Improving the Performance of DGEMM with MoA and Cache-Blocking

Preprint

Stephen Thomas,[1] Lenore Mullin,[2] and
Katarzyna Swirydowicz[3]

1 National Renewable Energy Laboratory
2 University of Albany
3 Pacific Northwest National Laboratory

Presented at ARRAY '21
June 20–26, 2021

NREL is a national laboratory of the U.S. Department of Energy
Office of Energy Efficiency & Renewable Energy
Operated by the Alliance for Sustainable Energy, LLC

Conference Paper
NREL/CP-2C00-80232
February 2022

Contract No. DE-AC36-08GO28308

Improving the Performance of DGEMM with MoA and Cache-Blocking

Preprint

Stephen Thomas,[1] Lenore Mullin,[2] and
Katarzyna Swirydowicz[3]

1 National Renewable Energy Laboratory
2 University of Albany
3 Pacific Northwest National Laboratory

Suggested Citation
Thomas, Stephen, Lenore Mullin, and Katarzyna Swirydowicz. 2022. *Improving the Performance of DGEMM with MoA and Cache-BlockingTitle: Preprint*. Golden, CO: National Renewable Energy Laboratory. NREL/CP-2C00-80232. https://www.nrel.gov/docs/fy22osti/80232.pdf.

Conference Paper
NREL/CP-2C00-80232
February 2022

National Renewable Energy Laboratory
15013 Denver West Parkway
Golden, CO 80401
303-275-3000 • www.nrel.gov

NOTICE

This work was authored in part by the National Renewable Energy Laboratory, operated by Alliance for Sustainable Energy, LLC, for the U.S. Department of Energy (DOE) under Contract No. DE-AC36-08GO28308. Funding provided by the U.S. Department of Energy Office of Science, Advanced Scientific Computing Research.

Cover Photos by Dennis Schroeder: (clockwise, left to right) NREL 51934, NREL 45897, NREL 42160, NREL 45891, NREL 48097, NREL 46526.

Improving the Performance of DGEMM with MoA and Cache-Blocking

Stephen Thomas
Stephen.Thomas@nrel.gov
National Renewable Energy Lab
Golden, Colorado, USA

Lenore Mullin
lenore@albany.edu
University of Albany
Albany, New York
USAlenore@albany.edu

Katarzyna Świrydowicz
kasia.swirydowicz@pnnl.gov
Pacific Northwest National
Laboratory, Richland, WA
Richland, Washington, USA

ABSTRACT

The goal of this paper is to demonstrate performance enhancements of the high performance dense linear algebra matrix-matrix multiply DGEMM kernel, widely implemented by vendors in the basic linear algebra subroutine BLAS library. The mathematics of arrays (MoA) paradigm due to Mullin (1988) results in contiguous memory accesses in combination with Church-Rosser complete language constructs optimized for target processor architectures [3]. Our performance studies demonstrate that the MoA implementation of DGEMM combined with optimal cache-blocking strategies results in at least a 25% performance gain on both Intel Xeon Skylake and IBM Power-9 processors over the vendor supplied Intel MKL and IBM ESSL basic linear algebra libraries. Results are presented for the NREL Eagle and ORNL Summit supercomputers.

KEYWORDS

Mathematics of Arrays, contiguous memory, cache-blocking

ACM Reference Format:

Stephen Thomas, Lenore Mullin, and Katarzyna Świrydowicz. 2021. Improving the Performance of DGEMM with MoA and Cache-Blocking . In *ARRAY '21: ACM Symposium on Array Programming, 20 – 26 June, 2021, Virtual.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The DGEMM kernel is critical for both dense and sparse linear solver stacks and Exascale physics simulations. Both the Hypre and Trilinos DOE solver frameworks include low synchronization Krylov iterations for linear solvers [9], together with algebraic multigrid preconditioners that rely on BLAS kernels. Sparse direct solvers such as SuperLU employ multi-frontal factorizations that lead to small dense matrices that require a DGEMM kernel [6]. Numerical linear algebra computations in general require fast matrix multiplication for a variety of algorithms. These include optimization, data compression and stochastic gradient descent (SGD) for the acceleration of training algorithms in AI. More recently, lower precision FP-16 tensor-core processors are being provided by graphics processing unit (GPU) vendors such as NVIDIA and our next goal is to extend our approach to these many-core architectures. In the present study the focus is on improving the sustained performance of DGEMM in FP-64 on the Intel Xeon Skylake and IBM Power-9 CPUs.

A recent paper by Antz et. al. [2] reviews mixed precision algorithms for numerical linear algebra, including both direct and iterative (Krylov) solvers. The direct solvers rely on LU, LDL^T and QR matrix factorizations, whereas Krylov solvers are based on Gram-Schmidt orthogonalization algorithms. The most widely known iterative Krylov solver algorithms are the symmetric Lanczos and non-symmetric Arnoldi-QR iterations. Dense matrix-matrix multiplication is also required for so-called s-step and block variants of these iterative solvers. In this case the matrices are tall and skinny rather than square with dimensions $N \times N$. All of these solvers would directly benefit from fast matrix-vector and matrix-matrix multiplication kernels.

