMM [8]. However, regardless of these considerations, our proposed solutions are easily extended to all BLAS kernels including the level 1 and 2 algorithms.

The key aspect of our approach is as follows: given a batch of small matrices spread throughout RAM we first reorganize the elements of the matrices into a contiguous array, using a block interleaved memory format, which allows us to process the small independent problems as a single large matrix problem. The large problem is solved using blocking strategies that attempt to optimize the use of the cache. The solution is then converted back to the original storage format.

Compared to the MKL batched BLAS implementation and an OpenMP `for loop` around MKL BLAS kernels, our implementation is up to 4 times faster for DGEMM and up to 14 times faster for DTRSM on the new self-hosted Intel Xeon Phi processors, code named Knights Landing (KNL). By extending this idea to LAPACK routines, specifically the Cholesky factorization and solve (POSV), we can see that our approach can be extremely efficient and performs up to 40 times faster than using an OpenMP `for loop` on the Intel KNL architecture.

The paper is organized as follows. In Section 2 we present the current state of the art in batch BLAS algorithms and their limitations. Section 3 describes our approach, the block interleaved batch BLAS, followed by some performance analyses. In section 4, we discuss how to extend batch operations to include LAPACK routines, with a focus on Cholesky factorization and solve. We then discuss the main performance issues raised when using NUMA nodes in Section 5 before giving some concluding remarks in Section 6.

2 Related work

Motivated by the efficiency of vendor supplied libraries for small problems on many/multi-core CPU architectures, the currently accepted method for solving batches of small problems is to have a single core per problem in the batch [6]. Therefore, most of the effort in recent years has been devoted to developing efficient batch kernels for GPUs. Our aim is to challenge the conventional wisdom in many/multi-core CPU architectures, however a reader interested in efficient CUDA kernels for batch BLAS operations may look at [2, 7, 9, 10].

2.1 Multicore CPUs and Xeon Phi implementations

At first glance, batched BLAS operations on multicore CPUs seemed to be reduced to the choice between: (i) solving one problem at the time using all the available cores or (ii) solving many independent problems in parallel using a single core per problem. Whenever small matrices are used the second approach is preferred can be implemented simply: merely an OpenMP `for loop` around vendor supplied BLAS kernels is required.

When processing thousands of very small matrices, the error checking procedure implemented by most of optimized vendor kernels can be significantly time-consuming. To alleviate this overhead, Intel MKL allows us to skip the error checking thanks to the `MKL_DIRECT_CALL` or `MKL_DIRECT_CALL_SEQ` macros. Hence, the common wisdom for a fixed batched BLAS implementation consists in checking the arguments once, as all the problems in the batch share the same error prone arguments, then perform an OpenMP `for loop` over optimized BLAS kernels.

While these solutions are acceptable for batches of matrices of medium size, they may fail to exploit efficiently wide vector units on modern architectures. For example, the AVX-512 vector units available in the Intel KNL, enable the completion of 8 double precision vector operations within each cycle, while a 2×2 matrix can fill only half of such a vector unit.

Furthermore, some BLAS routines don't offer enough parallelism. For example in the case of batched TRSM, the computation of each entry of each right-hand side requires a single division before the updates. When one right-hand side is required, regardless of the matrix size, the common approach will perform only one double precision division in one clock cycle on a core capable

of 8 double precision divisions. However, by using the interleaved memory layout described in Section 3 one can saturate the vector units at all steps of the algorithm thanks to cross-matrix vectorization.

3 Data layout optimization

Dealing with thousands of independent, small matrices requires a careful choice of memory layout, and a good memory layout should be user-friendly without penalizing performance. There are currently 3 competing data layouts advocated by the linear algebra community for batched BLAS operations. In this section, we illustrate the underlying idea of each data layout using the example of solving three independent 2×2 matrix problems ($A^{(1)}$,$A^{(2)}$, $A^{(2)}$).

