High-performance matrix-matrix multiplications of very small matrices

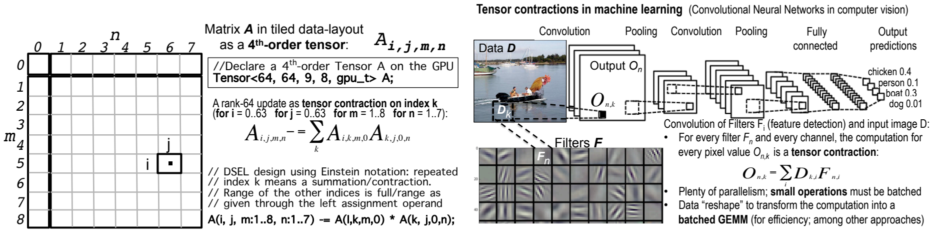
I. Masliah[2], A. Abdelfattah[1], A. Haidar[1], S. Tomov[1], M. Baboulin[2], J. Falcou[2], and J. Dongarra[1,3]

[1] Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA
[2] University of Paris-Sud, France
[3] University of Manchester, Manchester, UK

Abstract. The use of the general dense matrix-matrix multiplication (GEMM) is fundamental for obtaining high performance in many scientific computing applications. GEMMs for *small matrices* (of sizes less than 32) however, are not sufficiently optimized in existing libraries. In this paper we consider the case of many small GEMMs for a wide range of computer architectures, including multicore CPUs, ARM, Intel Xeon Phi, and GPUs. This is a case that often occurs in applications like big data analytics, machine learning, high-order FEM, and others. The GEMMs are grouped together in a single *batched* routine. We present specialized for these cases algorithms and optimization techniques to obtain performance that is within 90% of the optimal. For example, on a P100 GPU for square matrices of size 32, we achieve an execution rate of about $1,030$ Gflop/s in double precision arithmetic, which is 90% of the theoretically derived peak for this computation on a P100 GPU. We show that our results outperform currently available state-of-the-art implementations and vendor-tuned math libraries, including Intel MKL, Nvidia CUBLAS, and OpenBLAS.

1 Introduction

Parallelism in todays 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. Multicore CPUs are combined with discrete high-performance GPUs, or even become integrated parts with them as a system-on-chip (SoC) like in the NVIDIA Tegra mobile family of devices. To extract full performance from systems like these, the 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 $\mathcal{O}(100)$ and smaller. According to a recent survey among the Sca/LAPACK and MAGMA [21] users, 40% of the responders needed this functionality for applications in machine learning, big data analytics, signal processing, batched operations for sparse preconditioners, algebraic multigrid, sparse direct multifrontal solvers, QR types of factorizations on small problems, astrophysics, and high-order FEM. At some point in their execution, applications like these must perform a computation that is cumulatively very large, but whose individual parts are very small; when such operations are implemented naively using the typical approaches, they perform poorly. To address the challenges, we designed a standard for Hybrid Batched BLAS [7], and developed innovative algorithms [12], data and task abstractions [1], as well as high-performance implementations based on the standard that are now released through MAGMA 2.0 [6, 11]. Figure 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, as already noted: the overall computation is very large but is made of operations of interest that are in general small, must be batched for efficiency, and various transformations must be explored to cast the batched small computations to regular and therefore efficient to implement operations, e.g., GEMMs. We note that applications in big data analytics and machine learning target higher dimension and accuracy computational approaches (e.g., ab initio-type) that model mutilinear relations, thus, new data abstractions, e.g., tensors, may be better suited vs. the traditional approach of flattening the computations to linear algebra on two-dimensional data (matrices). Indeed, we developed these tensor data abstractions and accelerated the applications using them significantly [1] compared to other approaches.



```
//Declare a 4th-order Tensor A on the GPU
Tensor<64, 64, 9, 8, gpu_t> A;
```

A rank-64 update as **tensor contraction on index k** (**for i** = 0..63 **for j** = 0..63 **for m** = 1..8 **for n** = 1..7):

$$A_{i,j,m,n} -= \sum_k A_{i,k,m,0} A_{k,j,0,n}$$

```
// DSEL design using Einstein notation: repeated
// index k means a summation/contraction.
// Range of the other indices is full/range as
// given through the left assignment operand
A(i, j, m:1..8, n:1..7) -= A(l,k,m,0) * A(k, j,0,n);
```

Convolution of Filters F_i (feature detection) and input image D:

- For every filter F_n and every channel, the computation for every pixel value $O_{n,k}$ is a **tensor contraction**:

$$O_{n,k} = \sum_i D_{k,i} F_{n,i}$$

- Plenty of parallelism; **small operations** must be batched
- Data "reshape" to transform the computation into a **batched GEMM** (for efficiency; among other approaches)

Fig. 1. Left: Example of a 4^{th}-order tensor contractions design using Einstein summation notation and a Domain Specific Embedded Language (or *DSEL*). **Right**: Illustration of batched computations needed in machine learning.

There is a lack of sufficient optimizations on the batched GEMMs needed and targeted in this paper. We show this in comparison to vendor libraries like CUBLAS for NVIDIA GPUs and MKL for Intel multicore CPUs. Related work on GEMM and its use for tensor contractions [1] target only GPUs and for very small sizes (16 and below). Batched GEMM for fixed and variable sizes in the range of $\mathcal{O}(100)$ and smaller were developed in [2]. The main target here are batched GEMMs for multicore CPUs, ARM, Intel Xeon Phi, and GPU architectures on matrices of sizes up to 32.

2 Contributions to the Field

The evolution of semiconductor technology is dramatically transforming the balance of future computer systems, producing unprecedented changes at every level of the platform pyramid. From the point of view of numerical libraries, and the myriad of applications that depend on them, three challenges stand out: 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. Below, we highlight our main contributions related to the algorithm's design and optimization strategies aimed at addressing these challenges on multicore CPU, ARM, Xeon Phi, and GPU architectures.

