Parallel Computing 81 (2019) 1–21

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Algorithms and optimization techniques for high-performance matrix-matrix multiplications of very small matrices

I. Masliah [c,*], A. Abdelfattah [a], A. Haidar [a], S. Tomov [a], M. Baboulin [b], J. Falcou [b], J. Dongarra [a,d]

[a] Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA
[b] University of Paris-Sud, France
[c] Inria Bordeaux, France
[d] University of Manchester, Manchester, UK

ARTICLE INFO

Article history:
Received 23 September 2017
Revised 10 September 2018
Accepted 16 October 2018
Available online 17 October 2018

Keywords:
Matrix-matrix product
Batched GEMM
Small matrices
HPC
Autotuning
Optimization

ABSTRACT

Expressing scientific computations in terms of BLAS, and in particular the general dense matrix-matrix multiplication (GEMM), is of fundamental importance for obtaining high performance portability across architectures. However, GEMMs for *small matrices* of sizes smaller than 32 are not sufficiently optimized in existing libraries. We consider the computation of many small GEMMs and its performance portability for a wide range of computer architectures, including Intel CPUs, ARM, IBM, Intel Xeon Phi, and GPUs. These computations often occur in applications like big data analytics, machine learning, high-order finite element methods (FEM), and others. The GEMMs are grouped together in a single *batched* routine. For these cases, we present algorithms and their optimization techniques that are specialized for the matrix sizes and architectures of interest. We derive a performance model and show that the new developments can be tuned to obtain performance that is within 90% of the optimal for any of the architectures of interest. For example, on a V100 GPU for square matrices of size 32, we achieve an execution rate of about 1600 gigaFLOP/s in double-precision arithmetic, which is 95% of the theoretically derived peak for this computation on a V100 GPU. We also show that these results outperform currently available state-of-the-art implementations such as vendor-tuned math libraries, including Intel MKL and NVIDIA CUBLAS, as well as open-source libraries like OpenBLAS and Eigen.

1. Introduction

The available parallelism to exploit in today's computer architectures is pervasive—not only in systems from large supercomputers to laptops, but also in small portable devices like smartphones and watches. Along with parallelism, the level of heterogeneity in modern computing systems is also gradually increasing. Multi-core CPUs are combined with discrete high-performance GPUs, or even become integrated parts as a system-on-chip (SoC) like in the NVIDIA Tegra mobile family of devices. Heterogeneity makes the parallel programming for technical computing problems extremely challenging, especially in modern applications that require fast linear algebra on many independent problems that are of size 100 and smaller. According to a recent survey among the Scalable Linear Algebra PACKage (ScaLAPACK) and Matrix Algebra on

* Corresponding author.
E-mail address: ian.masliah@lip6.fr (I. Masliah).

https://doi.org/10.1016/j.parco.2018.10.003
0167-8191/

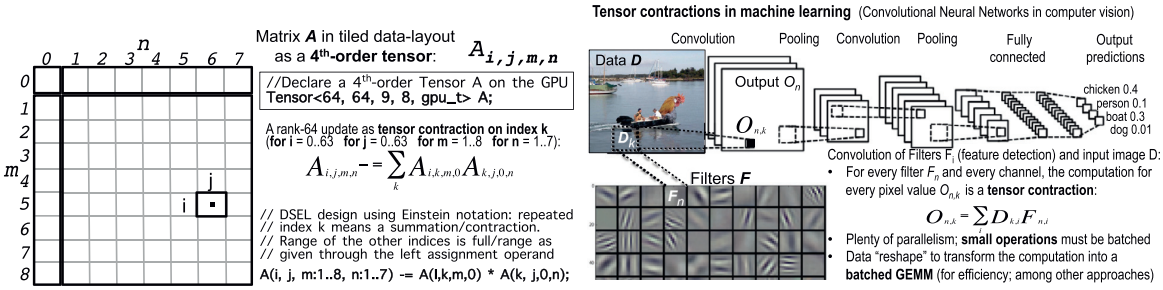


Fig. 1. Left: Example of a 4th-order tensor contractions design using Einstein summation notation and a domain-specific embedded language (*DSEL*). **Right**: Illustration of batched computations needed in machine learning.

GPU and Multicore Architectures (MAGMA) [1] users, 40% of the respondents needed this functionality for applications in machine learning, big data analytics, signal processing, batched operations for sparse preconditioners, algebraic multigrid, sparse direct multi-frontal solvers, QR types of factorizations on small problems, astrophysics, and high-order finite element methods (FEM). At some point in their execution, applications like these must perform a computation that is cumulatively very large and which often needs to run on large-scale distributed memory systems, but the individual parts of which are very small; when such operations are implemented naively using the typical approaches, they perform poorly. To address these challenges, there are efforts in the community to extend the basic linear algebra subprograms (BLAS) standard to include API for Hybrid Batched BLAS [2], as well as to develop innovative algorithms [3], data and task abstractions [4], and high-performance implementations based on the standard. Some of these efforts have been released as examples through the MAGMA library since version 2.0 [5,6]. Fig. 1 illustrates how the need for batched operations and new data types arises in areas like linear algebra (Left) and machine learning (Right). The computational characteristics in these cases are common to many applications where the overall computation is very large but is made of operations of interest that are generally small. The small operations must be batched for efficiency and various transformations must be explored to cast them to regular, and therefore efficient, to implement operations, like GEMMs. This is the case in a number of applications that are currently of great interest, like data analytics and machine learning, where *tensor* data structures and APIs are used to represent higher-dimension multi-linear relations data; but still, for high performance and efficiency the computation is flattened to linear algebra on two-dimensional matrix data through Batched GEMMs [4].

There is a lack of sufficient optimizations on the Batched GEMMs that we target in this paper and that are needed in a number of applications. We show that this is the case through a theoretical performance analysis and a comparison between the results from the techniques introduced in this paper and vendor libraries like cuBLAS for NVIDIA GPUs, and MKL for Intel multi-core CPUs, as well as comparison to the open-source library called Eigen [7]. Related work on GEMM and its use for tensor contractions [4] target only GPUs and for very small sizes (16 and smaller). Batched GEMM for fixed and variable sizes in the range of 1000 and smaller were developed in [8]. The main target here is Batched GEMMs for multi-core CPUs, ARM, Intel Xeon Phi, and GPU architectures on matrices of sizes up to 32. In a preliminary study [9], we have investigated this problem and have laid out some of the ideas on algorithms and optimization techniques needed to accelerate them on modern architectures. In this paper, we extend these preliminary results by completing the algorithmic work, providing further details and extended functionalities, as well as generalizing the approach and the portability of the developments. We design a generic framework that incorporates all developments: the framework auto-generates kernels for every new architecture and autotunes them to find the best performing kernels. While we produce a single tool, the best kernels for different architectures and sizes are different, incorporating different optimization techniques, algorithms, and tuning parameters, which we highlight and analyze in this paper. Performance results are updated and include IBM Power8 processors and newer GPU architectures, e.g., the V100 GPU, We also add results and analysis on the performance differences and comparison to the Eigen library. A direct motivation for this work came from the need to accelerate large-scale, distributed-memory solvers in applications using high-order finite element discretizations, where tensor contraction computations are cast as Batched GEMMs [4].

2. Main contributions

The rapid advancements in semiconductor technologies today are dramatically transforming the balance of future computer systems, producing unprecedented changes at every level of the platform pyramid and the software stack. There are three challenges that stand out for numerical libraries and the myriad of applications that depend on them: (1) the need to exploit unprecedented amounts of parallelism; (2) the need to maximize the use of data locality and vectorized operations; and (3) the need to cope with component heterogeneity and portability. Below, we highlight our main contributions related to the algorithm's design and optimization strategies for addressing these challenges on multi-core CPU (Intel, ARM, IBM), Xeon Phi, and GPU architectures.

2.1. Algorithmic designs to exploit parallelism and vectorization

As clock frequencies are expected to stay near their current levels, or even decrease to conserve power, the primary method of increasing computational capability of a chip will be to dramatically increase the number of processing units (cores). This in turn will require an increase of orders of magnitude in the amount of concurrency that routines must be able to utilize. It will also require increasing the computational capabilities of the floating-point units by extending them to the classical Streaming single instruction, multiple data (SIMD) Extensions set (SSE-1, to SSE-4 in the early 2000s, and recently to Advanced Vector Extensions AVX, AVX-2, AVX-512). We developed specific optimization techniques that demonstrate how to use the many cores (currently, 10–20 multi-socket cores for the Haswell CPU, 4 cores for a Cortex A57 processor [NEON SIMD], 10 cores for the POWER8 processor [Altivex VMX SIMD], 68 cores for an Intel Knights Landing [KNL] 7250 and 56×64 CUDA cores for the Tesla P100 GPU) to get optimal performance.

2.2. Performance-portable framework for batched GEMMs

We developed a performance-portable framework by binding the architecture-specific developments into a single generator that is combined with autotuning to empirically find the best performing kernels, up to exploring a predefined design search space. While this produces a single tool, the best kernels for different architectures and sizes are different, incorporating different optimization techniques, algorithms, and tuning parameters. The optimization techniques, the algorithms, and the overall framework can be used to develop other batched Level 3 BLAS kernels and to accelerate numerous applications that need linear algebra on many independent problems.

2.3. Hierarchical communications that maximize data locality and reuse

Time per floating-point operation (FLOP), memory bandwidth, and communication latency are all improving, but at exponentially different rates [10]. Therefore, computations on very small matrices, which can be considered compute bound on old processors, are becoming communication-bound today and in the future—and will, consequently, depend more on the communication between levels of the memory hierarchy. We demonstrate that performance is indeed harder to achieve on new many-core architectures unless hierarchical communications and optimized memory management are considered in the design. We show that our implementations reach optimal performance only after we developed algorithmic designs that feature multi-level blocking of the computations and use multi-level memory communications.

2.4. Performance analysis and autotuning

We derive theoretical maximal performance bounds that could be reached for computation on very small matrices. We studied various instructions and performance counters, as well as proposed a template design with different tunable parameters in order to evaluate the effectiveness of our implementation and optimize it to reach the theoretical limit. The best for performance parameters are architecture-specific and were derived through an empirical autotuning process, yielding an approach to performance portability across the architectures of interest.