3.1 Pointer-to-pointer layout

Most of the existing interfaces for both CPU and GPU architectures use an array of pointers, where each pointer leads to a matrix in memory. We call this the pointer-to-pointers (P2P) layout. As depicted in Figure 1, it allows us to allocate matrices independently. This is the solution currently used in *cblas_dgemm_batch* and *cublasDgemmBatched*, the batch DGEMM kernels available in Intel MKL 11.3 beta and NVIDIA cuBLAS version 4.1, respectively. This approach is very flexible but has two main issues as reported in [7] and [11]. First, the allocation and deallocation of thousands of small matrices can be excessively time-consuming. Second, processing very small matrices stored separately can increase the number of memory accesses required and induces sub-optimal cache use. In addition, the array of pointers approach suffers from high data movement costs when data is offloaded to hardware accelerators.

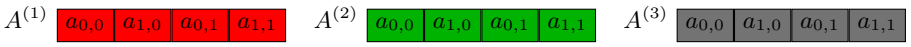


Fig. 1: Pointer to pointer (P2P) memory layout. The three matrices are stored in different memory locations in column major order.

3.2 Strided layout

To alleviate the design issues intrinsic to the pointer to pointers memory layout, NVIDIA cuBLAS advocated another interface called the strided layout [12]. It consists of storing a collection of matrices in one contiguous block of memory. As illustrated in Figure 2, this involves allocating a large chunk of memory to store all the A^i matrices.

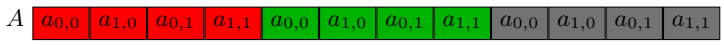


Fig. 2: Strided memory layout. The three matrices are stored in one contiguous chunk of memory.

3.3 Interleaved memory layout

Solving batches of small size matrix problems on modern architectures is challenging because these architectures are primarily designed to address large-scale problems. The main objective of the interleaved memory layout approach is to reformulate the thousands of independent small BLAS operations as a single large-scale problem. This involves providing a relevant way to store the independent matrices. Interleaving the entries of different matrices enables cross-matrix vectorization to fill the vector units on modern architectures. As illustrated in Figure 3, the interleaved layout is a permutation of the strided memory layout.

Fig. 3: Interleaved memory layout. The three matrices are stored in one contiguous chunk of memory, but their elements are mixed together.

3.4 Design of interleaved batch BLAS

While the interleaved layout has the potential for better vectorization and data locality, it requires redesigning the BLAS algorithms. This is achieved by adding inner `for loops` to the original algorithms in order to create batches of floating point operations. We illustrate this in a simplified version of an interleaved TRSM displayed in Algorithm 1. For the sake of simplicity and readability, A and B denote the interleaved layout containing $m \times m$ lower triangular matrices $A^{(i)}$ and the $m \times n$ right hand side matrices $B^{(i)}$, respectively; and the notation $A[i][j][idx]$ is used to refer to the entry $a_{i,j}$ of the matrix $A^{(idx)}$ in the batch.

Compared to the original TRSM algorithm, our interleaved version has an additional `for loop` (Algorithm 1, line 5) that accesses each matrix in the batch. Some operations have also been moved to the innermost loop (Algorithm 1, line 7 and 10), for the sake of better vectorization without affecting the numerical stability. The innermost loop contains thousands of floating point operations parallelized among cores thanks to the `#pragma openmp parallel for` directive whilst the `simd` directive makes use of vector pipelines within each core.

3.5 Block interleaved layout

While the interleave layout increases the vectorization within the floating point units, it may lead to a high cache miss rate: since the first entries of the matrices

Algorithm 1 Interleaved TRSM algorithm:$B^{(i)} \leftarrow \alpha(A^{(i)})^{-1}B^{(i)}$

```
1: for j ← 1 to n do                                   ▷ Iterate over n right hand sides
2:     for k ← 1 to m do                                    ▷ Iterate over rows of A
3:         for i ← k to m do                                ▷ Iterate over columns of A
4:             #pragma omp parallel for simd
5:             for idx ← 1 to batch_count do     ▷ Iterate over problems in the batch
6:                 if k == 0 then
7:                     B[i][j][idx] ← β × B[i][[j][idx]                        ▷ Apply α
8:                 end if
9:                 if i == k then
10:                    B[k][j][idx] ← B[k][j][idx]/A[k][k][idx]          ▷ Division by a_{k,k}
11:                    continue
12:                end if
13:                B[i][j][idx] ← B[i][j][idx] − B[k][j][idx] × A[i][k][idx]        ▷ Update
14:            end for
15:        end for
16:    end for
17: end for
```