2.1 Exploit Parallelism and Vector Instructions:

Clock frequencies are expected to stay near their current levels, or even decrease to conserve power; consequently, as we already see, the primary method of increasing computational capability of a chip will be to dramatically increase the number of processing units (cores), which in turn will require an increase of orders of magnitude in the amount of concurrency that routines must be able to utilize as well as increasing the computational capabilities of the floating point units by extending it to the classical Streaming SIMD Extensions set (SSE-1, to SSE-4) in the earlier 2000, 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 multisocket $10 - 20$ cores for the Haswell CPU, 4 cores for a Cortex A57 processor, 68 cores for an Intel Knights Landing 7250 (KNL) and 56×64 CUDA cores for the Tesla P100 GPU) to get optimal performance. The techniques and kernels developed are fundamental and can be used elsewhere.

2.2 Hierarchical Communication Techniques that Maximizes the use of Data Locality:

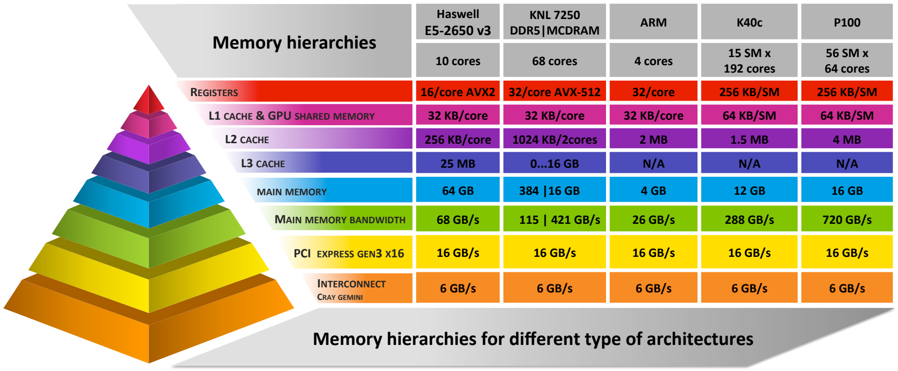
Recent reports (e.g., [9]) have made it clear that time per flop, memory bandwidth, and communication latency are all improving, but at exponentially different rates. So computation on very small matrices, that can be considered as computation-bound on old processors, is, –today and in the future– communication-bound and depends on the communication between levels of the memory hierarchy. We demonstrate that performance is indeed harder to get on new manycore architectures unless hierarchical communications and optimized memory management are considered in the design. We show that, only after we developed algorithmic designs that feature multilevel blocking of the computations and use multilevel memory communications, our implementations reach optimal performance.

2.3 Performance Analysis and Autotuning:

We demonstrate the 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.

3 Experimental hardware

All experiments are done on an Intel multicore system with two 10-cores Intel Xeon E5-2650 v3 (Haswell) CPUs, a 4-cores Cortex A57 ARM CPU, a 68-cores Intel Knights Landaing CPU 7250 and a Pascal Generation Tesla P100 GPU. Details about the hardware are illustrated in Figure 2. We used gcc compiler 5.3.0 for our CPU code (with options -std=c++14 -O3 -avx -fma), as well as the icc compiler from the Intel suite 2016.0.109, and the BLAS implementation from MKL (Math Kernel Library) 16.0.0 [14]. We used CUDA Toolkit 8.0 for the GPU. For the CPU comparison with the MKL library we used two implementations: 1) An 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, and 2) The batched dgemm routine that has been recently added to the MKL library.



Memory hierarchies	Haswell E5-2650 v3	KNL 7250 DDR5|MCDRAM	ARM	K40c	P100
	10 cores	68 cores	4 cores	15 SM x 192 cores	56 SM x 64 cores
Registers	16/core AVX2	32/core AVX-512	32/core	256 KB/SM	256 KB/SM
L1 cache & GPU shared memory	32 KB/core	32 KB/core	32 KB/core	64 KB/SM	64 KB/SM
L2 cache	256 KB/core	1024 KB/2cores	2 MB	1.5 MB	4 MB
L3 cache	25 MB	0...16 GB	N/A	N/A	N/A
main memory	64 GB	384 |16 GB	4 GB	12 GB	16 GB
Main memory bandwidth	68 GB/s	115 | 421 GB/s	26 GB/s	288 GB/s	720 GB/s
PCI express gen3 x16	16 GB/s	16 GB/s	16 GB/s	16 GB/s	16 GB/s
Interconnect Cray Gemini	6 GB/s	6 GB/s	6 GB/s	6 GB/s	6 GB/s

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

4 Methodology, Design, and Optimization : Performance model

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 and T_{min} is the fastest time to solution. For simplicity, consider $C = \alpha AB + \beta C$ on square matrices of size n. We have $F \approx 2n^3$ and $T_{min} = min_T(T_{Read(A,B,C)} + T_{Compute(C)} + T_{Write(C)})$. Note that we have to read/write $4n^2$ elements, or $32n^2$ Bytes for double precision (DP) calculations. Thus, if the maximum achievable bandwidth is B (in Bytes/second), and we assume $T_{Compute(C)} \to 0$ for very small computation, then $T_{min} = T_{Read(A,B,C)} + T_{Write(C)} = 4n^2/B$ in DP. Note that this time is theoretically achievable if the computation totally overlaps the data transfer and does not disrupt the maximum rate B of read/write to the GPU memory. Thus,

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

The achievable bandwidth can be obtained by benchmarks. For our measures, we used the STREAM benchmark [19] 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. Our tests show that the practical CPU bandwidth we are able to achieve using different benchmarks is about 44 GB/s per socket. On the Tesla P100 GPU, the peak is 580 GB/s, so in that case P_{max} is 2.75 n GFlop/s per socket for the CPU and 36.25 n GFlop/s for the Tesla P100 GPU. The curve representing this theoritical maximal limit is denoted by the "upper bound" line on Figures 6 and 14a. Thus, when $n = 16$ for example, we expect a theoretical maximum performance of 580 GFlop/s in DP on the P100 GPU.

