Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 108C (2017) 495–504

Procedia Computer Science

International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland

The Design and Performance of Batched BLAS on Modern High-Performance Computing Systems

Jack Dongarra[1,2], Sven Hammarling[2], Nicholas J. Higham[2], Samuel D. Relton[2], Pedro Valero-Lara[3], 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.
{nick.higham, samuel.relton, mawussi.zounon}@manchester.ac.uk
[3] Barcelona Supercomputing Center, Barcelona, Spain.
pedro.valero@bsc.es

Abstract

A current trend in high-performance computing is to decompose a large linear algebra problem into batches containing thousands of smaller problems, that can be solved independently, before collating the results. To standardize the interface to these routines, the community is developing an extension to the BLAS standard (the batched BLAS), enabling users to perform thousands of small BLAS operations in parallel whilst making efficient use of their hardware. We discuss the benefits and drawbacks of the current batched BLAS proposals and perform a number of experiments, focusing on a general matrix-matrix multiplication (GEMM), to explore their affect on the performance. In particular we analyze the effect of novel data layouts which, for example, interleave the matrices in memory to aid vectorization and prefetching of data. Utilizing these modifications our code outperforms both MKL[1] and CuBLAS[2] by up to 6 times on the self-hosted Intel KNL (codenamed Knights Landing) and Kepler GPU architectures, for large numbers of double precision GEMM operations using matrices of size 2×2 to 20×20.

Peer-review under responsibility of the scientific committee of the International Conference on Computational Science

Keywords: BLAS; Batched BLAS; Scientific computing; High-performance computing; Memory management; Parallel processing

1 Introduction

Over the past few decades there has been a tremendous amount of community effort targeting the design and implementation of efficient linear algebra software. The main focus of this drive has been to solve larger problems in less time. As a result, numerous libraries have been designed to take advantage of advances in computer architecture and exploit the parallelism

[1] https://software.intel.com/en-us/intel-mkl
[2] http://docs.nvidia.com/cuda/cublas

1877-0509
Peer-review under responsibility of the scientific committee of the International Conference on Computational Science
10.1016/j.procs.2017.05.138

both within a single node (using hardware accelerators), and between nodes (communicating using MPI, for example).

In an attempt to utilize highly parallel computing resources more efficiently, there is a current trend towards splitting large linear algebra problems into thousands of smaller subproblems that can be solved concurrently [2]. One example of this is given by multifrontal solvers for sparse linear systems [7]. Many popular linear algebra libraries such as Intel MKL and NVIDIA CuBLAS have begun to provide limited support for this approach but no complete set of linear algebra routines operating on batches of small matrices is available.

The solution to this problem is to develop a new standard set of routines for carrying out linear algebra operations on batches of small matrices, building on the well-known Basic Linear Algebra Subproblems (BLAS) standard [5], [6], [10]. The idea behind the batched BLAS (BBLAS) is to perform multiple BLAS operations in parallel on many small matrices, making more efficient use of the hardware than a simple OpenMP for loop would allow. For example, if we consider a general matrix-matrix multiplication (GEMM) operation over a batch of N matrices then we would like to compute, in parallel,

$$C_i \leftarrow \alpha_i A_i B_i + \beta_i C_i, \qquad i = 1 : N. \tag{1}$$

In this example we might keep the sizes of the matrices and the values of α_i and β_i constant throughout the entire batch or allow them to vary, depending upon the application that we have in mind.

One particular application which can benefit dramatically from performing many small matrix multiplications in parallel is deep learning: the batched GEMM functionality in vendor libraries is already being utilized in popular machine learning libraries such as TensorFlow [1] and Theano [4]. Further examples of applications where the solution of many small problems are required include domain decomposition [3], the rendering of 3D graphics in web browsers [8], metabolic networks [9], astrophysics [12], matrix-free finite element methods [11], and the solution of separable elliptic equations [14].

Currently, libraries that implement BBLAS functionality use a relatively simple memory layout (explained in section 2) which generally gives suboptimal performance. One of our primary goals in this paper is to investigate a number of potential optimizations to increase the performance of BBLAS routines for small matrices on modern parallel architectures. We explore, amongst other things, the effect of different API designs and memory layouts on performance. Currently, there is no standard interface for BBLAS operations and no complete implementation of batched BLAS routines is available. Intel MKL has support for batched GEMM computation whilst NVIDIA CuBLAS supports batched GEMM and triangular solve (TRSM), plus some batched LAPACK routines; but these libraries do not share the same API.