are stored followed by the second entries etc., the next entries required by the algorithm are unlikely to be in the cache at any given time. To alleviate this problem, we divide the initial batch into small sub-batches (blocks), then apply the interleaved strategy within each block. The block size is selected such that each sub-batch could be solved efficiently by a single core. The optimal block size is a tunable parameter and depends on the number of cores and the memory hierarchy of the target machine. In our experiments we let InterleaveTRSM denote Algorithm 1. For the block interleaved TRSM (BlockInterleaveTRSM) we replace `#pragma omp parallel for simd` by `#pragma simd` in Algorithm 1 and use an OpenMP for loop over the blocks defined above.

3.6 Interleaved batch BLAS user interfaces

We note that data layout utilized by the user and that used internally to the computation need not be the same. Indeed our code has two interfaces: a simple P2P interface for user convenience (which performs all the memory layout conversion internally) and, for expert users, we expose the interleaved layout kernels and the associated conversion functions directly. For the simpler functions with P2P-based interfaces, the design is as follows:

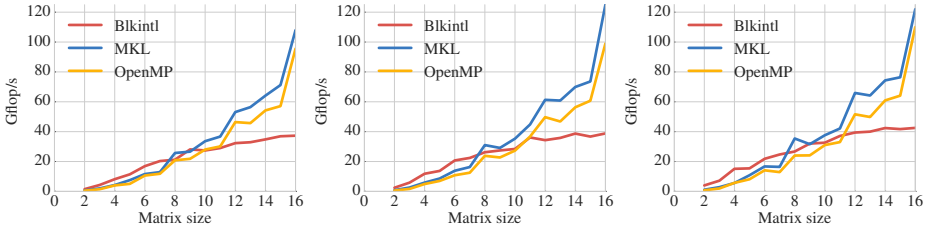
1. Convert from user layout to block interleaved layout.
2. Call block interleaved kernels.
3. Convert back to the user layout.

The conversion routines are designed for better data locality, and exploit both thread and vector level parallelism. For safety, the user is required to provide the

extra memory intended for conversion. More details on the API and the codes can be found on our Github repository[3].

3.7 Experimental results

The aim of this subsection is to evaluate how the block interleaved (`Blkintl`) batch kernels compare to both the optimized Intel MKL batch BLAS kernels (`MKL`) and OpenMP `for loop` over Intel MKL BLAS kernels (`OpenMP`). The experiments are performed on a 68-core Intel KNL[4] configured in flat mode with all data allocated in the high bandwidth memory. To obtain more reliable results, we take the average time over ten runs and carefully flush the cache between each run.



(a) 5,000 DGEMM (b) 10,000 DGEMM (c) 20,000 DGEMM

Fig. 4: Performance comparison of different implementations of batch DGEMM using 68 threads on the Intel KNL with different batch sizes on square matrices ranging in size from 2×2 to 16×16.

The first experiment displayed in Figure 4, compares the performance in GFlop/s (the higher the better) of three batch DGEMM implementations. A batch containing a few thousand matrices is enough to saturate the KNL, and the performance doesn't increase significantly when doubling the batch size. It is important to notice that we also consider layout conversion time in the performance of `Blkintl`. The conversion overhead is significant for GEMM because it involves three batches of matrices ($A^{(i)}$, $B^{(i)}$ and $C^{(i)}$). Despite this overhead, `Blkintl` outperforms `MKL` for very small matrices ranging from 2×2 to 7×7, and `OpenMP` for matrices up to 11×11. In the particular case of a batch of 20,000 2×2, `Blkintl` is four times faster than `MKL`. As the matrix sizes increase, both `MKL` and `OpenMP` outperform `Blkintl` for two main reasons: (i) the increasing cost of data layout conversion, and, (ii) the current `Blkintl` implementation is not taking advantage of advanced memory prefetching strategies. Since the three