5 Programming Model, Performance Analysis, and Optimization for CPUs

Our overall designs and software construction include the use of new features of C++ for better re-usability and adaptability of the code. By using advanced template techniques we can create high-level interfaces [18] 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.

The implementation of a matrix-matrix products kernel for very small matrices for CPUs requires specific design and optimizations. As we can store three double precision matrices of size up to 32×32 in the L1 cache of an Intel Xeon E5-2650 v3 processor (or any modern processor), one can expect that any implementation will not suffer from data cache misses. This can be seen on Figure 9b 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 implementation correspond to the simple matrix product implementation using for loops. The ikj version is cache-friendly as data accessed in a continuous fashion which also gives the possibility to the compiler for vectorization. In ijk version, data is not accessed contiguously but we can minimize the number of store operations by computing one value of C for each iterations of the innermost loop. For smaller sizes, the ijk implementation is more efficient than the ikj one, as it optimizes the number of stores (Figure 4a). To obtain a near optimal performance, we conduct an extensive study over the performance counters using the PAPI [22] tools. Our analysis concludes that in order to achieve an efficient execution for such computation, 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/register based loop transformations and, consequently, this kind of optimization should be studied and implemented by the software developer [16]. 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. In this CPU section, we use a batch count of 10000 which is enough to get the best performance for small size matrix-matrix products.

5.1 Programming techniques using C++14

The development of programming languages and their use has dramatically changed in recent years leading to continuous evolution. C++ is an example of such a programming language. The standardization committee has decided to make a new standard every 3 years, with the next release being the C++17 standard. 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` [15]. Consider the following declaration in Listing 1.1:

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

Listing 1.1. C++ auto

Here x will have the type `int` because it is the type of its initializer. In general, we can write the code in Listing 1.2

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

Listing 1.2. C++ generic auto

and x will be of the type from the value expression in Listing 1.2. For any variable, `auto` specifies that the type of the variable that is being declared will be automatically deduced from its initializer. This allows 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 [8]. 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 1.3, the fibonnaci function is guaranteed to be executed at compile time if the value passed x is available at compile time.

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

Listing 1.3. C++ constexpr

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 and ARM AArch64) while using a generic function for each call (see Listing 1.4).

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

// Defines the load operation for 256-bit simd
inline auto load(float const* ptr , std::integral_constant<unsigned long
    ,256> )
{
  return _mm256_loadu_ps(ptr);
}

// Defines the load operation for 512-bit simd on KNL
inline auto load(float const* ptr , std::integral_constant<unsigned long
    ,512> )
{
  return _mm512_loadu_ps(ptr);
}

#endif

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

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

#endif
```

Listing 1.4. C++ SIMD load

If we then want to do a multiplication using SIMD instructions, we can simply use the standard operator with our overloaded functions (see Listing 1.5).

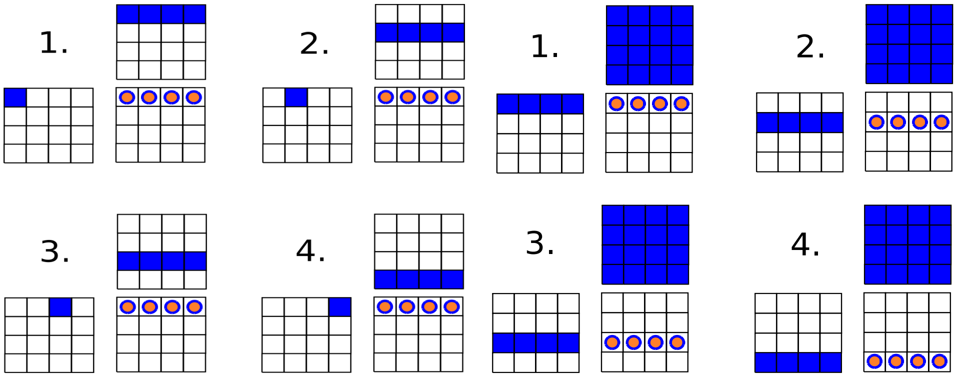
```
using simd_size = std::integral_constant<unsigned long,512>
// Propagate the value at A[iA*N] in the 512 SIMD register tmp
auto tmp = set( A[iA*N] , simd_size{} );
// Load B[i]..B[i+simd_size] and multiply
auto C = tmp *  load(&B[i] , simd_size{});
```

Listing 1.5. C++ multiply operation

These programming techniques allow us to have a single source file of around 400 lines that support Intel and ARM processors for very efficient small size matrix products. It is also very simple to extend. For example, adding support for IBM processors with Altivec SIMD instructions only requires us to add an overload for each SIMD functions we need.

5.2 Data Access Optimizations and Loop Transformation Techniques

In our design, we propose to order the iterations of the nested loops in such a way 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. Hence, loop unrolling, loop peeling, and loop interchange can be useful techniques for such algorithm [3, 4]. 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 inner-most 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 from the knowledge of the algorithm (see Figure 3), and based on the principle of locality of references [13], to optimize both the temporal and spatial data locality.



(a) Compute first row of matrix C (b) Compute all of matrix C

Fig. 3. Example of a 4-by-4 matrix product using SIMD

5.3 Register Data Reuse and Locality

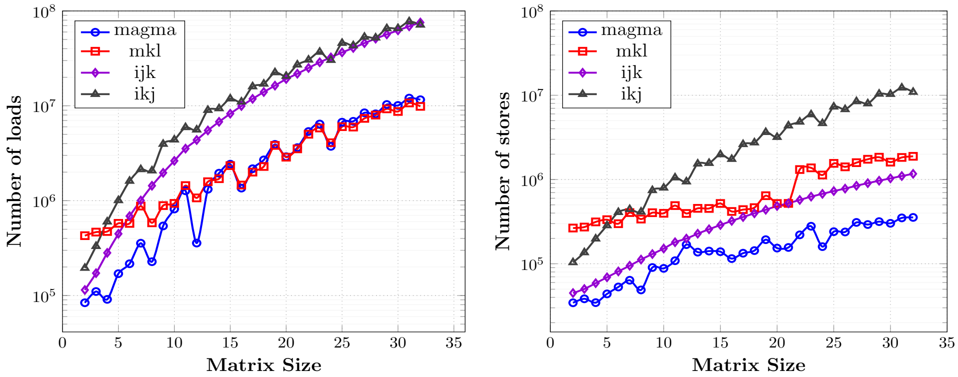
Similarly to the blocking strategies for better cache reuse in numerically intensive operations (e.g., large matrix-matrix products), we focus on register blocking to increase the performance. Our study concludes that the register reuse ends up being the key factor for performance. The idea is that when data is loaded into SIMD register, it will be reused as much as possible before its replacement by new data. The amount of data that can be kept into registers becomes an important tuning parameter. For example, an 8×8 matrix requires 16 256-bit AVX-2 registers to be completely loaded. As the targeted hardware consists of only 16 256-bit AVX-2 registers, one can expect that loading the whole B will not be optimal as we will have to reload the vectors for A 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. We propose an auto-tuning process to check all the possible scenarios and provide the best option. This reduces the number of load, store, and total instructions from $O(n^2)$ to $O(n)$, compared to a classical ijk or ikj implementation as depicted in Figures 4a, 4b, and 6a, respectively.

5.4 Algorithmic Advancements

Algorithm 1 is an example of our methodology for a matrix-matrix product of 16×16 matrices. In this pseudo-code, 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 such a way to minimize the load/store as shown in Figure 4. Each C ends up being loaded/stored once. We apply this strategy to matrix sizes ranging from 8 to 32 as for smaller sizes the whole matrix can fit in registers. Different blocking strategies (square versus rectangular) have been studied through our auto-tuning 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.