The remainder of this article is organized as follows. In section 2 we outline the different APIs for BBLAS and perform some experiments to compare their associated overheads. Section 3 contains discussion and experiments to determine the effect that the memory layout has on the performance of BBLAS operations and the transfer to and from hardware accelerators, which are an important consideration when designing an efficient API. In section 4 we focus on the performance of a novel data layout, which interleaves the batches of matrices in memory, on both GPUs and the self-hosted Intel KNL. Concluding remarks are given in section 5.

2 Batched BLAS

Two main approaches can be taken to allocate computational resources to batched BLAS operations. First, we could compute each BLAS operation in order and allocate all available cores

to each subproblem (which constitutes a fine-grained approach). The second (coarse-grained) approach allocates a single core to each subproblem, but solves all subproblems in parallel. When using the second approach all the cores work independently on their own input data.

Clearly, since BBLAS focuses on very small matrix operations, the second approach is to be preferred. Numerous small matrices can fit in the cache which allows each core to work asynchronously on the subproblems: using fine-grained parallelism forces the cores to synchronise after each subproblem has been computed.

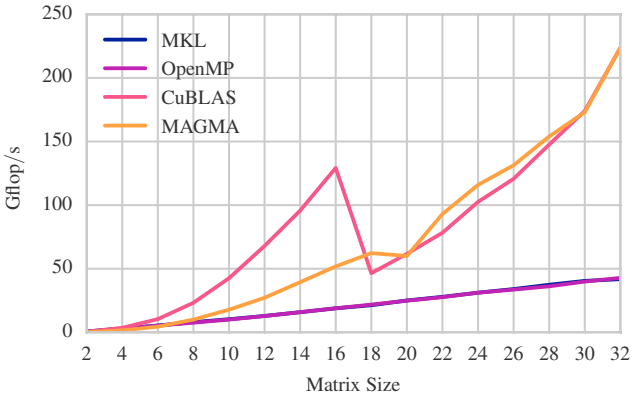


Figure 1: Performance of current libraries implementing batched DGEMM on batches of 10,000 matrices. All the matrices are square with the size denoted on the x-axis. The performance of MKL and OpenMP are very similar and cannot be easily distinguished.

In Figure 1 we perform a comparison of the current performance that can be obtained using Intel MKL, an OpenMP for loop, CuBLAS, and MAGMA[3] as a reference point for our future experiments. In particular we show the performance of computing a batch of 10,000 matrix-matrix multiplications in double precision arithmetic (DGEMM). The machine used in this experiment is a NUMA node with 2 sockets, using Intel Xeon CPU E5-2650 v3 chips (2.3GHz, Haswell architecture), for a total of 20 cores. The memory is interleaved between the two processors[4]. Both CuBLAS and MAGMA are run on a Kepler K40c GPU. Interestingly the OpenMP loop and Intel MKL have almost identical performance, whilst the two GPU implementations vary significantly.

All the matrices are chosen to have elements taken from a random uniform distribution on the interval $[0, 1]$. Note that throughout all of our experiments we ensure the cache of each processor is flushed before every invocation of a BBLAS operation, to avoid obtaining misleading performance results: by neglecting this step we can obtain performance results up to 4 times faster than those reported here in the cases where the data fits into cache memory. This is consistent with observations by Whaley and Castaldo [15].

Next, we briefly introduce two competing APIs for performing BBLAS operations being discussed within the linear algebra community. The APIs make a distinction between batches where all matrices are of the same size (called a "fixed batch") and batches where the matrices can vary in size (a "variable batch"). The reason for this distinction is that, in the fixed batch case, there are fewer parameters to check before computation can begin. A more detailed

[3] `http://icl.cs.utk.edu/magma/`
[4] Run with "`numactl --interleave=all`".

Table 1: Details of the architectures used throughout our experiments

Platform	Xeon E5-2650 v3	Xeon Phi KNL 7250	Kepler K40c
Cores	2×10 (2.3GHz)	68 (1.40GHz)	2880
On-chip Memory	L1 32KB (per core)	L1 32KB (per core)	SM 16/48KB (per MP)
	L2 256KB (per core)	L2 512KB (per core)	L1 48/16KB (per MP)
	L3 25MB (per socket)	MCDRAM 16GB	L2 1536KB (unified)
Main Memory	32GB DDR4	384GB DDR4	12GB GDDR5
Bandwidth	64 GB/s	115.2 GB/s	288 GB/s
Compiler	icc 16.0.0	icc 16.0.3	nvcc 7.5
BLAS	MKL 11.3	MKL 11.3	CuBLAS 7.0

explanation of each API, along with the corresponding calling sequences for both fixed and variable batch operations, can be found in [13].