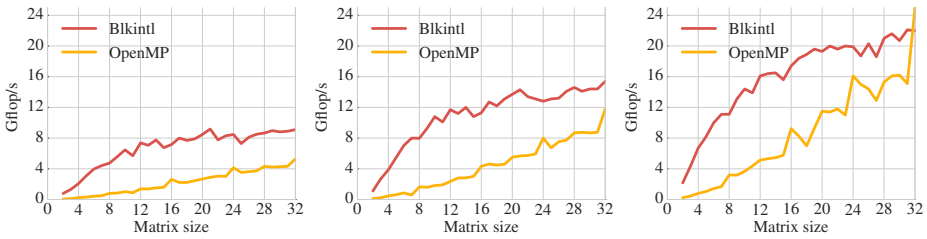
[3] https://github.com/sdrelton/bblas_interleaved

[4] https://ark.intel.com/products/94035/Intel-Xeon-Phi-Processor-7250-16GB-1_40-GHz-68-core

kernels are performing the same floating point operations in a different order, we can view this as a race to fill the vector units within the cores.

Furthermore, on average, MKL is 15% better than OpenMP. This suggests that the MKL approach to batch BLAS is more sophisticated than a simple OpenMP `for loop` over optimized BLAS kernels.

As MKL provides only batch kernels for DGEMM, in Figure 5 we can only compare the performance of Blkintl and OpenMP for a batch of 10,000 DTRSM. Compared to GEMM, TRSM has a lower numerical intensity, but the performance can be increased by operating on multiple right-hand sides. In Figures 5a and 5b, for example, the performance almost doubled for both OpenMP and Blkintl from one right-hand side to two. The superiority of Blkintl over OpenMP is significant even with matrix sizes up to 32×32, which is consistent with our analysis in Subsection 2.1. Interleaving multiple triangular solves alleviates the synchronization penalty of performing only one division per right-hand side before parallel updates. Another factor is a lower conversion overhead: since the TRSM algorithm operates on triangular matrices and a few right-hand sides, the conversion overhead is reasonably low when compared to GEMM.



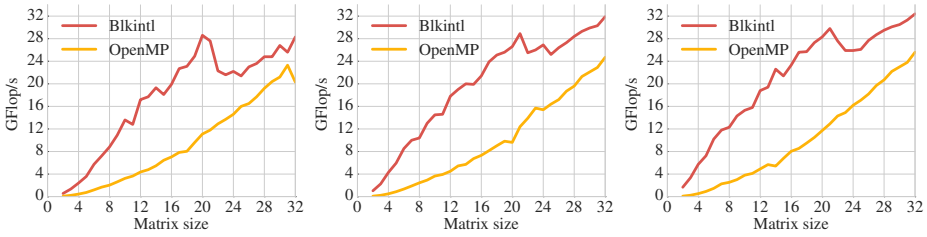
(a) One right-hand side (b) Two right-hand sides (c) Four right-hand sides

Fig. 5: Performance of a batch of 10,000 DTRSM operations using 68 threads on the Intel KNL with different numbers of right hand sides (rhs), on matrices ranging in size from 2×2 to 32×32. Blkintl is 14 times better than OpenMP in (a) for 2×2 matrices.

4 Application to batched Cholesky factorization and solve

Efficient LAPACK kernel implementations are commonly achieved by dividing the matrices in blocks or tiles, and taking advantage of Level 3 BLAS routines as much as possible to process the blocks or tiles. However, very small matrices cannot easily be divided into blocks. To solve batches of very small LAPACK problems we can extend the interleaved approach to LAPACK routines. This allows us to optimize the use of wide vector units and also take advantage of interleaved BLAS kernels whenever possible. In particular we will focus on the

Cholesky solve (POSV) algorithm which solves $Ax = b$, where A is a symmetric definite positive matrix. It starts with a Cholesky factorization (POTRF) $A = LL^T$, then performs a forward substitution (TRSM kernel, $Ly = b$) before finally performing a backward substitution (TRSM kernel, $L^T x = y$). In this example, the implementation effort involves mainly developing the `Blkintl` POTRF kernel, as `Blkintl` TRSM has already been discussed above.