```
 1: Load B0, B1, B2, B3
 2: Load $\alpha$, $\beta$
 3: S = 16
 4: for i = 0, 1, ... , S-1 do
 5:     Load A[i*S]
 6:     Mi0 = A[i*S] * B0; ... Mi3 = A[i*S] *B3
 7:     for u = 1, 2, ... , S-1 do
 8:         Load A[i*S + u]
 9:         Load Bu0, Bu1, Bu2, Bu3
10:         Mi0 += A[i*S+u] * Bu0; ... Mi3 += A[i*S+u] *Bui3
11:     end for
12:     Mi0 = $\alpha$ Mi0 + $\beta$ (Load Ci0); ... Mi3 = $\alpha$ Mi3 + $\beta$ (Load Ci3)
13:     Store Mi0, Mi1, Mi2, Mi3
14: end for
```

Algorithm 1: Generic matrix-matrix product applied to matrices of size 16 × 16



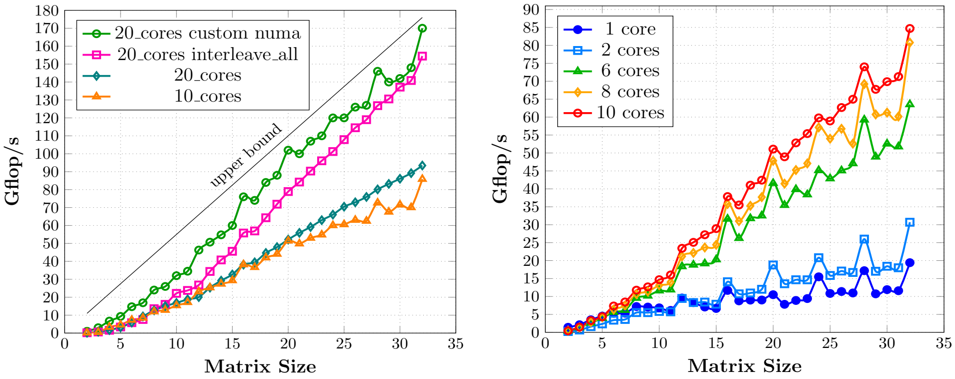
(a) # of load instructions

(b) # of store instructions

Fig. 4. CPU Performance counters measurement of the memory accesses on Haswell

5.5 Effect of the Multi-threading

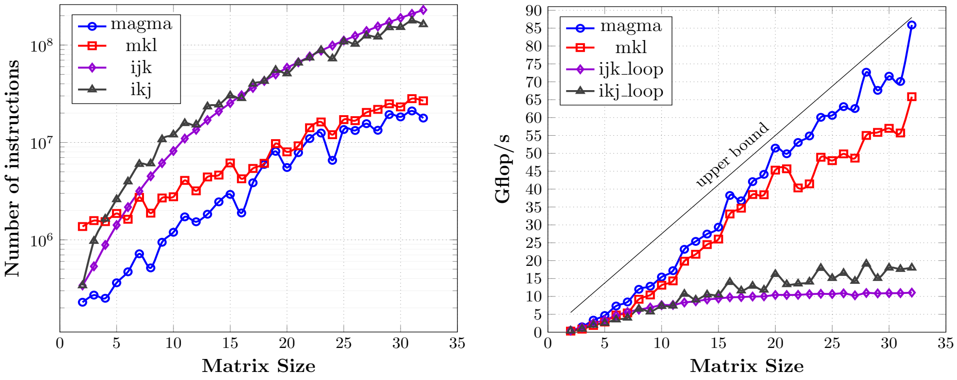
As described above, operating on matrices of very small sizes is memory-bound computation and thus, increasing the number of CPU cores may not always increase the performance since the performance will be limited by the bandwidth which can be saturated by a few cores. We performed a set of experiments towards clarifying this behavior and illustrate our findings in Figure 5b. 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 18 GB/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 44 GB/s.



(a) Effect of the NUMA memory management

(b) Effect of the number of CPU cores

Fig. 5. CPU Performance analysis on Haswell



(a) Total CPU instruction count

(b) CPU Performance comparison

Fig. 6. Experimental results of the matrix-matrix multiplication on Haswell

5.6 Effect of the NUMA-socket and Memory Location

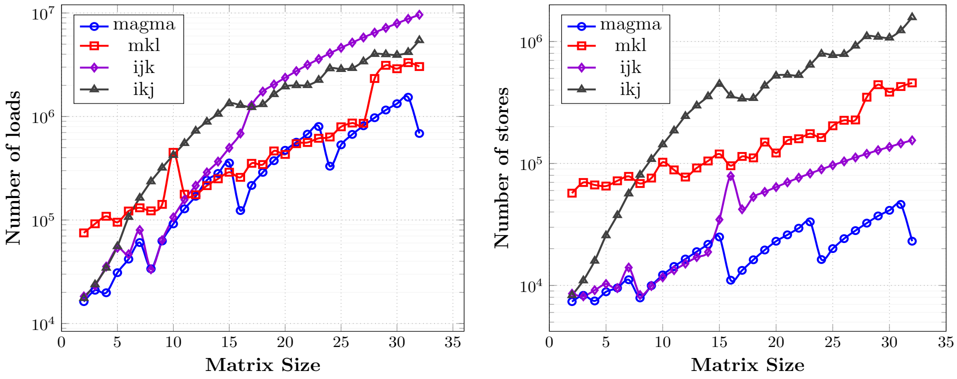
We also studied NUMA-socket (non-uniform memory access) [10] when using two Xeon sockets as seen in Figure 5a. A standard memory allocation puts all of the data in the memory slot associated to the first socket until it gets filled, then starts filling the second socket. Since the problem size we are targeting is very small, most of the data is allocated on one socket, and thus using extra 10 cores of the second socket 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 thus is limited by the bandwidth of one socket (44 GB/s). There are ways to overcome this issue. By using NUMA with the interleave=all option, which spreads the allocation over the two sockets by memory pages, we can improve the overall performance. However, for very small sizes, we observe that such solution remains far from the optimal bound since data is spread out over the memory of the two sockets without any rules 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 Figure 5a, this allows our implementation to scale over the two sockets and to reach close to the peak bound.

5.7 Application to the Intel KNL

The Intel KNL is a new architecture that provides improved hardware features such as 512-bit vector units, 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 [20]. An extensive study to apply the Roofline Performance Model [23] on the KNL [5] has shown the differences between the MCDRAM configurations and its impact on performance. Our study has ended up with 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.

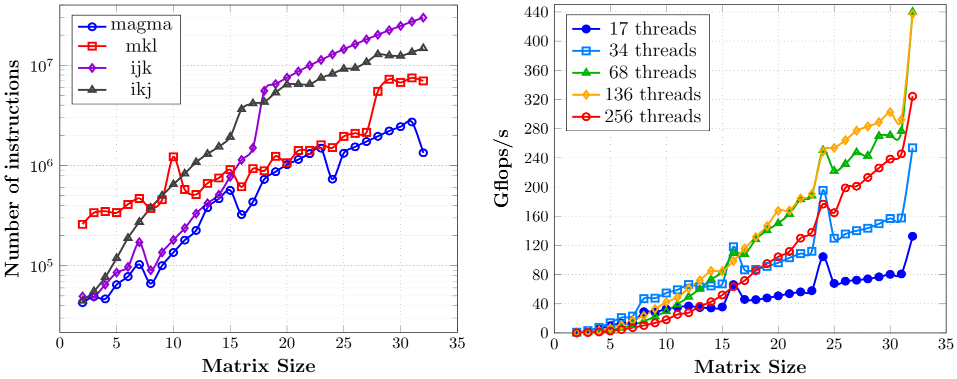
We can see in Figure 7 that the number of load 7a and store 7b instructions are following the same pattern as with the Haswell processor. The important drops we see on each graph for the KNL are a bit different than on the Haswell processor. This is due to the size of the vector unit going from 256-bit to 512-bit. For double precision operations, we see on every multiple of 8 a large drop in the number of load/store due to the matrix size being a multiple of the SIMD size.

We generally reach the same number of load instructions as the MKL since we cannot really optimize this parameter as seen with the Haswell CPU. However, by not using a standard blocking strategy, we are able to optimize even more the number of stores operations compared to the Haswell CPU due to the larger SIMD vector size. We can see on Figure 8a that we always have a lower total instruction count. On the KNL, it is possible to have up to 4 threads per core.