3. Performance model for batched GEMMs

To evaluate the efficiency of our algorithms, we derive theoretical bounds for the maximum achievable performance $P_{\max} = F/T_{\min}$, where F is the number of operations needed by the computation $F = 2n^3$, and $T_{\min}$ is the fastest time to solution. For simplicity, consider $C = \alpha AB + \beta C$ on square matrices of size n. In an ideal case, where we assume that there is overlap between computation and communication, the $T_{\min}$ becomes,

$$T_{min} = max(T_{Read(A,B,C)} + T_{Write(C)}, T_{Compute(C)})$$

Let β define the maximum achievable bandwidth in bytes/second and P_{peak} the peak performance that the system can provide. We have to read three matrices, A, B, and C and write back C—that's $4n^2$ elements ($\rho 4n^2$, where ρ is the precision size in bytes), or $8 \times 4n^2$ bytes for double precision (DP). As a consequence, in double precision, after dividing by β, $T_{Read(A,B,C)} + T_{Write(C)}$ is equal to $32n^2/\beta$. The time to perform the computation $T_{Compute(C)}$ can be defined by $T_{Compute(C)} = \frac{2n^3}{P_{peak}}$. Since on most of today's machines the ratio of the peak performance to the bandwidth is very large— >30 for most of today's CPUs or GPUs—we can easily deduce that $T_{Compute(C)} \to 0$ compared to $32n^2/\beta$ and thus $T_{\min} \approx 32n^2/\beta$ in double precision. Note that this is the theoretically achievable peak performance if the computation totally overlaps the data transfer and does not disrupt the maximum rate B of read/write to the GPU memory. Thus,

$$P_{max} = \frac{2n^3\beta}{32n^2} = \frac{n\beta}{16} \text{ in DP.}$$

The achievable bandwidth can be obtained by benchmarks. For our measurements, we used the Sustainable Memory Bandwidth in High Performance Computers (STREAM) benchmark [11] and the Intel Memory Latency Checker 3.0 tool for CPU. We also used NVIDIA's `bandwidthTest` and a set of microbenchmarks that we developed for GPU. For example, our tests

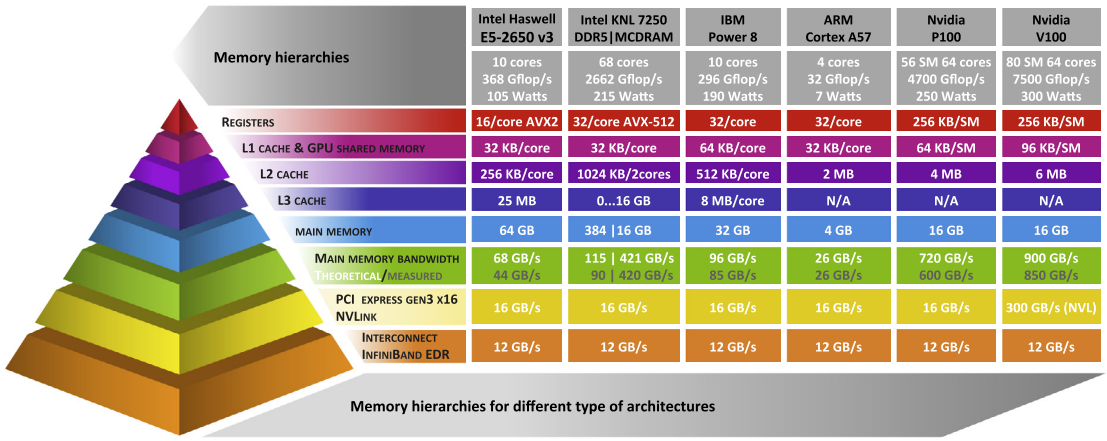


Fig. 2. Memory hierarchies of the experimental CPU and GPU hardware.

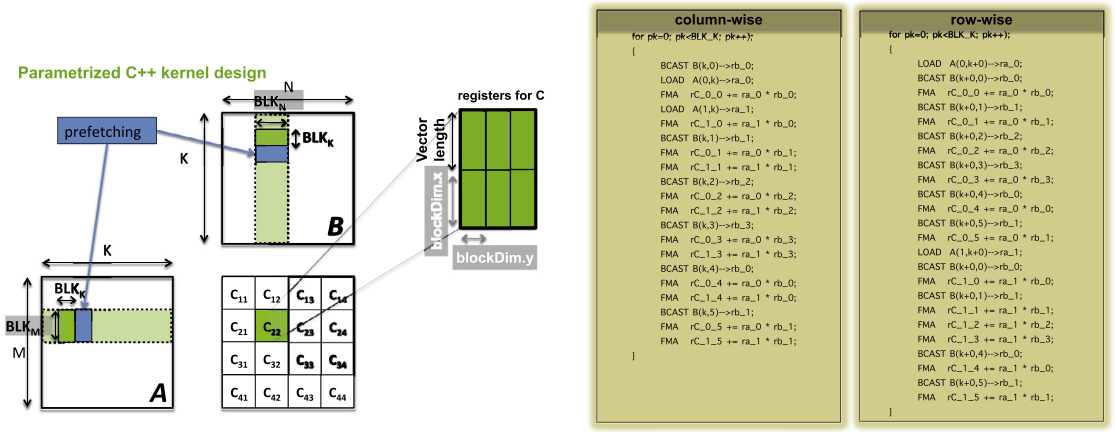
show that the best practical CPU bandwidth we are able to achieve on a 10-core Intel Xeon E5-2650 v3 processor (Haswell) using different benchmarks is about 44GB/s per socket. On the Intel Xeon Phi KNL 7250 system, the maximal achievable bandwidth is 92GB/s when the data is allocated in the DDR4 and about 420GB/s for data allocated in the MCDRAM. On the IBM POWER8 system (one socket), the measured bandwidth from the benchmark was about 85GB/s, while on the ARM Cortex A51 it was measured about 26GB/s (one socket of 4 cores). On the Tesla P100 GPU, the peak is 600GB/s. The curves representing these theoretical maximal limits for the different architectures are denoted by the "upper bound" lines in our performance graphs, e.g., see Figs. 8 and 18a.

4. Experimental hardware

All experiments are done on an Intel multi-core system with two 10-core Intel Xeon E5-2650 v3 (Haswell) CPUs, a 4-core Cortex A57 ARM CPU, two 10-core IBM Power8 CPUs, a 68-core Intel Knights Landing CPU 7250 and a Pascal Generation Tesla P100 GPU, and the newest Volta V100 GPU. Details about the hardware are illustrated in Fig. 2. We used GNU Compiler Collection (GCC) compiler 5.3 for our Xeon code (with options -std=c++14 -O3 -mavx2 -mfma), as well as the icc compiler from the Intel suite 2018, and the BLAS implementation from the Math Kernel Library (MKL) [12]. We used the XLC compiler 13.01 for our PowerPC code and the Engineering and Scientific Subroutine Library (ESSL) BLAS library from IBM. On the ARM processor, we used an OpenBLAS version optimized for the Cortex A57 with GCC 5.3. We used CUDA Toolkit 8.0 for the GPU. For the CPU comparison with the MKL library we used two implementations: (1) An Open Multi Processing (OpenMP) loop statically or dynamically unrolled among the cores (we choose the best results), where each core computes one matrix-matrix product at a time using the optimized sequential MKL `dgemm` routine, using the option -DMKL_DIRECT_CALL_SEQ and (2) The batched dgemm routine that has been recently added to the MKL library.

5. The kernel auto-generation and autotuning process

The `dgemm` kernel is parameterized and implemented using C++ features, including templates and overloaded functions. The kernel design for small matrix sizes is illustrated in Fig. 3a. The matrix C is split into blocks C_{ij} of size $BLK_M \times BLK_N$ that can be computed in parallel. The idea is that since C is where the computations are accumulated and the final result written, it is better to keep as large a part of C as possible in registers during the accumulation of the multiplication. Note that this one-level design of blocking is especially designed for small matrices; for larger matrices, a design with multiple levels of blocking may be better in order to account for blocking on the possibly multiple levels of the architecture's memory hierarchy layers. Any particular block C_{ij} of C will be held in registers for either the CPU or GPU case. The number of rows in C_{ij} is better to be multiple of the vector length for CPUs, or multiple of the number of threads in the "x" dimension for GPUs. Also, the number of columns will be dependent on the available registers (CPUs or GPUs) and on the number of threads in the "y" dimension for the GPU case. There is a sliding window of size $BLK_M \times BLK_K$ that reads data of the matrix A and, similarly, a sliding window of size $BLK_K \times BLK_N$ that reads data from the matrix B. This data can be read into register or into cache (shared memory or register in case of the GPU kernel). The innermost loop will multiply the green portion of A and B and will accumulate the result into the green portion of C. Note that the blue portion of A and B corresponds to the prefetching when it is enabled by the kernel generator (the kernel generator will generate two kernels w/o prefetching). The windows of A and B slide horizontally and vertically, respectively, and once finished, the block of C contains the final results of $A \times B$. This result is multiplied by α (when α is not equal to one) and added to the corresponding block of the matrix C (loaded from the main memory and multiplied by β—when β is not equal to one—before the addition, and the



(a) Parameterized C++ `dgemm` kernel design

(b) Auto-generated code for the `dgemm` kernel for the same configuration but with two different inner loop order.

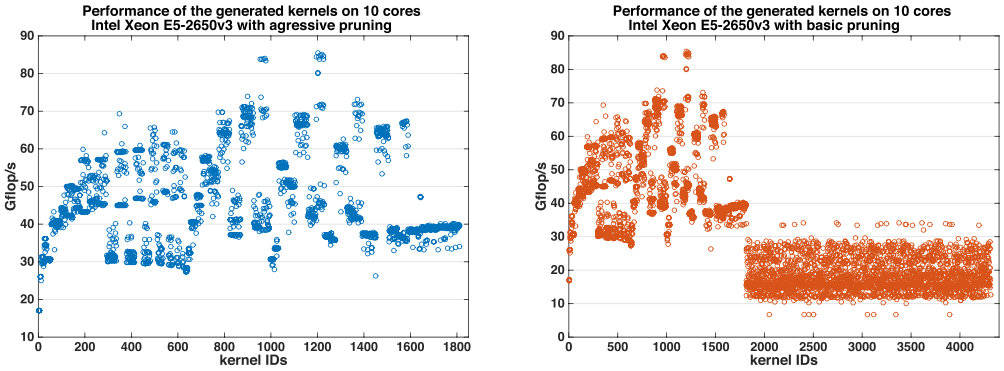
Fig. 3. Design and auto-generation of the `dgemm` kernel.

result is stored back into the main memory). If β is zero, the results of the multiplication are directly stored into the main memory.

The same methodology applies when any of the matrices is transposed, and the code generation is always handled automatically. C_{ij} is always of size $BLK_M \times BLK_N$ and the reading of A and B always happens following the block design (e.g., contiguous block of the size $BLK_M \times BLK_K$ and $BLK_K \times BLK_N$, resp., for the Non-Transpose). As a result, the transpose is implicitly coded through the innermost loop when the data is already in cache. Moreover, the description here was provided for square matrices, but the same applies for rectangular matrices as well. The matrix C is always split over blocks, and therefore the case of rectangular matrices can be generalized to follow the same methodology. This is also valid for the GPU implementation. We also note that, since the read/store happens by block, a matrix stored in row-major format can also be handled by the same techniques. In this case, the window slides vertically on A and horizontally on B. It can also be handled by flipping the operations from non-transpose to transpose. For example, if the matrix A is the only matrix stored in row-major and the operation is $C = A \times B$, then this can be computed by the $C = A^T \times B$ kernel where A is considered stored in column-major format.

The ultimate goal is to explore all possible kernel configurations, called "*the autotuning search space*," and provide a clear description of the kernel generation and the autotuning process to be performed in order to get the best performance. As described above, for every architecture, there might be a very large number of possibilities for designing the matrix-matrix multiplication kernel. If we take for example an 8×8 matrix, on a hardware that has 16 256-bit AVX-2 registers, we can decide to hold all of B in registers and keep loading/reloading A and C, or we can decide to use only 8 registers to hold a portion of B and minimize the number of loads/reloads on A and C, and so on. The same scenario will be applicable to C and to A. Thus, the decision of how many registers we must dedicate to each array (e.g., A, B, and C) can generate many configurations (about a thousand). Furthermore, one configuration might be good for one matrix size but bad for other matrix sizes. In addition to that, there is the loop order: should the innermost loop go "*row-wise*" or "*column-wise*," should we implement the *ijk, ikj, kij*, or other loop orders? Thus, for every loop order configuration, since we have about one thousand configurations for the registers, one might end up with about ten thousand configurations. This is what makes up the search space. Then, in order to exploit such a large search space of possibility in the shortest time, we apply an aggressive pruning technique to reduce it. A condition of the pruning is that only the kernel configurations that have absolutely no chance of achieving good performance be eliminated.

A commonly used technique by performance engineers is to apply some rule of thumb constraints when tuning or designing kernels. For example, when designing how many registers to reserve for C or B, one can let the possibility always be an even value, e.g., to correlate it to the hardware specifications. In our work, we want to replace these kinds of rules with a set of derived constraints that have a direct relation to performance. Based on our analysis, we can define many of these constraints. One of the best examples is the occupancy of the computational unit, which is a function of multiple variables. For example, for CPUs it includes: the instruction order, the vectorized instruction, the amount of reuse, the contiguous vs. noncontiguous read/store of data from/to the cache. The occupancy of the computational unit is also an important factor in the autotuning and the kernel generation phase for GPUs as well. It also depends on many variables such as the number of threads in a block, the number of registers required by each thread and the amount of shared memory required by each block. More details on the autotuning process for GPU are described in the next section. Occupancy threshold is a very effective and safe pruning constraint, as most kernels have no chance of achieving good performance at low occupancy



(a) Tuning of the dgemm kernel using the aggressive pruning strategy.