The first API clearly distinguishes between fixed and variable batches by creating separate functions for the different batch types. Since fixed batches require less parameters the main benefit of this API is its simplicity, especially for the commonly used fixed batch operations. The major drawback is that a separate function is required for variable batch computation, exposing users to twice the number of library functions.

The second API, which we call the "group-based API", takes a different approach. This API is designed to facilitate the computation of multiple fixed batches from within one function call. Each fixed batch is called a "group" and we can operate on multiple groups using just one function call. Current versions of Intel MKL use this approach in their batched GEMM routines. The main problem with this interface is that it makes both fixed and variable batch operations more difficult. However, it could be useful for the situation where multiple fixed batch operations are required. It is not yet clear how much this would differ from making multiple calls to the fixed batch routine from the previous API.

2.1 Experiments using the group-based API

In this subsection we perform a number of experiments in order to clarify what benefit might be gained from adopting the group-based API as opposed to making multiple calls to a routine designed for fixed batches. Here we assume that the matrices have already been sorted into groups. The experiments are based upon those shown by Sarah Knepper (Intel) at the recent workshop on BBLAS, held at the University of Tennessee[5]. All the experiments in this subsection are performed on a NUMA architecture, the details of which can be found in the first column of Table 1.

The experiment in Figure 2 compares the difference in performance between running a batch DGEMM using one group of 10,000 matrices versus using 10,000 groups with one matrix in each, for a variety of matrix sizes. The Intel MKL function `dgemm_batch` is used to perform both of these operations. The idea of this experiment is to compare the best and worst possible scenarios for the group interface: when treating the computation as 10,000 groups with one matrix in each, the routine needs to perform more argument checks than when everything is collated in a single group. The reported Gflop rates are averaged over 10 runs.

[5]PDF slides available at `http://www.netlib.org/utk/people/JackDongarra/WEB-PAGES/Batched-BLAS-2016/`. Accessed on 9th February 2017.

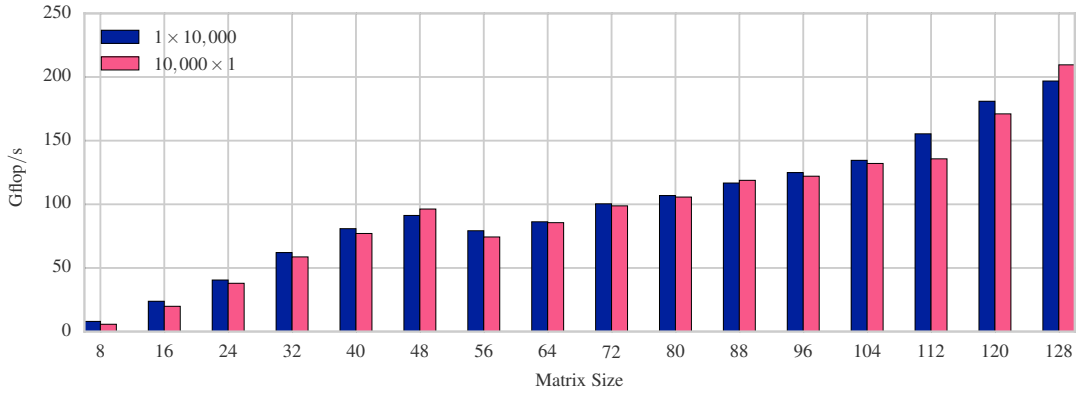


Figure 2: Performance of the MKL batched DGEMM on batches of 10,000 matrices using either one group of 10,000 ($1 \times 10,000$) or 10,000 groups of one ($10,000 \times 1$). All the matrices are square with the size denoted on the x-axis.

We see that the performance increases with the matrix size and that, generally, running the computation using just one group is slightly faster due to the reduced number of parameter checks. In most cases the difference in performance between the two routines is not very dramatic: the mean difference between the performance of the two approaches is 7.7%. Remember that running 10,000 groups containing one matrix each is essentially the worst possible case for the group API so that, with a less dramatic difference between the two groupings, we would expect the performance difference to be negligible.