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

Fig. 7. CPU Performance counters measurement of the memory accesses on KNL

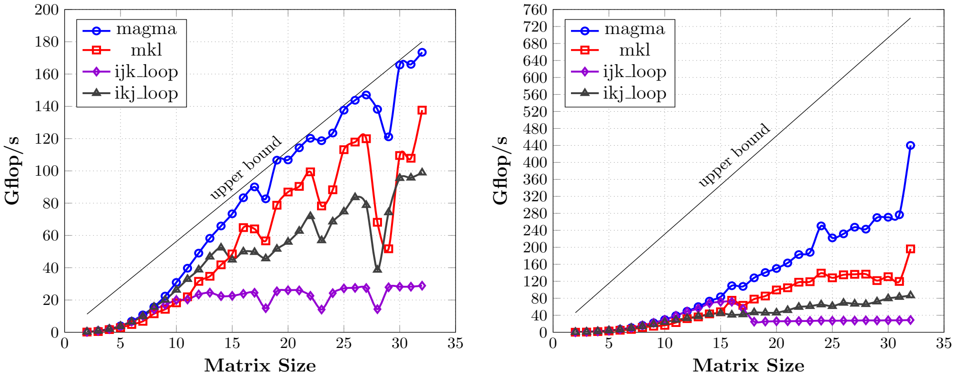


(a) Total KNL instruction count (b) Effect of the number of CPU cores

Fig. 8. CPU performance counters and scaling analysis on KNL

Using the maximum number of threads is never efficient as see on Figure 8. Using 2 threads per core can sometime yields 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 translates on the performance obtained (see Figure 9). The performance with our generated code in MAGMA is always better than that of the MKL. We can see that the use of the MCDRAM as the main memory instead of the DDR4 heavily impacts the performance. We observe an overall performance increase of two when using the MCDRAM. We end up far from the upper bound due to data in the MCDRAM not being read multiple times which limits the bandwidth usage. Using the MCDRAM also leads to more stable performance. Memory bound problems tend to be less stable in terms of performance when using SIMD instructions and multi-threading due to limited resources. This is even more prevalent with the KNL as its very large SIMD instructions (512-bit) corresponds to the size of an L1 cache line (64 Bytes).



(a) CPU Performance comparison without MCDRAM

(b) CPU Performance comparison with MCDRAM

Fig. 9. Experimental results of the matrix-matrix multiplication on KNL - 68 threads

5.8 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 Figure 10, we compare the performance between our MAGMA code, an ijk code, an ikj code and OpenBLAS [24] using the latest version available from the develop branch on Github.

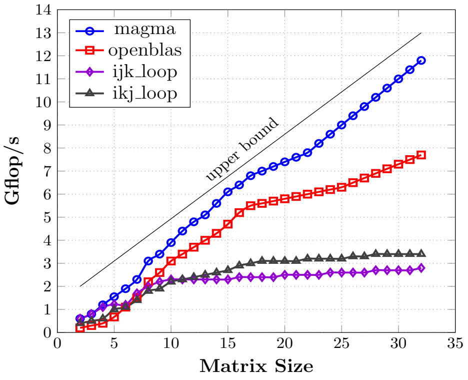


Fig. 10. Experimental results of the matrix-matrix multiplication on the Tegra X1

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 version stale and reach a limit set around 3.5 Gflops. The OpenBLAS version provides good performance but is limited by its blocking model which is not adapted for very small sizes.

6 Programming Model, Performance Analysis, and Optimization for GPUs

Considering the development for GPUs, we set a goal to have a unified code base that can achieve high performance for very small sizes. The design strategy is different from the MAGMA batched GEMM kernel for medium and large sizes [2]. 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 C. Obviously, there will be redundant reads of both A and B among TBs. Considering extremely small sizes (e.g. up to 32), redundant reads cannot be afforded, since the memory bandwidth becomes the main bottleneck for such computational workload.

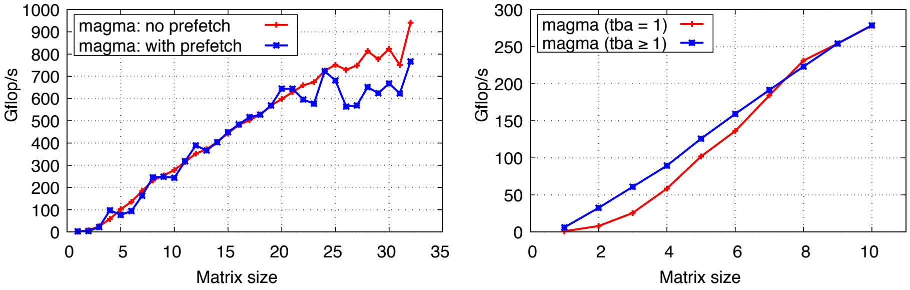
Instead, we adopt a strategy where a single TB performs the entire multiplication of at least one problem, computing all of C with no subdivision involved. We start by an initial design that represents a special case of a $1{\times}1$ blocking technique, in order to avoid redundant reads from global memory. Since the sizes considered are very small, there is enough resources on the SM to store all of A, B, and C in shared memory and/or registers. Similar to the design proposed in [2], 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 auto-tuning and performance counter analysis, to improve its performance on the Tesla P100 GPU over the original design proposed in [17].

6.1 A Shared Memory Approach

Our previous work [17] showed that using shared memory to exploit data reuse is superior to using the read-only data cache. We start by 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 2D 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.