Mathematics of Arrays (MoA) is a way of describing and representing arrays, of any dimension, and is a collection of algebraic operations on arrays [8]. MoA is based on the Psi calculus developed by Mullin in [8]. Psi calculus is, simply, a calculus of indexing and shapes. MoA has several advantages that make it attractive. First, it is domain agnostic. Second, no matter what the array dimensions are, MoA accesses the arrays in a contiguous fashion. This makes it very memory, and cache-friendly. The overall performance of a program based on MoA is predictable. Third, the steps from the high-level description of the problem to program generation can be fully automated due to linear and multi-linear transformations [3].

The mathematics of arrays paradigm results in contiguous memory accesses in combination with Church-Rosser complete language constructs optimized for target processor architectures. We demonstrate that the MoA implementation of matrix-matrix multiply (DGEMM) combined with cache-blocking strategies results in at least a 25% performance gain on both Intel Xeon Skylake and IBM Power-9 processors over the vendor supplied Intel MKL and IBM ESSL basic linear algebra, which contain optimized implementations of the BLAS and LaPACK libraries. Modern processor architectures such as these provide SIMD vector arithmetic units with fused multiply-add instructions. In the case of the IBM Power-9 the SMT vector units are reconfigurable. The Power-9 core comes in two variants, a four-way multi-threaded SMT-4 and an eight-way SMT-8. These can be utilized in slices for vector processing with a 12-stage pipeline. Similar gains are anticipated on NVIDIA and AMD GPUs that implement single-instruction multiple thread SIMT architectures. In addition, these are well-suited to tensor based mathematics on 2×2 and 4×4 matrix-multiplication tensor-core hardware with low-precision FP-16 arithmetic [1, 5].

2 MATRIX MULTIPLICATION

A basic linear algebra kernel (BLAS) is matrix-matrix multiplication, known as DGEMM in double precision floating point arithmetic and available in numerical linear algebra libraries provided by vendors such as the IBM ESSL and Intel MKL. For matrices A and B with conforming dimensions $n \times p$ and $p \times m$, the resulting matrix $C = A \times B$ has dimensions $n \times m$. Because the DGEMM is such an important component in many applications, much effort is devoted to achieving the highest possible execution rates on current microprocessor and many-core architectures such as GPUs. In the present study our focus is on single-processor performance on the Intel Xeon SkyLake and IBM Power-9 processors.

An example code will be derived below for matrix multiplication. Consider square matrices with dimensions $N \times N$. Given two matrices A and B with elements $a_{i,j}$ and $b_{i,j}$ with $0 \le i, j < N$ their product is

$$(AB)_{i,j} = \sum_{k=0}^{N-1} a_{i,k}\, b_{k,j} = a_{i,1} b_{1,j} + a_{i,2} b_{2,j}, + \cdots + a_{i,(N-1)}\, b_{(N-1),j}$$

A straight-forward ‘C’ implementation of this algorithm is given below: The two input matrices are `mul1` and `mul2`. The result matrix `res` is assumed to be initialized to all zeroes. It is a nice and simple implementation. While `mul1` is accessed sequentially, the inner loop advances the row number for `mul2`. The memory access pattern for the matrix is not stride-1 and leads to slow execution rates because of cache misses. There is one possible remedy one can easily try. Because each element in the matrices is accessed multiple times it might be worthwhile to rearrange or “transpose" the second matrix `mul2` before using it.

$$(AB)_{i,j} = \sum_{k=0}^{N-1} a_{i,k}\, b^T_{j,k} = a_{i,1} b^T_{j,1} + a_{i,2}\, b^T_{j,2} + \cdots + a_{i,(N-1)}\, b^T_{j,(N-1)}$$

After the transposition, both matrices are accessed sequentially. The corresponding ‘C’ code is given below.

```c
double tmp[N][N];

for (i = 0; i < N; ++i)
   for (j = 0; j < N; ++j)
      tmp[i][j] = mul2[j][i];

for (i = 0; i < N; ++i)
   for (j = 0; j < N; ++j)
      for (k = 0; k < N; ++k)
         res[i][j] += mul1[i][k] * tmp[j][k];
```

A temporary variable contains the transposed matrix. This requires touching additional memory, but this cost is, hopefully, recovered because the N non-sequential accesses per column are more expensive (at least on modern hardware). The search for an alternative implementation should start with a close examination of the math involved and the operations performed by the original implementation. Our linear algebra knowledge allows us to see that the order in which the additions for each element of the result matrix are performed is irrelevant as long as each addend appears exactly once. This understanding allows us to look for solutions which reorder the additions performed in the inner loop of the original code.