(b) Tuning of the dgemm kernel using the basic hardware constraint pruning.

Fig. 4. Auto-generation and tuning process of the dgemm kernel on 10 cores Intel Xeon E5-2650 v3 processors.

levels. One example is if a kernel uses only 3 or 4 registers of the 16 available registers. Another very important factor for very small computations is the way data is loaded or prefetched from the main memory into the cache. In particular, transferring data without using vectorized load/store instructions is going to dramatically affect the performance. As studied in Section 3, working with small matrices is strongly correlated to the transfer of the data between the cache levels and the main memory. Thus, having vectorized load/store instruction is a major factor in the design, and thus any configuration that does not exploit vectorized load/store instructions is considered to lead to low performance and, as a result, can be dropped from the search space. These constraints, along with others, have been integrated into our pruning process, which happens during the configuration generation of the autotuner and kernel design phase.

In Fig. 4, we show the tuning results on 10-core Intel Xeon E5-2650 v3 processors. On the left, in Fig. 4a, we illustrate the performance obtained from each of these kernels on a matrix of size 32×32. There were about 1812 kernels generated. It can be observed that there are two small sets of kernel configurations that provided performance around 85 gigaFLOP/s, while most of the other kernels provided performances somewhere in between 30 gigaFLOP/s and 75 gigaFLOP/s. When we checked the configuration details of these two sets, we found that they are very similar in terms of the blocking sizes (BLK_M, BLK_N, and BLK_K) and the outer loop order, while the difference was only of using $+/-$ 1 or 2 registers on the reading of A and B, or a slightly different order of the fused multiply–add (FMA) instructions in the generated code.

In order to emphasize the benefit of the automated tuning and the pruning process, we deactivated the aggressive pruning and only left the hardware constraints. This increased the number of possible configurations by about 2500. We ordered these extra configurations at the end of the previous configurations and wanted to verify that our pruning process was safe and accurate. We represent the performance obtained by the tuning process of this new list of configurations with basic pruning in Fig. 4b. As can be seen, all the extra configurations provided very low performance, ranging from 5 gigaFLOP/s to 35 gigaFLOP/s (see kernels with IDs in the range from 1812 to 4326). This highlights the advantage of our pruning process and the importance of the analysis that we outlined above.

We mention that a similar pattern has been observed for tuning on other architectures such as the IBM POWER8, the KNL or the GPUs. We also indicate that the entire generation and tuning process is automated and requires dozens of hours per architecture to finish.

Because our design is parameterized, once all the possible and acceptable configurations are created, the kernel generator creates one or many kernels for every configuration. For every configuration, the difference between the kernels can be the fashion of the innermost loop, e.g., "*row-wise*" or "*column-wise*," the whole nested loop order (e.g., *ijk, ikj, kij*, etc.), the instruction order, etc. For example, a configuration specifies the blocking sizes (BLK_M, BLK_N, and BLK_K) and the number of registers allocated for each variable *A, B*, and *C*. Then, the generator creates many possible kernels for this configuration. An example of two CPU generated kernels for the same configuration (2 registers for *A*, 4 registers for *B*, and 6 registers for *C*) is depicted in Fig. 3b.

This new flexible and automated design for code and configuration generation enables us to easily design kernels for any architecture and to tune them and find the best kernel for each. This automated design did not exist in our previous work where we had to have different code snippets for every architecture and then tune it. Furthermore, we were able to extract from this tuning process the best configuration for these small sizes and write a parameterized C++ code for prefetch and loop unrolling on CPUs, which we describe next.

6. Programming model, performance analysis, and optimization for CPUs

The overall design fits the general description given in Section 5. However, there are specifics for the CPU and GPU cases. Here we provide in more detail the specifics for our CPU considerations, design, and optimizations.

```
// x is the type of 7 : int
auto x = 7;
```

Listing 1. C++ auto.

```
// x is of the type of expression
auto x = expression;
```

Listing 2. C++ generic auto.

```
constexpr long long fibonacci(const int x)
{
    return x <= 1 ? 1 : fibonacci(x - 1) + fibonacci(x - 2);
}
```

Listing 3. C++ constexpr.

In order to design a framework that has better code re-usability and adaptability, our overall designs and software construction choices include the use of new features of C++. By using advanced template techniques we can create high-level interfaces [13] without adding any cost, even for small matrix-matrix products. To do so, we designed a batch structure which contains a C++ vector for the data and static dimensions. By using the C++ `constexpr` keyword and integral constants we developed a generic batched code that dispatches at compile time the correct version depending on the size of matrices. We use this environment for each code sequence that we generate.

6.1. Programming techniques using C++14

The development of programming languages and their use have dramatically changed in recent years, leading to continuous evolution. C++ is an example of such a programming language. The cause of these changes is the need for higher-level language that provides better idioms for generic and generative programming and support for parallel computing. Here we discuss the new features of the C++14 standard that we use to develop our matrix-matrix product.

The first feature of the C++14 language that we discuss is `auto` [14]. Consider the following declaration in Listing 1:

Here x will have the type `int` because it is the type of its initializer. In general, we can write the code in Listing 2 and x will be of the type from the value expression in Listing 2. For any variable, `auto` specifies that the type of the variable that is being declared will be automatically deduced from its initializer. This allows us to write high-level, complex code without having the burden of complex types that can appear. We can apply the `auto` keyword on several features of the C++ language.

Another important feature of the C++14 standard is the `constexpr` keyword [15]. The `constexpr` keyword provides a mechanism that can guarantee that an initialization is done at compile time. It also allows constant expressions involving user-defined types.

In Listing 3, the Fibonacci function is guaranteed to be executed at compile time if the value passed x is available at compile time.

Using `constexpr` and the features described previously also allow for integral constants. Integral constants are part of the C++ standard and wrap a static constant of a specific type in a class.

This allows us to easily support different SIMD extensions (Intel SSE, AVX2, AVX512, ARM AArch64 and, IBM VMX) while using a generic function for each call (see Listing 4).

If we then want to do a multiplication using SIMD instructions, we can simply use the standard operator with our overloaded functions (see Listing 5). These programming techniques allow us to have a single source file of around 400 lines for small size matrix products on CPUs that support Intel, ARM and IBM processors. They are also very simple to extend.

We have also designed two models for batched computing (Listing 6, Listing 7). The first one is based on allocating a single memory block for all the matrices to improve data locality and additional information overhead (e.g., matrix size), while the other is a group of same-size matrices.

Once we have defined these functions, we can call the kernel to compute a batched `dgemm`.

We also provide a simple C interface with pointers for our matrix product function on CPUs.

6.2. Optimizations for CPUs

The CPU implementation of a matrix-matrix product kernel for very small matrices requires specific design and optimizations, as we have seen previously. Here we will describe our C++ templated code based on our tuning approach described in Section 5. Because we can store three double-precision matrices of size up to 32×32 in the L1 cache on any modern CPU hardware (such as Intel Xeon, AMD, IBM POWER8, ARM Cortex A57, etc.), one can expect that any implementation will not suffer from data cache misses. This can be seen in Fig. 8b and c, where the performance of an *ijk* implementation—which is not cache-aware and cannot be vectorized—is pretty close to the *ikj* one. The *ijk* and *ikj* implementations correspond to

```
// This is true if the CPU is an Intel Processor
#if __x86_64__ && __AVX__

// Defines the load operation for 256-bit simd
inline auto load(double const* ptr , std::integral_constant<unsigned long,256> )
{
  return _mm256_loadu_pd(ptr);
}

#if __AVX512F__
// Defines the load operation for 512-bit simd on KNL
inline auto load(double const* ptr , std::integral_constant<unsigned long,512> )
{
  return _mm512_loadu_pd(ptr);
}
#endif
#endif

// This is true if the CPU is an ARM64 Processor
#if __aarch64__

inline auto load(double const* ptr , std::integral_constant<unsigned long,128>)
{
  return vld1q_f64(ptr);
}
#endif

// This is true if the CPU is an IMB64 Processor
#if defined(__powerpc64__) || defined(__ppc64__)
inline auto load(const double * ptr , std::integral_constant<unsigned long,128>)
{
  return vec_vsx_ld(0,ptr);
}

#endif
```

Listing 4. C++ SIMD load.

```
using simd_size = std::integral_constant<unsigned long,512>
// Propagate the value at A[iA*N] in the 512 SIMD register tmp
auto tmp = set( A[iA*N] , simd_size{} );
// Load B[i]..B[i+simd_size] and multiply
auto C = tmp *  load(&B[i] , simd_size{});
```

Listing 5. C++ multiply operation.

```
// Create a batch that will contain 15 matrices of size 5,5
constexpr auto batch<float, cpu_> b = make_batch(of_size(5_c,5_c) , 15);
// Accessing a matrix from the batch returns a view on it
constexpr auto view_b = b(0);
// Create a grouping of matrices
constexpr auto group<float,cpu_> g(of_size(5_c,5_c));
// Add a matrix to the group
constexpr auto matrix<float,cpu_> d_ts( of_size(5_c,5_c) );
g.push_back(d_ts);
```

Listing 6. Batched matrices.

```
constexpr auto batch<float, cpu_> b =  make_batch(of_size(5_c,5_c) , 15);
constexpr auto batch<float, cpu_> b1 = make_batch(of_size(5_c,5_c) , 15);
// Product of two batched matrices using C++ operator
constexpr auto c = b * b1;
// Product using the batch dgemm function that can be specialized depending on parameters
constexpr auto c = batch_gemm(b , b1 );
```

Listing 7. Batched operations.

```
for(int i=0 ; i < N ; ++i){
  for(int j = 0 ; j < N ; ++j){
    double c = 0.;
    for(int k = 0 ; k < N ; ++k){
      c += A[i*N + k ] * B[k*N + j ] ;
    }
    C[i*N + j] = beta*C[i*N + j] + alpha*c;
  }
}
```

Listing 8. ijk loop.

```
for(int i=0 ; i < N ; ++i){
  for(int k = 0 ; k < N ; ++k){
    for(int j = 0 ; j < N ; ++j){
      tmp[j] = A[i*N + k ] * B[k*N + j];
    }
  }
  for(int j = 0 ; j < N ; ++j){
    C[i*N + j] = beta*C[i*N + j] + alpha*tmp[j];
}
```

Listing 9. ikj loop.

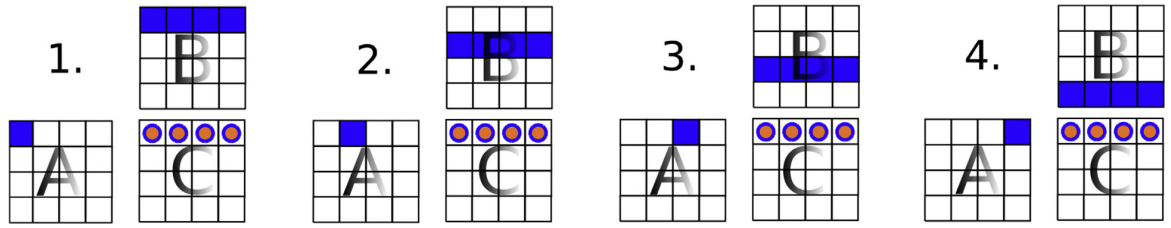


Fig. 5. Computing the first row of C in a 4-by-4 matrix product using SIMD.