6.2 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, Figure 11a shows that data prefetching does not result in performance gains except for slight improvements for few certain sizes. We list two major reasons for this behavior. The first is that the prefetching technique uses 4× 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.



(a) Data prefetching

(b) Aggregation

Fig. 11. Impact of data prefetching and aggregation on performance. The experiment is performing 100,000 `GEMM` operations in double precision on a Tesla P100 GPU.

6.3 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 assigs 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`). Figure 11b shows the impact of aggregation (after tuning `tba` for every size) on performance, where we observe performance improvements on sizes less 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.

6.4 Recursive Blocking

We propose a new optimization technique that helps improve the performance as the sizes get larger. For a multiplication of size N, the original design uses N×N threads and $2N \times N$ of shared memory per TB. This configuration can limit the number of TBs that can execute concurrently per SM. In order to mitigate this effect, we recursively block the computation in 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:

$$1 < \frac{N}{\hat{N}} \leq 2 \; . \tag{1}$$

The kernel reads A, B, and C once into registers. Since the shared memory resources can only accomodate two $\hat{N} \times \hat{N}$ blocks, the computation is performed on several stages. Equation 1 ensures a 2×2 blocking of the form:

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

We point out that 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 is exactly the same for A and B. The scaling with β is done upon reading C. In order to compute C_{00}, the kernel loads A_{00} and B_{00} into shared memory to compute $\alpha A_{00} \times B_{00}$, and then similarly $\alpha A_{01} \times B_{10}$. The accumulation occurs in the registers holding $\beta \times C_{00}$. The computation of other blocks of C is performed similarly to C_{00}. Eventually, the kernel is performing one `GEMM` operation using much less 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 accomodate such increase. The overall result is an improved performance for relatively larger sizes as shown in Figure 12. In the range of sizes [20:32], we observe speedups ranging from 3% up to 31%. The non-blocking kernel loses performance as we increase the sizes, unlike the blocking kernel which maintains a steady performance.

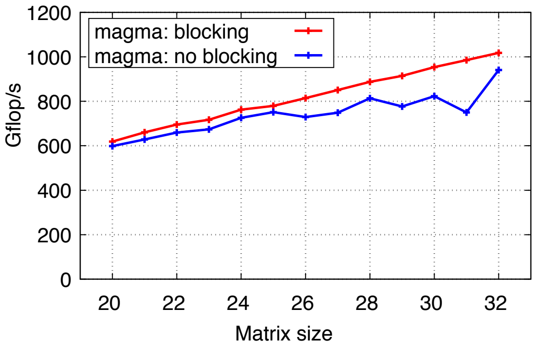
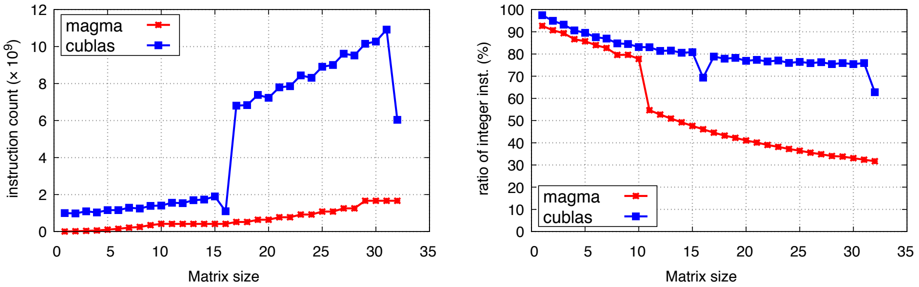


Fig. 12. Impact of recursive blocking on performance. The experiment is performing 100,000 `GEMM` operations in double precision on a Tesla P100 GPU.

Eventually, our solution combines all of the aforementioned techniques, with the exception of data prefetching. We can subdivide the range [1:32] into three segments. The first represents the tiny sizes in the range [1:10], where we use a non-blocking kernel with `tba`$>$ 1. The second is the midrange [11:19], where we still use the non-blocking kernel, but setting `tba` $= 1$. The third is the relatively larger sizes in [20:32], where we call the blocking kernel.

6.5 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 such 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 predefined boundary can be easily unrolled and optimized by the Nvidia compiler. Using CUDA C++ templates that are instantiated with 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. Figure 13 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 less integer instructions than CUBLAS. It also has the smallest ratio for most 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.



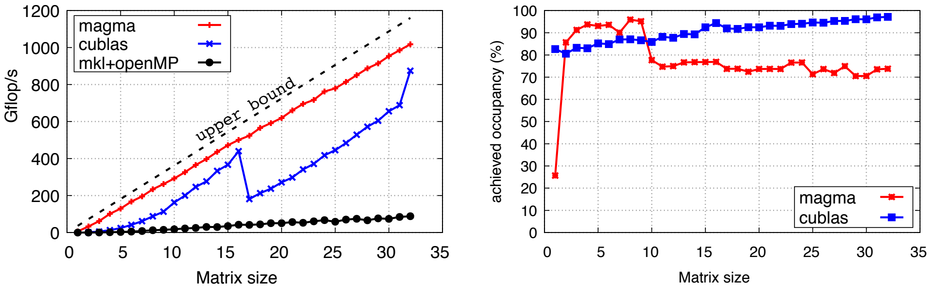
(a) Integer instruction count (b) Percentage of integer instructions

Fig. 13. Profiling the instruction mix of MAGMA versus CUBLAS. The experiment is performing 100,000 `GEMM` operations in double precision on a Tesla P100 GPU.

6.6 Performance and Profiling Results