(a) 5,000 DPOTRF. (b) 10,000 DPOTRF (c) 20,000 DPOTRF

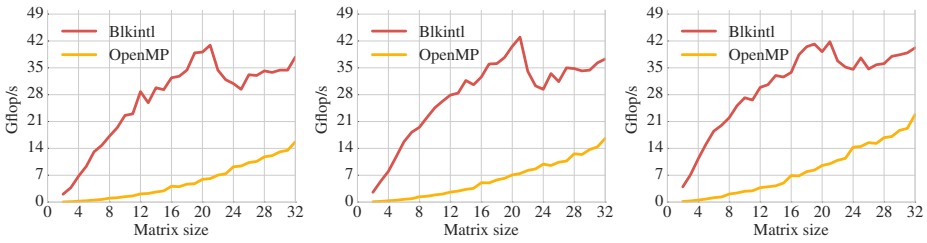
Fig. 6: Performance of batch Cholesky factorization (DPOTRF) using 68 threads on the Intel KNL, with different batch sizes, on matrices ranging in size from 2×2 to 32×32. `Blkintl` is 18 times better than `OpenMP` in (c) for 2×2 matrices.

As illustrated in Figure 6, `Blkintl` POTRF outperforms the `OpenMP` version for the same reasons discussed for the `Blkintl` TRSM kernel: better use of the vector units and low memory conversion overhead, and the conversion cost is even lower than the TRSM case since it involves only one triangular matrix per problem in the batch. An overview of the `Blkintl` POSV algorithm is provided in Algorithm 2.

Algorithm 2 `Blkintl` POSV algorithm: $B^{(i)} \leftarrow (A^{(i)})^{-1} B^{(i)}$

1: Conversion of $A^{(i)}$ and $B^{(i)}$ into `Blkintl` format
2: Call `Blkintl` POTRF
3: Call `Blkintl` TRSM (forward substitution)
4: Call `Blkintl` TRSM (backward substitution)
5: Convert $A^{(i)}$ and $B^{(i)}$ back to the user's format

The two main features of Algorithm 2 are: (i) conversions are performed once before using the three `Blkintl` kernels, (ii) reuse of `Blkintl` BLAS kernels. In particular, performing the conversion only once allows us to obtain very good performance with this approach. The results shown in Figure 7, for example, show that the gap in performance between `Blkintl` and `OpenMP` is larger than the one observed for TRSM in Figure 5.



(a) One right-hand side (b) Two right-hand sides (c) Four right-hand sides

Fig. 7: Performance on a batch of 10,000 Cholesky solve (DPOSV) using 68 threads on the Intel KNL with different numbers of right-hand sides, on matrices ranging in size from 2×2 to 32×32. `Blkintl` is 40 times better than `OpenMP` in (a) for 2×2 matrices.

The same strategy is applicable to other batched LAPACK kernels, with lots of potential for large speedups over an OpenMP `for loop`.

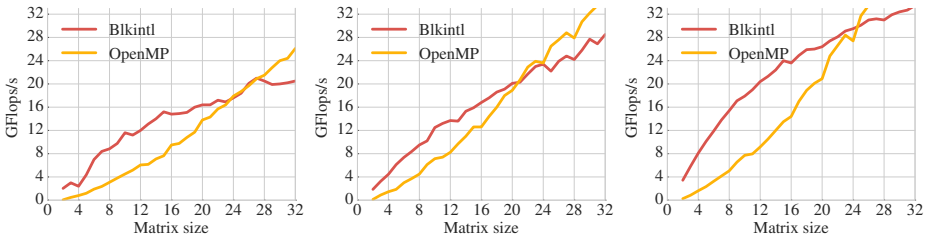
5 Efficient Batch linear algebra on NUMA nodes

As explained in Subsection 3.7, obtaining good performance is a race to fill the vector units of the cores as quickly as possible. In addition, data layout conversions required by `Blkintl` make our algorithms sensitive to data locality and data movement. These two factors are potential limitations for achieving good performance on non-uniform memory access (NUMA) nodes. In fact, when running a batch of very small matrices on a 2-socket NUMA node for example, the matrices are more likely to be allocated on a single socket, and the second socket will have only a remote access to data. This induces a high communication cost and performance drop due to the cost of remote memory access. This issue is commonly addressed by interleaving the data allocation thanks to the `numactl –interleave=all` option available on Linux systems. Memory will then be allocated using a round robin procedure between the nodes. As depicted in Figure 8, there is a slight performance improvement for both `Blkintl` and `OpenMP` when changing the standard memory allocation (Figure 8a) into the interleaved allocation configuration (Figure 8b). In general spreading the memory allocation improves the performance but, in the case of batch operations, there is no guarantee that we will allocate all data required for each independent problem on the same node. For example $A^{(i)}$ may be allocated on the first socket while the corresponding $B^{(i)}$ allocated on the second socket.