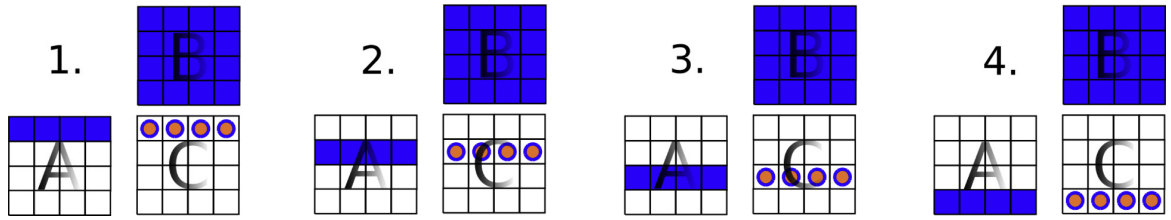


Fig. 6. Example of a 4-by-4 matrix product using SIMD.

the simple matrix product implementation using `for` loops (Listing 8, Listing 9). The *ikj* version is cache-friendly as data is accessed in a continuous fashion, which also gives the possibility to the compiler for vectorization. In the *ijk* version, the data is not accessed contiguously, but we can minimize the number of store operations by computing one value of *C* for each iteration of the innermost loop. For smaller sizes, the *ijk* implementation is more efficient than the *ikj* one, as it optimizes the number of stores (Fig. 7a).

To obtain a near-optimal performance, we conduct an extensive study on the performance counters using the Performance Application Programming Interface (PAPI) [16] tools. Our analysis concludes that in order to achieve an efficient execution for such computations, we need to maximize the CPU occupancy and minimize the data traffic while respecting the underlying hierarchical memory design. Unfortunately, today's compilers cannot introduce highly sophisticated cache- or register-based loop transformations without indications and, consequently, this kind of optimization should be studied and implemented by the software developer [17]. This includes techniques like reordering the data so that it can be easily vectorized, reducing the number of instructions so that the processor spends less time in decoding them, prefetching the data that will be reused in registers, and using an optimal blocking strategy.

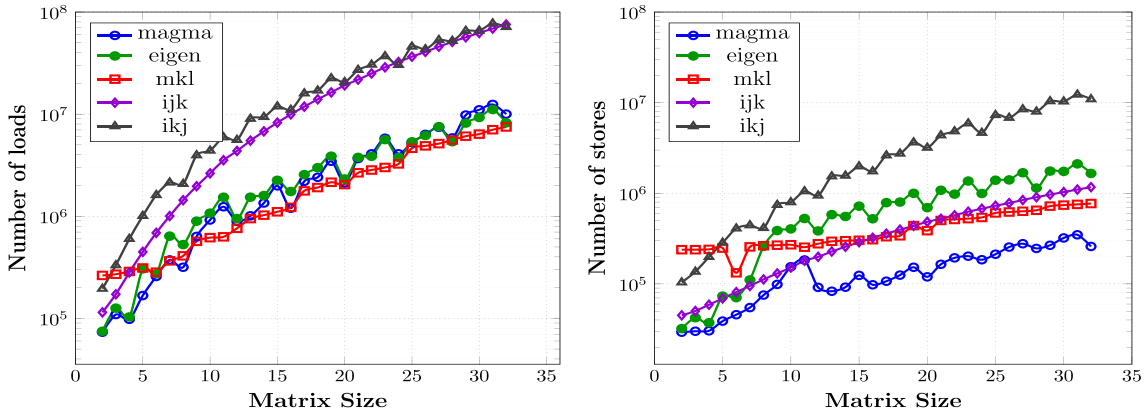
6.3. Data access optimizations and loop transformation techniques

In our design, we propose to order the iterations of the nested loops such that we increase locality and expose more parallelism for vectorization. The matrix-matrix product is an example of perfectly nested loops, which means that all the assignment statements are in the innermost loop. Thus, loop unrolling, loop peeling, and loop interchange can be useful techniques for such algorithms [18,19]. These transformations improve the locality and help to reduce the stride of an array-based computation. In our approach, we propose to unroll the two innermost loops so that the accesses to matrix B are independent from the loop order, which also allows us to reorder the computations for continuous access and improved vectorization. This technique enables us to prefetch and hold some of the data of B into the SIMD registers.

Here, we manage to take advantage of the knowledge of the algorithm (see Figs. 5 and 6), and, based on the principle of locality in reference [20], optimize both the temporal and spatial data locality. In Fig. 5, we can see that to compute one line of the matrix C we actually need to load the full matrix B into the L1 cache. Also, for each subsequent line of C that we compute, we will also need the matrix B (see Fig. 6). Therefore, the more values of B that we can pre-load and keep in the L1 cache, the fewer memory accesses will have to be done.

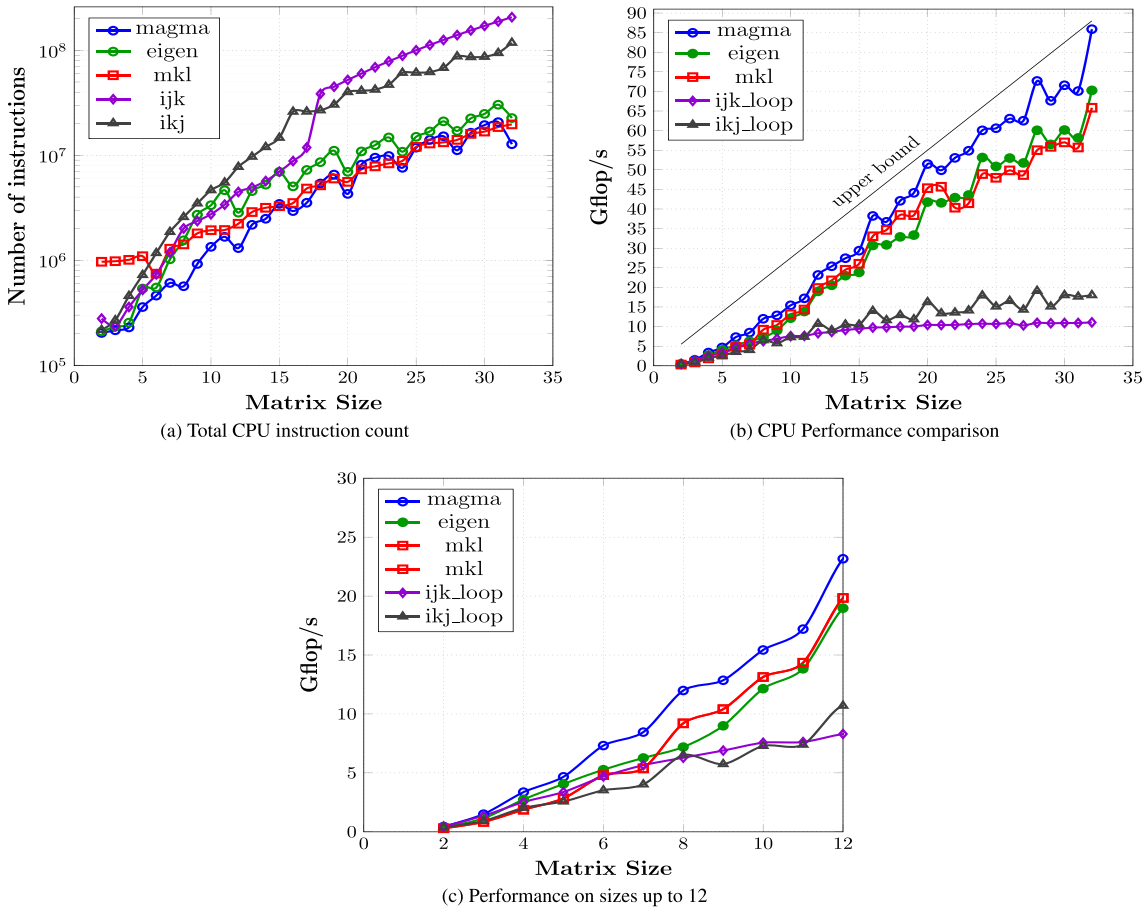
6.4. Register data reuse and locality

Similarly to the blocking strategies for better cache reuse in numerically intensive operations (e.g., large matrix-matrix products), we focus on register blocking to increase performance. Our study concludes that register reuse and minimizing store operations on the C matrix ends up being the key factors for performance. The idea is that when data is loaded into an SIMD register, it will be reused as much as possible before its replacement by new data. The amount of data that can be kept in registers becomes an important tuning parameter. For example, on a system with an AVX-2 register, an 8×8 matrix requires 16 256-bit AVX-2 registers to be completely loaded. If the targeted hardware consists of only 16 256-bit AVX-2 registers, one can expect that loading the entire B will not be optimal, as we will have to reload the vectors for A



(a) # of load instructions (b) # of store instructions

Fig. 7. CPU Performance counters measurement of the memory accesses on Intel Xeon E5-2650 v3 processor (e.g., Haswell).



(a) Total CPU instruction count (b) CPU Performance comparison

(c) Performance on sizes up to 12

Fig. 8. Experimental results of the matrix-matrix multiplication on Haswell.

Algorithm 1 Generic matrix-matrix product applied to a matrix of size 16×16 with 16 256-bit registers.

```
1: Load rB_00, rB_01, rB_02, rB_03                          ▷ load first row of B
2: Load α, β
3: S = 16
4: for i = 0, 1, ... , S-1 do
5:     rA ← Load A[i*S]                                     ▷ load one value of A
6:     rC_i0 = rA * rB_00; ... rC_i3 = rA * rB_03
7:     for u = 1, 2, ... , S-1 do
8:         rA ← Load A[i*S + u]
9:         Load rB_u0, rB_u1, rB_u2, rB_u3                  ▷ load row "u" of B
10:        rC_i0 += rA * rB_u0; ... rC_i3 += rA * rB_u3
11:    end for
12:    rC_i0 = α rC_i0 + β (Load C_i0); ... rC_i3 = α rC_i3 + β (Load C_i3)
13:    Store rC_i0, rC_i1, rC_i2, rC_i3                     ▷ store row "i" of C
14: end for
```

and C. However, if we load only 8 registers for B, which is equal to 4 rows, we can compute a row of C at each iteration and reuse these 8 registers for each iteration. This reduces the number of loads, stores, and total instructions from $O(n^2)$ to $O(n)$, compared to a classical *ijk* or *ikj* implementation as depicted in Figs. 7a, b, and 8a, respectively. Similarly, if we had a CPU with at least 32 registers of 256-bit, we could fit the full matrix B in the registers and never reload it during the computation. The Intel KNL and IBM POWER8 architectures correspond to this case.

6.5. Algorithmic advancements

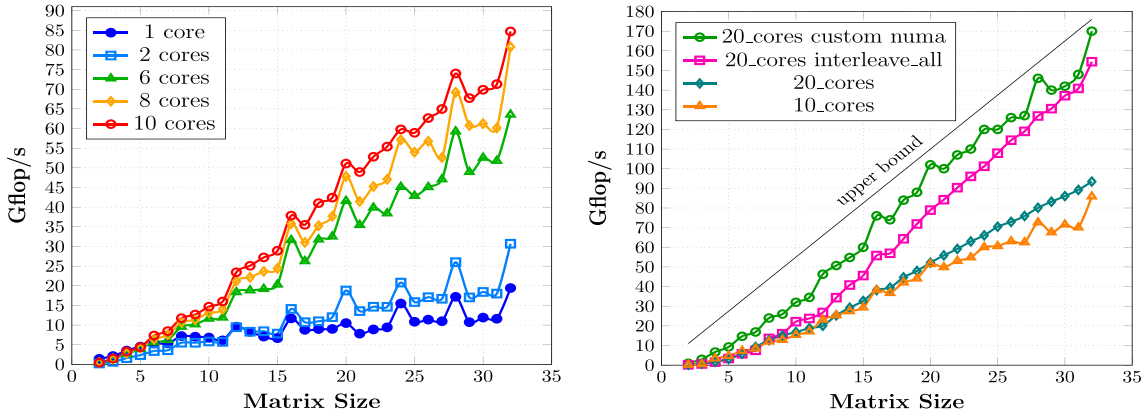
Algorithm 1 is an example of our methodology for a matrix-matrix product of 16×16 matrices with an AVX2 instruction set and 16 registers. In this pseudocode, we start by loading four 256-bit AVX-2 registers with values of B which correspond to the first row. These registers are reused throughout the algorithm. In the main loop (Lines 4–14), we start by computing the first values of every multiplication (stored into a register named M=A × B) based on the prefetched register in line 1. Then, we iterate on the remaining rows (Lines 7–11) loading B, multiplying each B by a value of A, and adding the result into M. Once the iteration over a row is accomplished, the value of M is the final result of A × B; and thus, we can load the initial values of C, multiply by α and β, and store it back before moving toward the next iteration in such a way that minimizes the load/store, as shown in Fig. 7. Each C ends up being loaded/stored once. We apply this strategy to matrix sizes ranging from 8 to 32 because the matrices can fit in the $L1$ cache for these small sizes. Different blocking strategies (square versus rectangular) have been studied through our autotuning process in order to achieve the best performance. We generate each matrix-matrix product function at compile time with C++ templates. The matrix size is passed as a function parameter using C++ integral constants.