Figure 14a shows the final performance of the proposed kernel against the batched `GEMM` kernel from CUBLAS, as well as against a CPU reference implementation that combines MKL with OpenMP. We also show the upper bound of the performance, as estimated in Section 4. 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. Similarly, MAGMA speedups against the CPU implementation range from 11.4× up to 164.4×. On another hand, the MAGMA kernel is up to 88% close to the performance upper bound.

An interesting observation is shown in Figure 14b, which shows that the the CUBLAS kernel achieves higher occupancy than the MAGMA kernel, starting

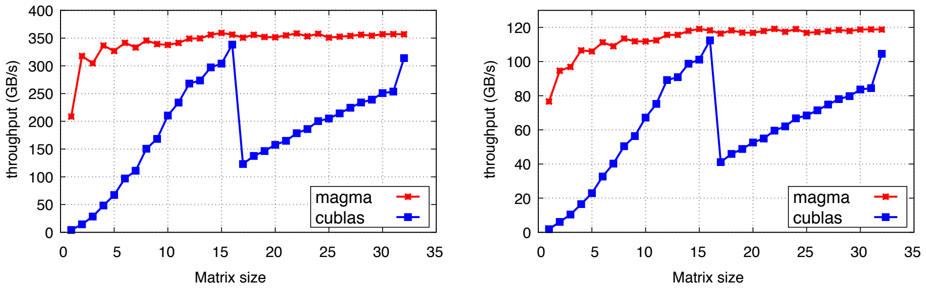


(a) Final Performance

(b) Achieved Occupancy

Fig. 14. Final performance and achieved occupancy. The experiment is performing 100,000 `GEMM` operations in double precision on a Tesla P100 GPU.

from size 10. We point out that the achieved occupancy metric does not necessarily give good insight to 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, Figure 13 shows that the CUBLAS kernels executes a lot more integer instructions than the MAGMA kernel. Moreover, since the computation is memory bound, we show a more representative metric. Figure 15 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.



(a) Read throughput

(b) Write throughput

Fig. 15. DRAM read and write throughputs. The experiment is performing 100,000 `GEMM` operations in double precision on a Tesla P100 GPU.

7 Conclusions and future directions

We presented work motivated by a large number of applications, ranging from machine learning to big data analytics, that require fast linear algebra on many independent problems that are of size 32 and smaller. The use of batched GEMM for small matrices is fundamental for obtaining high performance in applications like these. We presented specialized algorithms for these cases – where the overall computation is memory bound but still must be blocked – to obtain performance that is within 90% of the optimal, significantly outperforming currently available state-of-the-art implementations and vendor-tuned math libraries. Here, the optimal is the time to just read the data once and write the result, disregarding the time to compute. The algorithms were designed for modern multi-core CPU, ARM, Xeon Phi, and GPU architectures. The optimization techniques and algorithms can be used to develop other batched Level 3 BLAS and to accelerate numerous applications that need linear algebra on many independent problems.

Future work includes further optimizations and analyses, e.g., on how high performance can go using CUDA. It is known that compilers have their limitations in producing top performance codes for computations like these, thus, requiring the use of lower level programming languages. Current results used intrinsics for multi-core CPUs and CUDA for GPUs, combined with auto-tuning in either case, to quickly explore the large algorithmic variations developed in finding the fastest one. Future work includes also use in applications, development of application-specific optimizations, data abstractions, e.g., tensors, and algorithms that use them efficiently.

Acknowledgments

This material is based in part upon work supported by the US NSF under Grants No. CSR 1514286 and ACI-1339822, NVIDIA, the Department of Energy, and in part by the Russian Scientific Foundation, Agreement N14-11-00190.

References

1. Abdelfattah, A., Baboulin, M., Dobrev, V., Dongarra, J., Earl, C., Falcou, J., Haidar, A., Karlin, I., Kolev, T., Masliah, I., Tomov, S.: High-Performance Tensor Contractions for GPUs. In: International Conference on Computational Science (ICCS'16). Elsevier, Procedia Computer Science, San Diego, CA, U.S.A. (06 2016)
2. Abdelfattah, A., Haidar, A., Tomov, S., Dongarra, J.: Performance, Design, and Autotuning of Batched GEMM for GPUs. In: The International Supercomputing Conference (ISC High Performance 2016). Frankfurt, Germany (06 2016)