However, we note that the time to sort the matrices into groups was not included in this experiment. If the matrices are generated at random and need to be fully sorted, then creating the groups can take longer than the computation itself. Therefore, to take advantage of the this API, applications should be revisited to generate matrices that can naturally be assigned to the different groups.

3 Impact of the Data Layout

Whilst we have discussed some of the features of the different APIs, another major issue is the way that the matrices are stored in memory. As our experiments will show, the memory layout is a critical part of obtaining good performance for BBLAS routines on modern hardware.

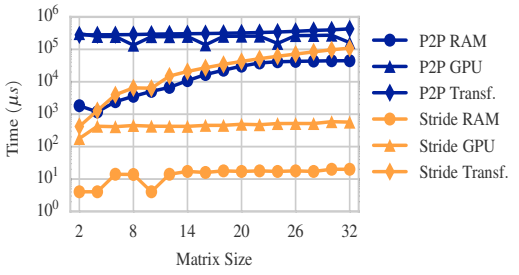
There are three main approaches to the data layout that we have identified. These are the pointer-to-pointer (P2P) layout, the "strided" data layout, and the "interleaved" layout.

The P2P data layout involves passing each BBLAS function arrays of pointers where each element of an array is a pointer to a memory location containing a matrix. The main benefit of this approach is its flexibility: it is very easy to add more matrices into the batch by simply appending their respective memory locations onto the array of pointers. However, the major drawback of this approach is that allocating memory for each matrix separately means the data will be scattered throughout the RAM. When performing the computation this means that the CPU will need to load memory from many different locations which is much slower than loading contiguous memory. This difference in memory access speed is much more apparent when we consider offloading computation to hardware accelerators. In this case, since the matrices are

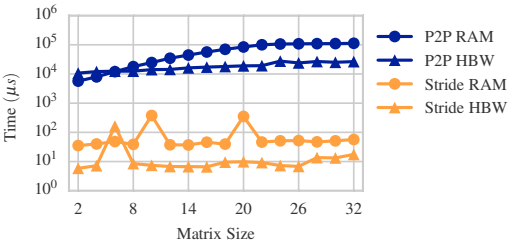
spread throughout the RAM, they must be sent separately which incurs a (comparatively large) latency cost.

One solution to this problem is to use the strided memory layout. For storing the matrices A_i in a fixed batch DGEMM, this involves creating one large array containing all the A_i in contiguous memory along with a `strideA` parameter which gives the number of memory locations between the matrices. For example, if `ptr` was a variable pointing to the first element of A_1 then `ptr + strideA` would be a pointer to the first element of A_2 and, in general, `ptr + (i-1)*strideA` would point to the first element of A_i.

The strided memory layout stores the matrices in contiguous memory and is therefore more efficient when allocating, loading into the CPU cache, and offloading to hardware accelerators as illustrated in Figure 3. However, we have lost the flexibility of the P2P approach. In order to add extra matrices to our batch we need to allocate an additional large block of contiguous memory locations and copy all the data across, which can be extremely expensive relative to the actual computation time.



(a) Evaluation of memory allocation cost on the Intel Xeon E5-2650 v3 (Haswell) and GPU (Kepler K40c) as well as the data transfer cost.



(b) Cost of memory allocation on RAM versus the fast MCDRAM of the Intel KNL. In the legend MCDRAM allocations are denoted by HBW (high-bandwidth).

Figure 3: Evaluation of the impact of the P2P and strided memory layouts on memory allocation (and data transfer for accelerators) on three architectures: NVIDIA GPU, Intel Xeon and Intel KNL. In each case, we use a fixed batch of 10,000 small matrices ranging from 2×2 to 32×32.

Alternatively we might consider a memory layout that consists of "interleaving" the matrices in memory. In this case we create one large array and fill it as follows. Firstly, store the first element of each matrix in turn, followed by the second element of each matrix etc. To illustrate this, if we have a batch of three matrices

$$D = \begin{bmatrix} d_{11} & d_{12} \\ d_{21} & d_{22} \end{bmatrix}, \quad E = \begin{bmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \end{bmatrix}, \quad F = \begin{bmatrix} f_{11} & f_{12} \\ f_{21} & f_{22} \end{bmatrix},$$

and work in column major order, then the strided and interleaved memory layouts will store the elements as follows.