In the following subsections, we will compare on different architectures the performance of our MAGMA code to the vendor-tuned matrix-matrix product and a simple implementation without any optimization.

6.6. Application to an Intel Haswell Processor

As described above, operating on matrices of very small sizes is a memory-bound problem—thus, increasing the number of CPU cores may not always increase the performance since it will be limited by the bandwidth, which can be saturated by a few cores. We performed a set of experiments to clarify this behaviour and illustrate our findings in Fig. 9a. As shown, the notion of perfect speed-up does not exist for a memory-bound algorithm, and adding more cores increases the performance slightly. We performed a bandwidth evaluation when varying the number of cores to find that a single core can achieve about 18GB/s while 6 and 8 cores (over the available 10 cores) can reach about 88% and 93% of the practical peak bandwidth, which is about 44GB/s.

In Fig. 8, we compare the performance on a single node of our MAGMA code with an MKL, Eigen, *ikj* and *ijk* code. Our implementation, based on a different variation of blocking and unrolling, reaches better performance than Eigen or MKL. We have seen in Fig. 7a that our solution is very similar in terms of the number of load operations. However, our blocking strategies for these small sizes allow us to more efficiently reduce the number of register load operations on B and C store operations (Fig. 7b). This reduces the total number of memory operations and instructions which also alleviates the instruction cache. The small performance gap between Eigen and MKL can be attributed to a difference in the size used for the blocking strategy. Different versions of MKL may also have small variations in performance due to a change in blocking strategy. As both Eigen and MKL do not minimize the number of stores operations, the performance reached is lower than our MAGMA code. The performance of libxsmm [21], another library from Intel with highly tuned BLAS, was better than older MKL versions but similar to the newest MKL 2018 which we used, and therefore we do not explicitly include it in our comparison.



(a) Effect of the number of CPU cores

(b) Effect of the NUMA memory management

Fig. 9. CPU Performance analysis on Haswell.

We also studied non-uniform memory access (NUMA) [22] using a node with two Xeon CPUs as seen in Fig. 9b. A standard memory allocation puts all of the data in the memory slot associated with the first socket until it gets filled, then starts filling the second socket. Since the problem size we are targeting is very small, most of the data is allocated on one socket, and thus using the second socket's extra 10 cores will not increase the performance. This is due to the fact that the data required by the cores of the second socket goes through the memory bus of the first socket, and is thus limited by the bandwidth of one socket (44GB/s). There are ways to overcome this issue. By using NUMA with the `interleave=all` option, which spreads the allocation over the two sockets via memory pages, we can improve the overall performance. However, for very small sizes, we observe that such a solution remains far from the optimal bound since data is spread out over the memory of the two sockets without any rules dictating that cores from socket 0 should only access data on socket 0, and vice versa. To further improve performance, we use a specific NUMA memory allocation, which allows us to allocate half of the matrices on each socket. As shown in Fig. 9b, this allows our implementation to scale over the two sockets and to reach close to the peak bound.

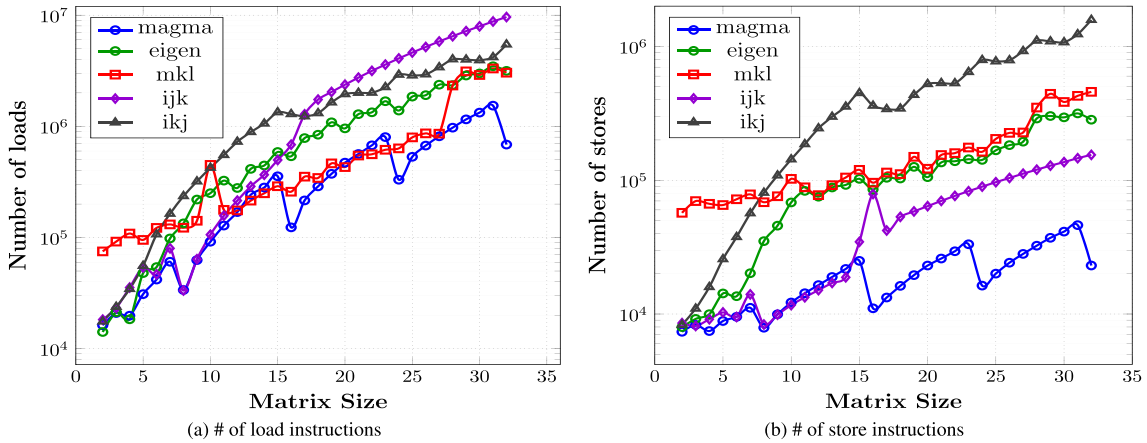
6.7. Application to the Intel KNL

The Intel KNL is a new architecture that provides improved hardware features such as 512-bit vector units, 32 vector registers instead of 16, up to 288 hardware threads, and a high-bandwidth memory called MCDRAM. The KNL can be configured in different ways using the MCDRAM and sub-NUMA nodes which have been detailed in Sodani's Hot Chips presentations [23]. An extensive study to apply the Roofline Performance Model [24] on the KNL [25] has shown the differences between the MCDRAM configurations and the model's impact on performance. Our study comes to the same conclusion, and all application results we present use the quad-flat representation as all of the data fits in the MCDRAM. We use the Linux utility *numactl* to target the MCDRAM (flag $-m$ 1). To compile with GCC on the KNL, we add the $-march = knl$ flag for AVX512F instructions support.

We can see in Fig. 10 that the number of load 10a and store 10b instructions follow the same pattern as with the Haswell processor. The important drops we see on each graph for the KNL are a bit different than on the Haswell processor. This is due to the size of the vector unit increasing from 256-bit to 512-bit. For double-precision operations, we see on every multiple of 8 a large drop in the number of load/store instructions due to the matrix size being a multiple of the SIMD size.

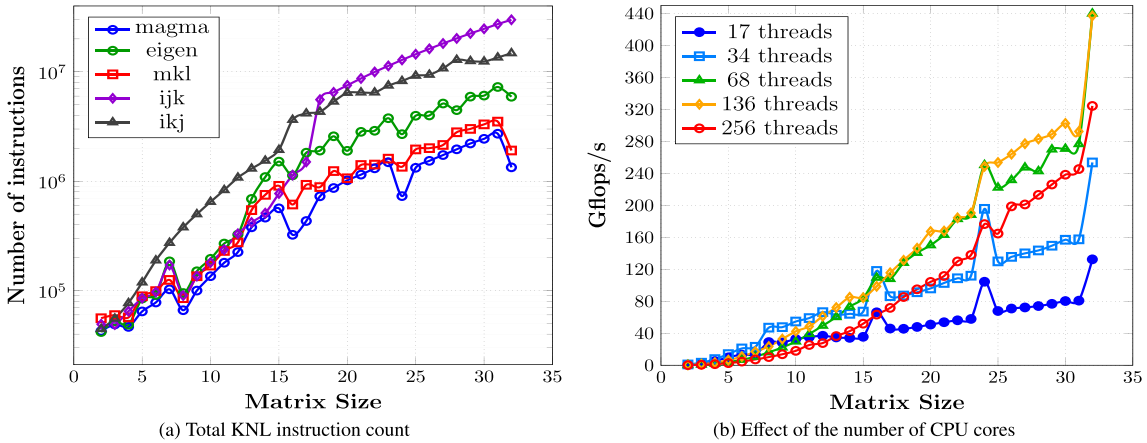
We generally reach the same number of load instructions as MKL since we cannot really optimize this parameter significantly, as seen with the Haswell CPU. The Eigen library has a greater number of load instructions than our MAGMA code or MKL, but has the same number of store instructions as MKL, implying that their blocking strategy is very similar but that the unrolling and register reuse may differ. By not using a standard blocking strategy, we are able to further optimize the number of store operations compared to the Haswell CPU due to the larger SIMD vector size. The prefetch and unroll strategy have also been tuned differently due to the increase in number of the vector registers and their size. We can see in Fig. 11a that we always have a lower total instruction count. On the KNL, it is possible to have up to 4 threads per core. Using the maximum number of threads is never efficient, as seen in Fig. 11. Using 2 threads per core can sometimes yield better performance, but the delta is quite negligible. Except for matrix sizes smaller than 12, it is always better to utilize every CPU core available on the KNL.

Similarly to what we saw with the Haswell processor, our analysis and design directly translate to performance obtained (see Fig. 12). The performance with our generated code in MAGMA is always better than that of the MKL or Eigen code. We can see that the use of the MCDRAM as the main memory instead of DDR4 heavily impacts the performance. We observe an overall performance increase of two when using MCDRAM. We end up far from the upper bound due to data in



(a) # of load instructions (b) # of store instructions

Fig. 10. CPU performance counters measurement of the memory accesses on KNL.



(a) Total KNL instruction count (b) Effect of the number of CPU cores

Fig. 11. CPU performance counters and scaling analysis on KNL.

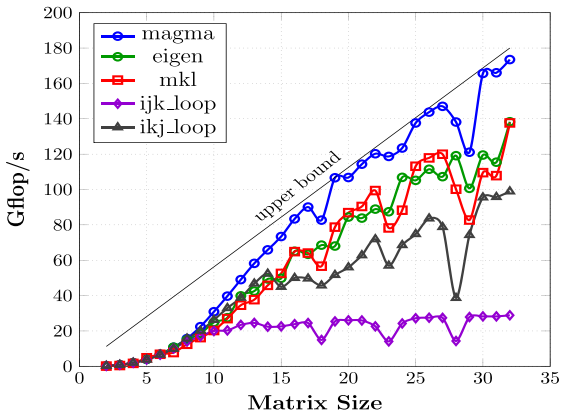
the MCDRAM not being read multiple times, which limits the bandwidth usage. Using MCDRAM also leads to more stable performance. Memory-bound problems tend to be less stable in terms of performance when using SIMD instructions and multithreading due to computing power/bandwidth ratio. This is even more prevalent with the KNL processor, as its very large SIMD instructions (512-bit) correspond to the size of an L1 cache line (64 Bytes).

6.8. Application to ARM processor

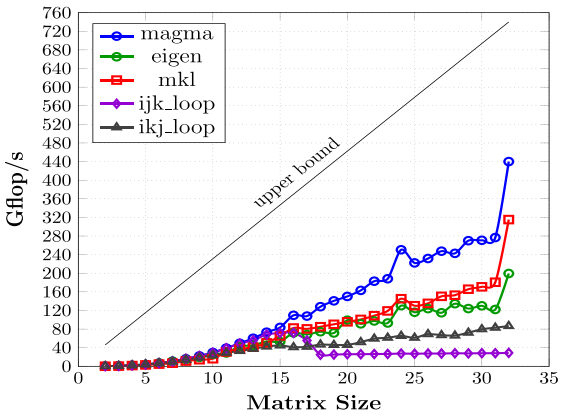
The ARM processor that we use for this benchmark is the CPU of the Tegra X1, a 4-core Cortex A57. The problems we detailed earlier still apply to the Tegra, but on a different scale. Indeed, the ARM intrinsics only support 128-bit vectors, which severely limit the SIMD use for double-precision computations.