3. Ahmed, N., Mateev, N., Pingali, K.: Tiling imperfectly-nested loop nests. In: Supercomputing, ACM/IEEE 2000 Conference. pp. 31–31 (Nov 2000)
4. Bacon, D.F., Graham, S.L., Sharp, O.J.: Compiler transformations for high-performance computing. ACM Comput. Surv. 26(4), 345–420 (Dec 1994)
5. Doerfler, D., Deslippe, J., Williams, S., Oliker, L., Cook, B., Kurth, T., Lobet, M., Malas, T., Vay, J.L., Vincenti, H.: Applying the roofline performance model to the intel xeon phi knights landing processor. In: International Conference on High Performance Computing. pp. 339–353. Springer (2016)
6. Dong, T., Haidar, A., Luszczek, P., Harris, A., Tomov, S., Dongarra, J.: LU Factorization of Small Matrices: Accelerating Batched DGETRF on the GPU. In: Proceedings of 16th IEEE International Conference on High Performance and Communications (August 2014)
7. Dongarra, J., Duff, I., Gates, M., Haidar, A., Hammarling, S., Higham, N.J., Hogg, J., Valero-Lara, P., Relton, S.D., Tomov, S., Zounon, M.: A proposed API for

Batched Basic Linear Algebra Subprograms. MIMS EPrint 2016.25, Manchester Institute for Mathematical Sciences, The University of Manchester, UK (Apr 2016), http://eprints.ma.man.ac.uk/2464/

8. Dos Reis, G., Stroustrup, B.: General constant expressions for system programming languages. In: Proceedings of the 2010 ACM Symposium on Applied Computing. pp. 2131–2136. ACM (2010)
9. Fuller, S.H., Lynette I. Millett, E.C.o.S.G.i.C.P.N.R.C.: The Future of Computing Performance: Game Over or Next Level? The National Academies Press (2011), http://www.nap.edu/openbook.php?record_id=12980
10. Hager, G., Wellein, G.: Introduction to High Performance Computing for Scientists and Engineers. CRC Press (2011)
11. Haidar, A., Dong, T., Luszczek, P., Tomov, S., Dongarra, J.: Batched matrix computations on hardware accelerators based on gpus. International Journal of High Performance Computing Applications (2015), http://hpc.sagepub.com/content/early/2015/02/06/1094342014567546.abstract
12. Haidar, A., Dong, T., Tomov, S., Luszczek, P., Dongarra, J.: A Framework for Batched and GPU-Resident Factorization Algorithms Applied to Block Householder Transformations. In: High Performance Computing, Lecture Notes in Computer Science, vol. 9137, pp. 31–47 (2015), http://dx.doi.org/10.1007/978-3-319-20119-1_3
13. Hennessy, J.L., Patterson, D.A.: Computer Architecture, Fifth Edition: A Quantitative Approach. Morgan Kaufmann Publ. Inc., San Francisco, CA, USA (2011)
14. Intel Math Kernel Library (2016), available at http://software.intel.com
15. Järvi, J., Stroustrup, B.: Decltype and auto (revision 3). ANSI/ISO C++ Standard Committee Pre-Sydney mailing (1607), 04–0047 (2004)
16. Loshin, D.: Efficient Memory Programming. McGraw-Hill Profess., 1st edn. (1998)
17. Masliah, I., Abdelfattah, A., Haidar, A., Tomov, S., Baboulin, M., Falcou, J., Dongarra, J.: 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. pp. 659–671 (2016), http://dx.doi.org/10.1007/978-3-319-43659-3_48
18. Masliah, I., Baboulin, M., Falcou, J.: Metaprogramming dense linear algebra solvers applications to multi and many-core architectures. In: 2015 iIEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August,. vol. 3, pp. 69–76 (2015)
19. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter pp. 19–25 (Dec 1995)
20. Sodani, A.: Knights landing (knl): 2nd generation intel® xeon phi processor. In: Hot Chips 27 Symposium (HCS), 2015 IEEE. pp. 1–24. IEEE (2015)
21. Tomov, S., Dongarra, J., Baboulin, M.: Towards dense linear algebra for hybrid GPU accelerated manycore systems. Parallel Computing 36(5-6), 232–240 (2010)
22. Weaver, V., M.Johnson, K.Kasichayanula, J.Ralph, P.Luszczek, D.Terpstra, S.: Measuring energy and power with PAPI. In: 41st International Conference on Parallel Processing Workshops (September 2012)
23. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Communications of the ACM 52(4), 65–76 (2009)
24. Xianyi, Z., Qian, W., Chothia, Z.: Openblas. URL: http://xianyi. github. io/OpenBLAS (2012)