- Strided:
  $[d_{11}, d_{21}, d_{12}, d_{22}, e_{11}, e_{21}, e_{12}, e_{22}, f_{11}, f_{21}, f_{12}, f_{22}]$
- Interleaved:
  $[d_{11}, e_{11}, f_{11}, d_{21}, e_{21}, f_{21}, d_{12}, e_{12}, f_{12}, d_{22}, e_{22}, f_{22}]$

From this we can see that the interleaved memory layout is simply a permutation of the strided memory layout. The reason that the interleaved format is interesting is that it may aid in

utilizing vectorization for very small matrices, a key component in obtaining high-performance in modern multi(many)-core architectures. For example, if the matrices are of size 2×2 but the length of the vector units is 8 double precision numbers, as in the self-hosted Intel KNL, then the strided and P2P memory layouts will not fill the vector units. By comparison, the interleaved memory layout can work on 8 matrices simultaneously to fill the vector units in each clock cycle. The current trend is for SIMD units to grow wider (e.g. the AVX-512 instruction set on Intel KNL), so maximizing vectorization is critical to obtaining good performance.

One drawback to the interleaved approach is the additional complexity of the format. This means that either the programmer must interleave their matrices manually—requiring significant code refactoring—or the software library automatically converts matrices in P2P format to interleaved format for performance gains, which costs both time and extra memory. In the next section we start all experiments from the P2P format and convert matrices into interleaved format within our routines: excluding the cost of this conversion makes our routines approximately twice as fast.

4 Interleaved data layout performance

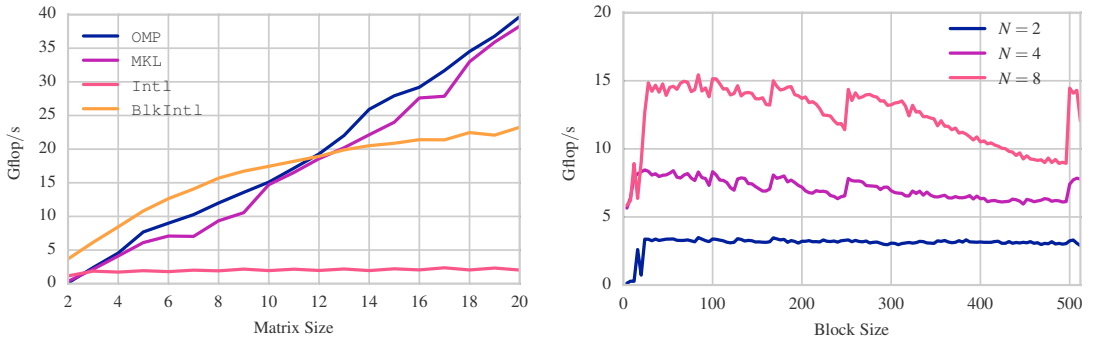
As mentioned previously, the main motivation for considering the interleaved memory layout was to aid vectorization, and therefore performance, for small matrices. In this section we perform experiments to substantiate that claim, using a NUMA node with 20 CPU cores, a self-hosted Intel KNL with 68 CPU cores, and a Kepler K40c GPU.[6] Further detail on these architectures is given in Table 1.

We will use two versions of the interleaved data format in our comparisons, both parallelized for CPU cores using OpenMP. The first version, as explained in the previous section, stores the first element of each matrix followed by the second element etc. One issue is that, for large batch sizes, the machine needs to make large jumps in memory to access the different elements of the matrices (leading to cache misses). For example, moving from the first to the second element of a matrix in the batch requires jumping `batch_count` memory locations, which can hinder the overall performance. We will refer to this "fully interleaved" approach as `Intl`.

To alleviate the impact of the jumps in memory we also use a "block interleaved" approach. In this memory layout we allocate one large chunk of memory and begin by interleaving the first k matrices, where the block size k is a tunable parameter. We then interleave the next k matrices in the next block of consecutive memory and so on, until all matrices have been stored. If `batch_count` is not a multiple of k then there will be a small amount of unused memory at the end of the final block. This is merely another permutation of the interleaved (and hence strided) memory layouts which attempts to balance the vectorization of the interleaved approach with the smaller memory jumps required in the strided memory layout. More detail on choosing the optimal block size is described separately for the Intel KNL and GPU architectures later in this section. This approach will be referred to as `BlkIntl`.