In Fig. 13, we compare the performance of our MAGMA code, an *ijk* code, an *ikj* code, an Eigen code and an OpenBLAS [26] code using the latest version available from the develop branch on Github.

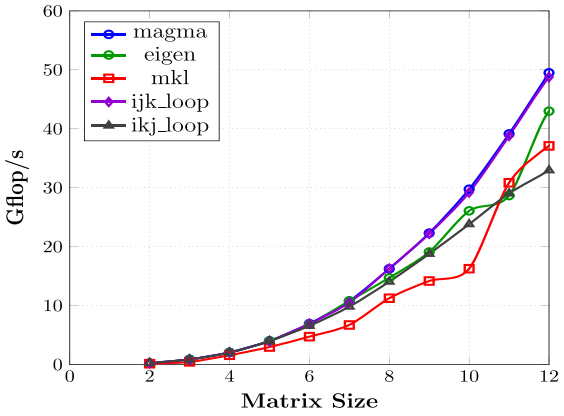
Results follow the same trend we saw on the Intel processors. On very small sizes, *ijk* and *ikj* versions are quite efficient as the arithmetic intensity is very low, limiting the usefulness of parallelism. With increased sizes, we start to see these versions stall and reach a limit set around 3.5 gigaFLOP/s. The OpenBLAS version provides good performance but is limited by its blocking model, which is not adapted for very small sizes. We obtain better results than OpenBLAS with the Eigen library but still lower than our MAGMA code. The difference lies in the blocking and unrolling strategies from our tuning approach compared to those of Eigen or OpenBLAS.



(a) CPU performance comparison without MCDRAM

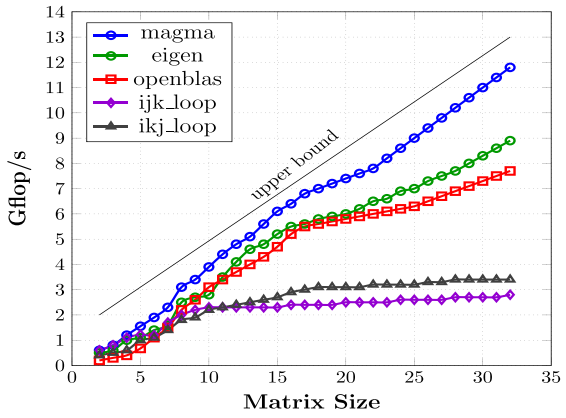


(b) CPU Performance comparison with MCDRAM

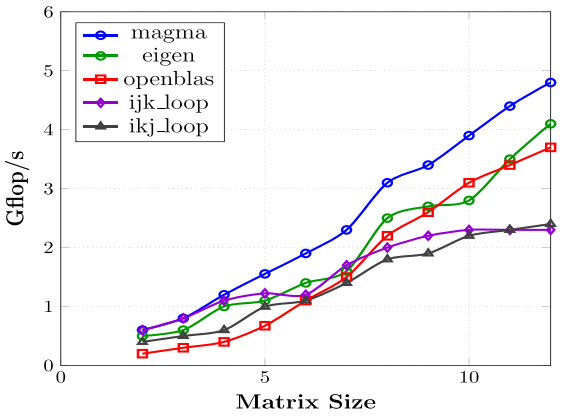


(c) Performance on sizes up to 12

Fig. 12. Experimental results of the matrix-matrix multiplication on KNL — 68 threads.

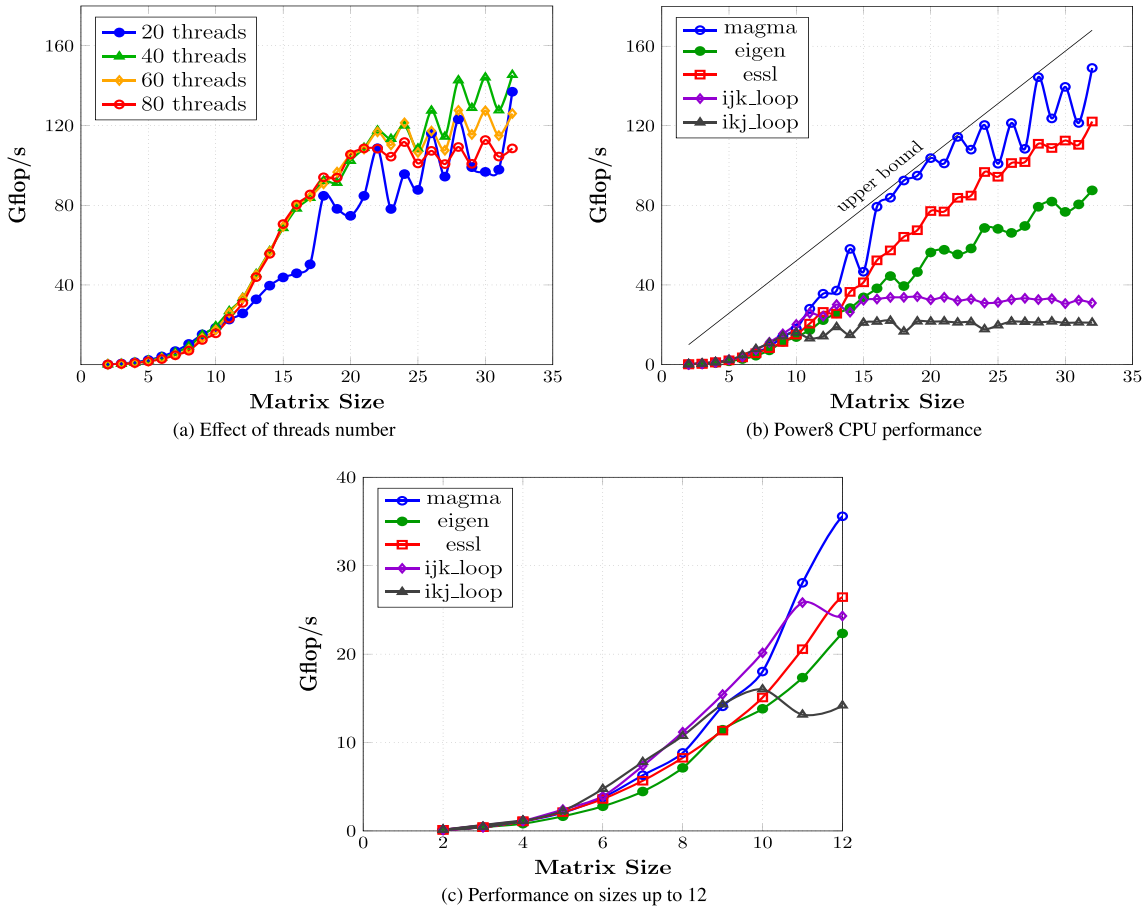


(a) Matrix sizes up to 32



(b) Matrix sizes up to 12

Fig. 13. Experimental results of the matrix-matrix multiplication on the Tegra X1.



(a) Effect of threads number

(b) Power8 CPU performance

(c) Performance on sizes up to 12

Fig. 14. Experimental results of the matrix-matrix multiplication on a POWER8 processor.

6.9. Application to IBM processor

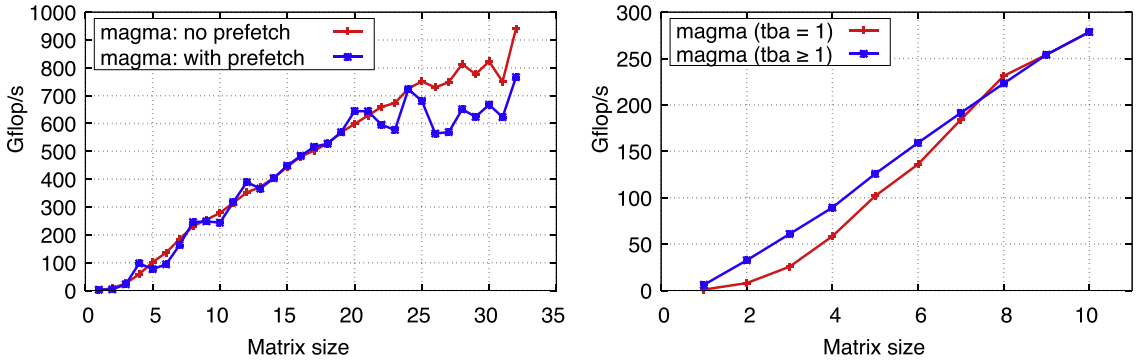
The IBM processor used for this benchmark is a POWER8 with 10 cores and 8 threads per core. The POWER8 has a high number of threads per core due to its higher on-chip memory with the embedded DRAM (eDRAM) and memory bandwidth ($\approx$ 85GB/s). The Altivec intrinsics support only 128-bit vectors, which is one of the limiting factors.

To efficiently use the high memory bandwidth of the POWER8 and maximize the pipeline occupancy, it is important to properly tune the number of threads per core (see Fig. 14a). We have found the best thread number for matrix ranging from a size of 8–32 to be 40 threads. Fig. 14b compares the performance of our MAGMA code to the different algorithms, ESSL being the IBM BLAS library for PowerPC architectures.

In Figs. 14a and b, we can see that the POWER has a hard time using both standard and vectorial instructions. This is why our MAGMA code has some performance drops on sizes not multiple of the SIMD vector size. It is possible to improve performance for these sizes, but this requires specific code optimization and different compiler options. As we want to provide a generic implementation of the code with a simple compilation process, we only give the results obtained with the overall best configuration. The Eigen version provides weaker results than expected compared to the ESSL library from IBM. The POWER8 provides very distinct architectural features from Intel or ARM architectures, so the blocking sizes and unrolling features in Eigen may not be best suited to this architecture.

7. Programming model, performance analysis, and optimization for GPUs

Concerning the development for GPUs, we set a goal to have a unified code base that can achieve high performance for very small matrices. The design strategy is different from the MAGMA batched GEMM kernel for medium and large sizes [8]. The latter uses a hierarchical blocking technique where different thread blocks (TBs) compute different blocks of the output matrix C. With such a design, a TB reads an entire block row of A and an entire block column of B to compute its share of



(a) Data prefetching (b) Aggregation

Fig. 15. Impact of data prefetching and aggregation on performance. The experiment is performing 100,000 `GEMM` operations in double precision on a Tesla P100 GPU.

C. Obviously, there will be redundant reads of both *A* and *B* among TBs. For extremely small sizes (e.g., up to 32), we cannot afford redundant reads, since the memory bandwidth becomes the main bottleneck for such computational workloads.

Instead, we adopt a strategy where a single TB performs the entire multiplication of at least one problem, computing all of *C* with no subdivision involved. We start by an initial design that represents a special case of a 1×1 blocking technique in order to avoid redundant reads from global memory. Since the sizes considered are very small, there are enough resources on the streaming multiprocessor (SM) to store all of *A, B*, and *C* in shared memory and/or registers. Similar to the design proposed in [8], we use CUDA C++ templates to have an abstract design that is oblivious to tuning parameters and precision. In this paper, we discuss the main design aspects of the proposed kernel, and how we managed, through an extensive autotuning and performance counter analysis, to improve its performance on the Tesla P100 GPU over the original design proposed in [9].