One way to significantly improve the performance is to split the batch into two independent batches and use one socket per batch. Unfortunately current OpenMP runtimes are not NUMA aware, however the user can manage the memory allocation themselves to enforce optimal data placement, using the `libnuma API` for example. The user can then call our batch BLAS kernel on each socket in

parallel. This strategy should improve the performance significantly as observed in Figure 8c, but requires a lot of user effort.

On the particular machine we used, the NUMA node vector units are half the size of the Intel KNL vector units. This explains the decrease of the performance gap between `Blkintl` and `OpenMP` when compared to those observed for Intel KNL. We believe that further studies can help in designing new efficient batch kernels which are specially optimized for NUMA nodes.



(a) Standard NUMA allocation

(b) Using numactl interleave

(c) Memory distributed half per node

Fig. 8: Performance of a batch of 10,000 Cholesky solve (DPOSV) operations using 20 threads on a NUMA node of two 10-core sockets, Intel Xeon E5-2650 v3 (Haswell), with different numbers of right-hand sides, on matrices ranging in size from 2×2 to 32×32.

6 Concluding remarks

In this research we have explained, and demonstrated the large potential of, the block interleaved strategy for batched BLAS operations. We have shown that our approach can offer significant performance improvements over an OpenMP `for loop` around vendor optimized BLAS kernels, with speedups of up to 40× for a batched Cholesky solve.

While generally satisfactory speedups are achieved on the Intel KNL architecture, we noted that further prefetching techniques may help to further improve the performance of the `Blkintl` DGEMM kernel. We have also shown that advanced memory placement configurations are necessary to increase the performance of batched kernels on NUMA nodes.

Finally, this study has focused only on an element-wise interleaving strategy. However, we believe that other data interleaving approaches such as row interleaving, column interleaving, and mixtures of the above could also provide similar (or even better) performance. It is clear that there is a large amount of further investigation to be done in this area.

Acknowledgements

The authors would like to thank The University of Tennessee for the use of their computational resources. This research was funded in part from the European Union's Horizon 2020 research and innovation programme under the NLAFET grant agreement No. 671633.

References

1. Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.
2. Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack J. Dongarra. Performance, design, and autotuning of batched GEMM for gpus. In *High Performance Computing - 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*, pages 21–38, 2016.
3. Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, et al. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.
4. Michael J Anderson, David Sheffield, and Kurt Keutzer. A predictive model for solving small linear algebra problems in gpu registers. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 2–13. IEEE, 2012.
5. Iain Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Trans. Math. Softw.*, 9(3):302–325, 1983.
6. Azzam Haidar, Tingxing Tim Dong, Stanimire Tomov, Piotr Luszczek, and Jack Dongarra. A framework for batched and gpu-resident factorization algorithms applied to block householder transformations. In *International Conference on High Performance Computing*, pages 31–47. Springer, 2015.
7. Chetan Jhurani and Paul Mullowney. A gemm interface and implementation on nvidia gpus for multiple small matrices. *Journal of Parallel and Distributed Computing*, 75:133–140, 2015.
8. Bo Kågström, Per Ling, and Charles van Loan. Gemm-based level 3 blas: High-performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Softw.*, 24(3):268–302, September 1998.
9. M. Graham Lopez and Mitchel D. Horton. *Batch Matrix Exponentiation*, pages 45–67. Springer International Publishing, Cham, 2014.
10. Ian Masliah, Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, Marc Baboulin, Joël Falcou, and Jack Dongarra. High-performance matrix-matrix multiplications of very small matrices. In *European Conference on Parallel Processing*, pages 659–671. Springer, 2016.
11. Samuel D. Relton, Pedro Valero-Lara, and Mawussi Zounon. A comparison of potential interfaces for batched BLAS computations. MIMS EPrint 2016.42, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, 2016.
12. Yang Shi, UN Niranjan, Animashree Anandkumar, and Cris Cecka. Tensor contractions with extended blas kernels on cpu and gpu. *arXiv preprint arXiv:1606.05696*, 2016.