For each architecture we compare the two interleaved memory approaches to the P2P memory layouts. We compare the performance of each approach by computing a batch of 10,000 matrix-matrix multiplications. We will focus firstly on the NUMA node architecture. For the `Intl` approach we use OpenMP to loop over the different matrices in the batch meanwhile, for the `BlkIntl` approach, we use OpenMP to loop over the blocks, allocating one core to each block. The optimal block size k will likely depend on the number of cores and the memory

[6] The interleaved DGEMM code used to run these experiments can be found at `https://www.github.com/srelton/bblas_interleaved` and `https://github.com/pedrovalerolara/CUDA_BATCH_DGEMM_INTERLEAVED` for Xeon Phi and GPU implementations, respectively.



(a) Performance when matrices vary in size from 2×2 to 20×20.

(b) Effect of the block size with matrices of size 2, 4, and 8, respectively.

Figure 4: Performance results for batched DGEMM using a batch of 10,000 square matrices on a NUMA node.

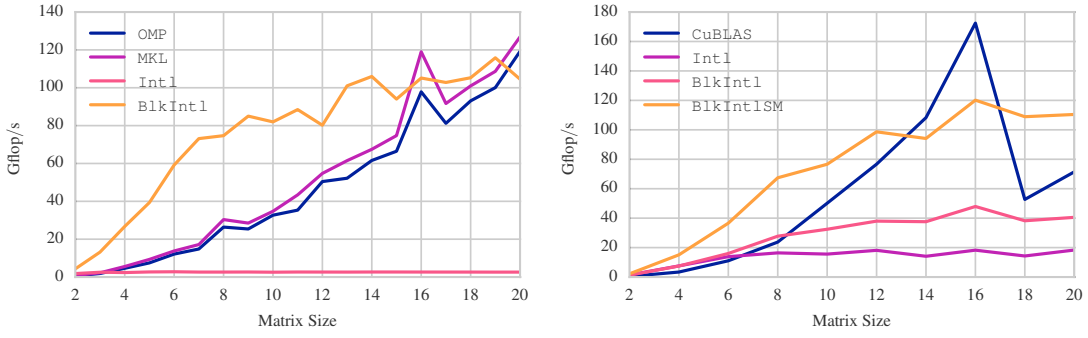
hierarchy so, for each invocation of the algorithm presented below, we will tune the block size to give the optimal performance.

We compare the two interleaved memory formats to two implementations using the P2P memory layout. The first P2P memory layout is Intel MKL's `cblas_dgemm_batch` function using the group-based API; whilst the second is a simple for loop over the batch of matrices (using OpenMP and single threaded `cblas_dgemm`) which uses a single-core to compute each DGEMM operation. We will refer to these approaches as `MKL` and `OMP`, respectively.

Figure 4a shows the results of this experiment. We see that the `Intl` approach is better than `MKL` and `OMP` for 2×2 and 3×3 matrices but quickly hits a limit of 2.5 GFlop/s. The large jumps in memory needed to access different elements of the matrices are the cause of this limit. Meanwhile the `BlkIntl` approach is superior to `MKL` and `OMP` for matrices until size 12×12. Furthermore, in Figure 4b we show the effect of the block size k on `BlkIntl` for matrices of size 2×2, 4×4, and 8×8. Whilst the block size has little impact on the performance for matrices of size 2×2, we can see that it has a significant impact for larger matrix sizes and must be tuned carefully depending on the architecture and matrix size. In general, over all matrix sizes, we typically 72 and 168 to be the optimal block sizes.

The next architecture we consider is the Intel KNL using the high-bandwidth MCDRAM. As before, we use OpenMP to loop over the matrices in the `Intl` approach and to loop over each block in the `BlkIntl` approach. We found that 80 and 152 were often the optimal block sizes, which are fairly similar to those seen in the NUMA architecture. We compare against `MKL` and `OMP` as described in the previous architecture.

The performance that we obtain is given in Figure 5a. For 2×2 and 3×3 matrices both interleaved approaches are faster than the P2P routines, but after this the `Intl` approach hits a limit of 2.5 Gflop/s. As before, this is due to the large jumps in memory that are required to access different elements of the matrix. Meanwhile, `BlkIntl` quickly attains its peak performance of around 100 Glop/s and is significantly faster than both P2P approaches until matrices of size 16×16 are used; shortly after this point the P2P approaches become faster. This peak performance of 115 Glop/s for the `BlkIntl` approach could be improved by prefetching the required data more efficiently. Indeed, both the `BlkIntl` and `MKL` approaches are performing the same operations in a different order, so obtaining good performance is merely



(a) Self-hosted Intel KNL

(b) Kepler K40c GPU

Figure 5: Performance results for batched DGEMM using a batch of 10,000 square matrices varying in size from 2×2 to 20×20.

a race to fill the vector units of the cores as fast as possible.