7.1. A Shared-memory approach

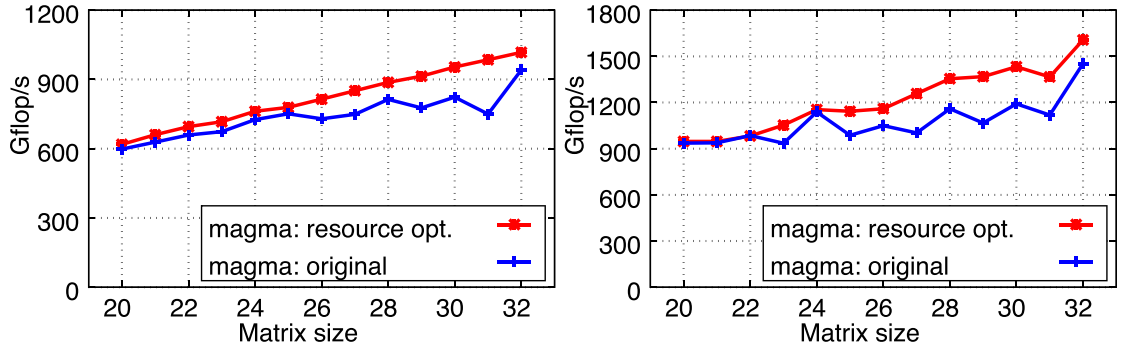
Our previous work [9] showed that using shared memory to exploit data reuse is superior to using the read-only data cache. We start with a simple design where *A* and *B* are stored in shared memory for data reuse, and *C* is stored in registers. Each TB performs exactly one GEMM operation. Eventually, the kernel launches as many TBs as the number of multiplications to be performed. Using a 2-D thread configuration, each thread computes one element in the output matrix *C*. The matrices *A, B*, and *C* are read only once from the global memory. Data reuse of *A* and *B* occurs only in shared memory, where each thread reads a row of *A* and a column of *B* to compute its respective output.

7.2. Data prefetching

Our first try to improve the performance adds data prefetching to the initial design. By assigning more multiplications per TB, we can prefetch the next triple *A, B*, and *C* while another multiplication is taking place. We choose to prefetch data in registers in order to reduce synchronization and avoid overloading the shared memory. Recall that the register file per SM is about 256KB, while the shared memory is 64KB at maximum. Surprisingly, Fig. 15a shows that data prefetching does not result in performance gains except for slight improvements for a few certain sizes. We list two major reasons for this behavior. The first is that the prefetching technique uses $4\times$ the register resources of the original design, which might limit the number of TBs per SM as the sizes get larger. The second is that there is a costly branch statement inside the kernel that checks whether there is more data to prefetch. Eventually, we decided to drop data prefetching from the final design.

7.3. Thread block-level aggregation

We adopt a different approach to assign multiple multiplications per TB. Considering the original design, we aggregate a number of TBs together into one bigger TB. Internally, the new TB is divided into smaller working groups, each taking care of one multiplication. Such a design significantly improves the performance for tiny sizes. The main reason is that the original design suffers from a bad TB configuration, which assigns very few warps, or even less than a warp, to a TB. The aggregation technique improves this configuration for tiny sizes. As an example, the original design launches 4 threads per TB for a multiplication of 2×2 matrices, which is one eighth of a warp. The aggregation technique groups 16 multiplications per TB, thus launching 2 warps per TB. The level of aggregation is controlled through a tuning parameter (`tba`). Fig. 15b shows the impact of aggregation (after tuning `tba` for every size) on performance, where we observe performance improvements on sizes smaller than 8. For example, aggregation achieves a speedups of 4.1×, 2.5×, 1.7×, 1.25×, and 1.20× for sizes 2, 3, 4, 5, and 6, respectively. For larger sizes, we observe that it is always better to set `tba`=1, since there are enough warps per TB.



(a) Performance on NVIDIA P100

(b) Performance on NVIDIA V100

Fig. 16. Impact of resource optimization on performance. The experiment is performing 100,000 `GEMM` operations in double precision.

7.4. Resource optimization

We propose a new optimization technique that helps improve the performance for the sizes in the interval [17:32]. Recall that the original design of the GPU kernel [9] has a negative impact on performance as the sizes become larger than 16. Typically, the performance stays within 90% of its roofline until size 16, and then drastically shifts away from the roofline, scoring as low as 60% of the upper bound. This is mainly a design issue with respect to the amount of resources required by the kernel. For a multiplication of size N, the original design uses $N \times N$ threads and $2N \times N$ of shared memory per TB. This configuration limits the number of TBs that can execute concurrently per SM. For example, if $N = 32$, each thread block requires 1024 threads and about 16KB of shared memory (for double precision). The shared memory requirement is one third of the multiprocessor capacity, but the number of threads limits the occupancy to just two TBs per multiprocessor. This means that at least one third of the shared memory in each multiprocessor is wasted. In order to mitigate this effect, we recursively block the computation in shared memory, which enables us to use fewer threads and less shared memory. The new design uses $\hat{N} \times \hat{N}$ threads and $2\hat{N} \times \hat{N}$, where $\hat{N}$ is a tuning parameter that is typically less than N, such that:

$$\left\lceil \frac{N}{2} \right\rceil \leq \hat{N} < N \ . \tag{1}$$

This optimization of threads/shared memory comes at the cost of extra resources from the register file, which is underutilized in the original design. The kernel reads A, B, and C once into registers. Since the shared memory resources can only accommodate two $\hat{N} \times \hat{N}$ blocks, the computation is performed in several stages. Eq. (1) enables a 2×2 blocking of the form:

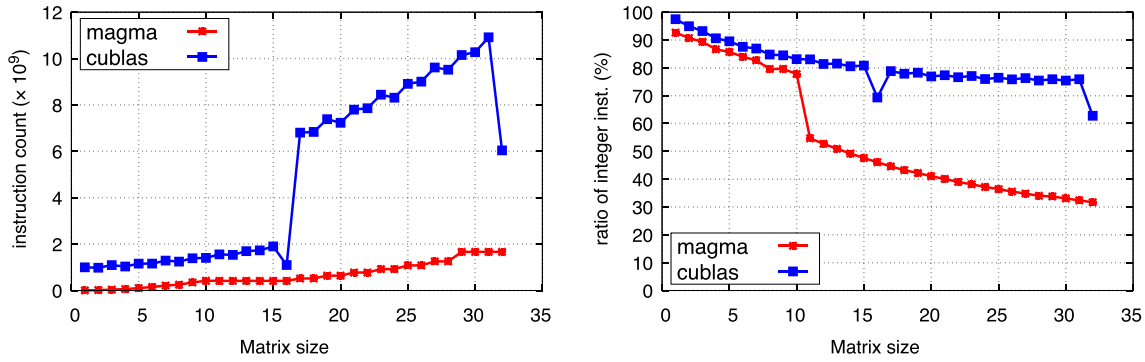
$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \alpha \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \times \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix} + \beta \begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} \tag{2}$$

The sizes of C_{00}, C_{01}, C_{10}, and C_{11} are $\hat{N} \times \hat{N}$, $\hat{N} \times (N - \hat{N})$, $(N - \hat{N}) \times \hat{N}$, and $(N - \hat{N}) \times (N - \hat{N})$, respectively, which are exactly the same for the A and B sub-blocks. The scaling with β is done upon reading C. In order to compute C_{00}, the kernel performs the following steps:

(1) Load A_{00} and B_{00} into shared memory
(2) Perform $C_{00} = C_{00} + \alpha A_{00} \times B_{00}$
(3) Load A_{01} and B_{10} into shared memory
(4) Perform $C_{00} = C_{00} + \alpha A_{01} \times B_{10}$

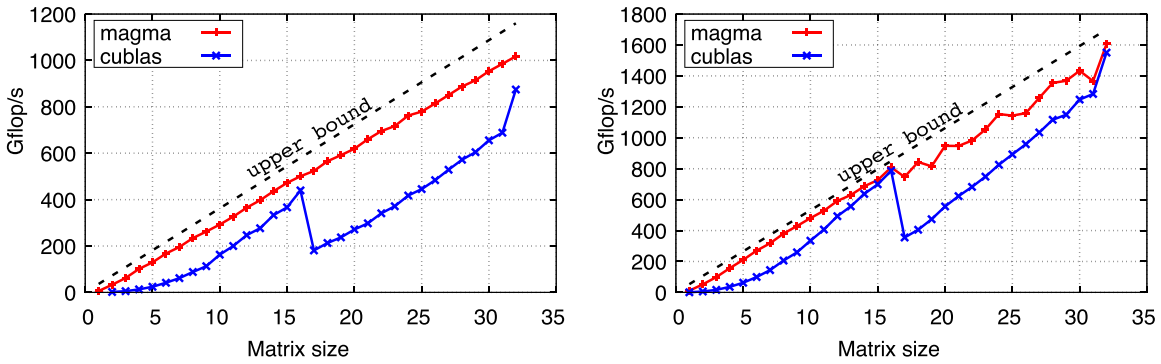
Similar steps are carried out for C_{01}, C_{10}, and C_{11}. Since C is stored in the register file, the accumulation occurs in the registers holding $\beta \times C$. Eventually, the kernel is performing one `GEMM` operation using much fewer resources in terms of shared memory and threads. While this comes at the cost of using more registers, the register file per SM is big enough to accommodate such an increase. The overall result is an improved performance for relatively larger sizes as shown in Fig. 16. In the matrix range from 20×20 to 32×32, we observe speedups ranging from 3% up to 31% for the P100 GPU. As for the V100 GPU, the speedup is up to 28%. Generally, the original kernel loses performance as we increase the sizes—unlike the new kernel, which has a more stable performance.

Eventually, our solution combines all of the aforementioned techniques, with the exception of data prefetching. We can subdivide the size range 1–32 into three segments. The first represents the tiny sizes in the range of 1–10, where we use the original kernel with `tba` > 1. The second is the midrange 11–19, where we still use the original kernel, but setting `tba` = 1. The third is the relatively larger sizes in the range of 20–32, where we call the new kernel with resource optimization.



(a) Integer instruction count (b) Percentage of integer instructions

Fig. 17. Profiling the instruction mix of MAGMA versus CUBLAS. The experiment is performing 100,000 `GEMM` operations in double precision on a Tesla P100 GPU.



(a) Performance on Nvidia P100 (b) Performance on Nvidia V100

Fig. 18. Final obtained performance of 100,000 `GEMM` operations in double precision on a Tesla P100 and V100 GPU.

7.5. Instruction mix

A common optimization in all of our designs is the instruction mix of the GPU kernel, which is crucial to performance when operating on matrices of very small sizes. Integer instructions, which are used for loop counters and memory address calculations, can be quite an overhead in such computations. Moreover, our study showed that a loop with a predefined boundary can be easily unrolled and optimized by the NVIDIA compiler. Using CUDA C++ templates that are instantiated with a compile-time tuning parameter, we are able to produce fully unrolled code for every size of interest. By profiling the kernel execution, we collected the number of integer instructions as well the number of the FP64 instructions. Fig. 17 shows the total number of integer instructions as well as the ratio of integer instructions to the total number integer and FP64 instructions. We observe that the MAGMA kernel always executes fewer integer instructions than cuBLAS. It also has the smallest ratio across all sizes. An interesting observation of the cuBLAS implementation, for this range of matrices, is that it uses a fixed blocking size of 16×16. This explains the drops at sizes 16 and 32, where the problem size matches the internal blocking size.

7.6. Support for different transposition modes and rectangular sizes

The `GEMM` routine, by definition, allows either of A or B to be transposed in the multiplication. In order to support such configurations with minimal changes to the computational part of the kernel, both A and B are transposed on the fly (if needed) while reading them from the global memory. This leaves the computational part unchanged, regardless of the input transposition modes of A and B.

The same kernel design applies for rectangular sizes. The difference from square cases comes on the side of thread configuration and shared memory requirements. For a rectangular matrix of size $M \times N$, the kernel should be configured with $\hat{M} \times \hat{N}$ threads, where $\hat{M}$ is computed in a similar manner to Eq. (1). The reading of A and B into shared memory is properly handled to prevent out-of-bounds memory reads or writes, especially when the common dimension K is not fully divisible by $\hat{M}$ or $\hat{N}$.