The GPU experiments are depicted in Figure 5b. The initial CUDA kernels of the `Intl` and `BlkIntl` do not make use of shared memory. We designed another block interleaved kernel (`BlkIntlSM`) taking advantage of the shared memory with a slight change in the memory hierarchy to use 48KB of shared memory and 16KB of L1 cache. This kernel outperforms cuBLAS for very small matrices up to 13×13. We believe that this approach has potential for GPU architectures and further tuning will enable better performance.

5 Conclusions

The main contribution of this paper is the experiments which show the performance gains the interleaved approach offers. We have summarised the current API and memory layout choices used for batched BLAS operations and performed a number of experiments to compare their efficiency. In particular, we found that the block interleaved memory format gives extremely promising performance for batches of small matrices over a range of architectures including a NUMA node, a Kepler K40c and the self-hosted Intel KNL. The peak performances we obtained were 24, 120, and 115 Gflop/s, respectively; and we have shown how our approach performs faster than vendor libraries when operating on batches of small matrices.

Although this work focused on GEMM the analysis, and particularly the interleaved memory approach, is also applicable to other BLAS kernels. In fact, most other level 3 BLAS operations, except for TRSM (triangular solve), can be viewed as specialized GEMM operations so we expect similar performance increases.

In the future it would be interesting to see the effect that interleaved memory layouts can have on the performance of batched TRSM operations. In particular, each column of the matrices requires one division so, by using the interleaved memory layout, we can perform large numbers of these divisions simultaneously and make more efficient use of the vector units in a CPU core. One could also extend batch operations to include LAPACK routines, for example batched LU and QR-based solvers are already available in CuBLAS. Extending our block interleaved approach to LAPACK routines could lead to significant performance increases.

Finally, these analyses and the comparisons between different APIs are vital in helping the community decide upon a standard interface for batched linear algebra operations which may

a have profound impact on the future of HPC, as the original BLAS standard certainly has.

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] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180(1), 2009.

[3] Emmanuel Agullo, Luc Giraud, and Mawussi Zounon. On the resilience of parallel sparse hybrid solvers. In *22nd IEEE International Conference on High Performance Computing, HiPC 2015, Bengaluru, India, December 16-19, 2015*, pages 75–84, 2015.

[4] 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.

[5] Jack Dongarra, J. Du Croz, Iain Duff, and Sven Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16:1–28, 1990.

[6] Jack Dongarra, J. Du Croz, Sven Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14:1–32, 1988.

[7] Iain Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Trans. Math. Softw.*, 9(3):302–325, 1983.

[8] Nicholas J. Higham and Vanni Noferini. An algorithm to compute the polar decomposition of a 3×3 matrix. *Numer. Algorithms*, 73(2):349–369, 2016.

[9] Ali Khodayari, Ali R. Zomorrodi, James C. Liao, and Costas D. Maranas. A kinetic model of Escherichia coli core metabolism satisfying multiple sets of mutant flux data. *Metabolic Engineering*, 25:50–62, 2014.

[10] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krough. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Softw.*, 5:308–323, 1979.

[11] Karl Ljungkvist. Matrix-free finite-element operator application on graphics processing units. In *Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II*, pages 450–461, 2014.

[12] O. E. B. Messer, J. A. Harris, S. Parete-Koon, and M. A. Chertjkow. Multicore and accelerator development for a leadership-class stellar astrophysics code. In *Proceedings of "PARA 2012: State-of-the-Art in Scientific and Parallel Computing."*, 2012.

[13] 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.

[14] Pedro Valero-Lara, Alfredo Pinelli, and Manuel Prieto-Matias. Fast finite difference Poisson solvers on heterogeneous architectures. *Computer Physics Communications*, 185(4):1265–1272, 2014.

[15] R. Clint Whaley and Anthony M. Castaldo. Achieving accurate and context-sensitive timing for code optimization. *Softw. Pract. Exper.*, 38(15):1621–1642, 2008.