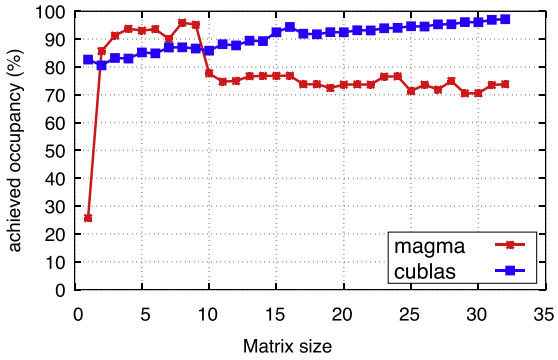
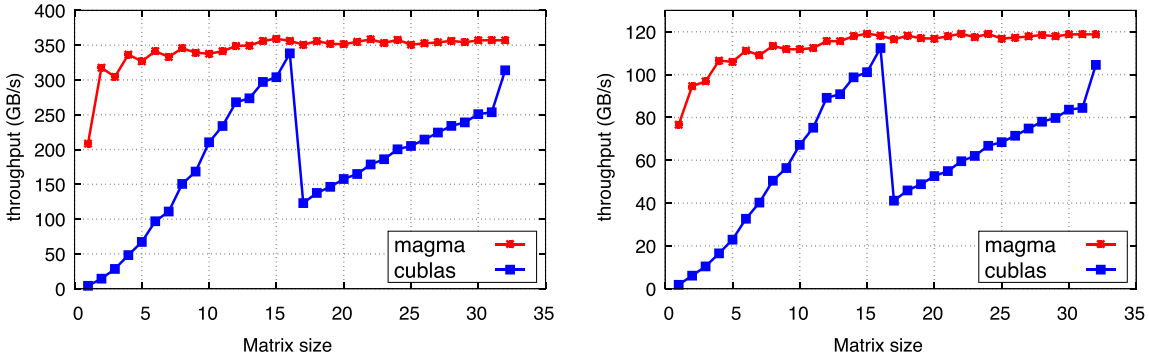


Fig. 19. Profiling the achieved occupancy of MAGMA versus cuBLAS. The experiment is performing 100,000 GEMM operations in double precision on a Tesla P100 GPU.



(a) Read throughput (b) Write throughput

Fig. 20. DRAM read and write throughputs. The experiment is performing 100,000 GEMM operations in double precision on a Tesla P100 GPU.

7.7. Performance and profiling results

Fig. 18a shows the final performance of the proposed solution against cuBLAS using the NVIDIA P100 GPU. We also show the upper bound of the performance, as estimated in Section 3. The results show that MAGMA is significantly faster than cuBLAS, scoring speedups that range from 1.13× (at size 32) up to 18.2× (at size 2). We observe that the smaller the size, the larger the speedup. In addition, the MAGMA kernel is up to 88% of the performance upper bound.

Now considering the V100 GPU (Fig. 18b), the trends and observations are similar, but the performance is uniformly lifted up, reaching about 1600 gigaFLOP/s for matrices of size 32. This 50% performance boost is in proportion to the 50% more powerful V100 GPU. Fig. 18b shows more details, as it compares the MAGMA performance versus cuBLAS. While both graphs achieve similar performances on sizes 15, 16, 31, and 32, MAGMA outperforms cuBLAS on all other sizes, scoring speedups between 1.08× and 9.3× (at size 2). MAGMA is also up to 95% close to the performance upper bound.

An interesting observation is depicted in Fig. 19, which shows that the cuBLAS kernel achieves higher occupancy than the MAGMA kernel, starting from size 10. We point out that the achieved occupancy metric does not necessarily give good insight into performance, and it has to be combined with other metrics. In fact, the achieved occupancy is defined as the ratio of the average active warps per active cycle to the maximum number of warps supported on the SM. However, the measurement of busy warps does not mean that they are doing useful work. In fact, Fig. 17 shows that the cuBLAS kernel executes far more integer instructions than the MAGMA kernel. Moreover, since the computation is memory bound, we show a more representative metric. Fig. 20 shows the read and write throughputs of the GPU memory during execution. The proposed MAGMA kernel achieves significantly higher throughput than cuBLAS in both reads and writes, with an up to 22 × higher throughput in reads and up to 15 × higher throughput in writes.

8. Conclusions and future directions

This paper presented an extensive study of the design and optimization techniques for Batched GEMMs on small matrices. The work is motivated by a large number of applications ranging from machine learning to big data analytics, to high-order finite element methods and more, which all require fast linear algebra on many independent problems that are

of size 32 and smaller. The use of standard Batched BLAS APIs in applications is essential for their performance portability. However, this performance portability can be obtained provided that—similar to BLAS—vendors start developing and supporting high-performance implementations in their libraries. This is happening now; but still, as shown, GEMMs for small matrices are not yet sufficiently optimized in existing libraries. Therefore, we first developed theoretical models quantifying the peak performances for the architectures of interest, and then developed algorithms and optimization techniques that get very close (within 90%) to those peaks. The results presented significantly outperform currently available state-of-the-art implementations in the vendor-tuned math libraries, as well as popular open source libraries like OpenBLAS and Eigen.

The algorithms were designed for modern multi-core CPU, ARM, Xeon Phi, and GPU architectures. Our solution is to bind all the developments into a single generator that is combined with autotuning to empirically find the best performing kernels, up to exploring a predefined design search space. While this produces a single tool, we note that the best kernels for different architectures and sizes vary, incorporating different optimization techniques, algorithms, and tuning parameters. We provided detailed analysis and the optimization techniques for the different architectures. The optimization techniques, the algorithms, and the overall framework can be used to develop other batched Level 3 BLAS kernels and to accelerate numerous applications that need linear algebra on many independent problems.

Future work includes further optimizations and analyses for other Batched BLAS kernels and their use in applications. One particular application-specific optimization challenge is how to fuse a sequence of Batched BLAS kernels into a single batched kernel. This is often needed in applications as an optimization technique to reduce communications/data transfers.

Finally, it is known that compilers have their limitations in producing top performance codes for computations like the ones addressed here. This influenced our decision to rely not only on compilers and our domain-specific code generation techniques, but also on the use of lower-level programming languages when needed. Current results used intrinsics for multi-core CPUs and CUDA for GPUs—combined with autotuning in either case—to quickly explore the large algorithmic variations developed in finding the fastest one. This is an area that must be further developed and optimized.

Acknowledgments

This material is based in part upon work supported by the US NSF under Grants no. OAC-1740250, NVIDIA, and under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

References

[1] S. Tomov, J. Dongarra, M. Baboulin, Towards dense linear algebra for hybrid GPU accelerated manycore systems, Parallel Comput. 36 (5–6) (2010) 232–240.
[2] J. Dongarra, I. Duff, M. Gates, A. Haidar, S. Hammarling, N.J. Higham, J. Hogg, P. Valero-Lara, S.D. Relton, S. Tomov, M. Zounon, A Proposed API for Batched Basic Linear Algebra Subprograms, MIMS EPrint 2016.25, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, 2016.
[3] A. Haidar, T. Dong, S. Tomov, P. Luszczek, J. Dongarra, A framework for batched and GPU-resident factorization algorithms applied to block householder transformations, in: High Performance Computing, in: Lecture Notes in Computer Science, 9137, 2015, pp. 31–47.
[4] A. Abdelfattah, M. Baboulin, V. Dobrev, J. Dongarra, C. Earl, J. Falcou, A. Haidar, I. Karlin, T. Kolev, I. Masliah, S. Tomov, High-performance tensor contractions for GPUs, in: International Conference on Computational Science (ICCS'16), Elsevier, Procedia Computer Science, San Diego, CA, U.S.A., 2016.
[5] T. Dong, A. Haidar, P. Luszczek, A. Harris, S. Tomov, J. Dongarra, LU factorization of small matrices: accelerating Batched DGETRF on the GPU, in: Proceedings of 16th IEEE International Conference on High Performance and Communications, 2014.
[6] A. Haidar, T. Dong, P. Luszczek, S. Tomov, J. Dongarra, Batched matrix computations on hardware accelerators based on gpus, Int. J. High Perform. Comput. Appl. (2015).
[7] G. Guennebaud, B. Jacob, et al., Eigen v3, 2010, (http://www.eigen.tuxfamily.org).
[8] A. Abdelfattah, A. Haidar, S. Tomov, J. Dongarra, Performance, design, and autotuning of batched GEMM for GPUs, in: The International Supercomputing Conference (ISC High Performance 2016), Frankfurt, Germany, 2016.
[9] I. Masliah, A. Abdelfattah, A. Haidar, S. Tomov, M. Baboulin, J. Falcou, J. Dongarra, High-performance matrix-matrix multiplications of very small matrices, in: Euro-Par 2016: Parallel Processing – 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24–26, 2016, Proceedings, 2016, pp. 659–671.
[10] S.H. Fuller, L.I. Millett, The Future of Computing Performance: Game Over Or Next Level?, National Academy Press, 2011.
[11] J.D. McCalpin, Memory bandwidth and machine balance in current high performance computers, in: IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, 1995, pp. 19–25.
[12] Intel Math Kernel Library, 2016. Available at http://www.software.intel.com.
[13] I. Masliah, M. Baboulin, J. Falcou, Metaprogramming dense linear algebra solvers applications to multi and many-core architectures, in: 2015 iIEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August,, 3, 2015, pp. 69–76.
[14] J. Järvi, B. Stroustrup, Decltype and auto (revision 3), ANSI/ISO C++ Standard Committee Pre-Sydney Mailing 1607 (2004) 04–0047.
[15] G. Dos Reis, B. Stroustrup, General constant expressions for system programming languages, in: Proceedings of the 2010 ACM Symposium on Applied Computing, ACM, 2010, pp. 2131–2136.
[16] P.J. Mucci, S. Browne, C. Deane, G. Ho, Papi: A portable interface to hardware performance counters, in: In Proceedings of the Department of Defense HPCMP Users Group Conference, 1999, pp. 7–10.
[17] D. Loshin, Efficient Memory Programming, first ed., McGraw-Hill Profess., 1998.
[18] N. Ahmed, N. Mateev, K. Pingali, Tiling imperfectly-nested loop nests, in: Supercomputing, ACM/IEEE 2000 Conference, 2000, doi:10.1109/SC.2000.10018. 31–31
[19] D.F. Bacon, S.L. Graham, O.J. Sharp, Compiler transformations for high-performance computing, ACM Comput. Surv. 26 (4) (1994) 345–420, doi:10.1145/197405.197406.
[20] J.L. Hennessy, D.A. Patterson, Computer Architecture, Fifth Edition: A Quantitative Approach, Morgan Kaufmann Publ. Inc., San Francisco, CA, USA, 2011.
[21] A. Heinecke, G. Henry, M. Hutchinson, H. Pabst, Libxsmm: accelerating small matrix multiplications by runtime code generation, in: SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2016, pp. 981–991, doi:10.1109/SC.2016.83.
[22] G. Hager, G. Wellein, Introduction to High Performance Computing for Scientists and Engineers, CRC Press, 2011.
[23] A. Sodani, Knights landing (knl): 2nd generation intel® xeon phi processor, in: Hot Chips 27 Symposium (HCS), 2015 IEEE, IEEE, 2015, pp. 1–24.

[24] S. Williams, A. Waterman, D. Patterson, Roofline: an insightful visual performance model for multicore architectures, Commun. ACM 52 (4) (2009) 65–76.
[25] D. Doerfler, J. Deslippe, S. Williams, L. Oliker, B. Cook, T. Kurth, M. Lobet, T. Malas, J.-L. Vay, H. Vincenti, Applying the roofline performance model to the intel xeon phi knights landing processor, in: International Conference on High Performance Computing, Springer, 2016, pp. 339–353.
[26] Z. Xianyi, W. Qian, Z. Chothia, Openblas, URL: http://xianyi.github.io/OpenBLAS (2012).