At the algorithmic level, the matrix multiplication is expressed as the product

$$C = A \times B$$

which is the inner product of arrays A and B to produce the result array C. This high-level representation is transformed, using Psi-calculus operations on shapes, to a Denotational Normal Form (DNF), requiring the least amount of computation and memory access. The following equation gives a DNF for matrix multiplication of A and B.

$$\vec{i} \;\psi\; (\, A \cdot B \,) \equiv (((\; \vec{i} \# \vec{k} \;) \;\psi\; A) \times (\, k\psi\; B\,)$$

Vector $\vec{i}$ means an entire row of the product matrix $(\, A \cdot B\,)$ is accessed. ψ is an index function and the entire row is the result of addition reduction operation of the product of elements of A, accessed through a concatenation (symbol #) of vectors $\vec{i}$ (accessing rows) and $\vec{k}$ accessing columns from B. A detailed discussion of DNF forms can be found in [8]. The DNF notation makes no assumptions about the layout of the arrays in the memory. The next step is to translate this DNF into Operational Normal Form (ONF). In ONF form, the arrays are flattened and accessed linearly using, start, stop, and stride arguments. The ONF shows the details of the array layout and how to generate the addresses. The ONF form can be adjusted to use any hardware features such as vectors, and threads. An example implementation of a transformation to the DNF, is Python-MoA .

The last step relates the ONF to the available hardware using dimension lifting of the arrays indices. For example, with double precision floating arrays on a machine with 128-bit vector instructions, then elements are processed two at a time. Therefore, indices have to be adjusted.

A simple example of a 2×2 matrix multiplication using the classical and MoA formulations illustrates how MoA accesses both matrices linearly, in a contiguous manner [7]. The traditional inner product form is given by

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} \times \begin{bmatrix} 4 & 5 \\ 6 & 7 \end{bmatrix} = \begin{bmatrix} 0 \times 4 + 1 \times 6 & 0 \times 5 + 1 \times 7 \\ 2 \times 4 + 3 \times 6 & 2 \times 5 + 3 \times 7 \end{bmatrix}$$

whereas the MoA formulation is expressed as follows

$$\begin{bmatrix} 0 \times (\, 4,\ 5\,) + 1 \times (\, 6,\ 7\,) \\ 2 \times (\, 4,\ 5\,) + 3 \times (\, 6,\ 7\,) \end{bmatrix} =$$

$$\begin{bmatrix} (\, 0 \times 4 \quad 0 \times 5\,) + (\, 1 \times 6 \quad 3 \times 7\,) \\ (\, 2 \times 4 \quad 2 \times 5\,) + (\, 3 \times 6 \quad 3 \times 7\,) \end{bmatrix}$$

MoA differentiates between the DNF, which describes the arrays by their shapes and uses a function ψ to define indices, and between the ONF which takes into account the arrays layout in memory which is row-major. The resulting ‘C’ code is given below with a linear array for storage. The inner-most loop employs stride-1 accesses for `mul1`.

```c
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < N; k++)
      res[i*N+j] += mul1[i*N+k] * mul2[k*N+j];
```

3 CACHE BLOCKING

The memory hierarchy and cache blocking strategies have a direct impact on the execution rate of matrix-matrix multiply. Our analysis of the memory hierarchy is based on Draper [4]. The cache prefetch strategy and 'C' code presented in this earlier work can be improved by modifications to the inner-most loop pointer arithmetic and array indexing. These changes and the resulting execution rates are presented below.

Let $N = 1000$ and let us examine the actual problem in the execution of the original code. The order in which the elements of `mul2` are accessed is: $(0, 0), (1, 0), \ldots, (N - 1, 0), (0, 1), (1, 1), \ldots$. The elements $(0, 0)$ and $(0, 1)$ are in the same cache line but, by the time the inner loop completes one round, this cache line has long been evicted. For this example, each round of the inner loop requires, for each of the three matrices, 1000 cache lines (with 64 bytes for the Intel Xeon processor). This adds up to much more than the 32k of L1d data cache available.

However, consider when two iterations of the middle loop are combined while executing the inner loop. In this case, two double values from the cache line are used, which is guaranteed to be in the L1d data cache. Thus, the L1d data cache miss rate is cut in half. That is certainly an improvement, however, depending on the cache line size, it still might not be optimal. The Intel Xeon processor has a L1d data cache line size of 64 bytes.

With `sizeof(double)` being 8 this means that, to fully utilize the cache line, the middle loop should be unrolled 8 times. Continuing this analysis, to effectively use the `res` matrix as well, i.e. to write 8 results at the same time, unroll the outer loop 8 times as well. Assume here cache lines of size 64 but the code works also well on systems with 32 byte cache lines since both cache lines are also 100% utilized. In general it is best to hard-code cache line sizes at compile time.

If the binaries are supposed to be generic, the largest cache line size should be employed. With very small L1d data caches this might mean that not all the data fits into the cache but such processors are not suitable for high-performance programs in any case. The resulting code is given below:

```
define SM (CLS / sizeof (double))

for (i = 0; i < N; i += SM)
  for (j = 0; j < N; j += SM)
    for (k = 0; k < N; k += SM)
      for (i2 = 0, rres = &res[i][j],
         rmul1 = &mul1[i][k]; i2 < SM;
         ++i2, rres += N, rmul1 += N)
        for (k2 = 0, rmul2 = &mul2[k][j];
             k2 < SM; ++k2, rmul2 += N)
          for (j2 = 0; j2 < SM; ++j2)
            rres[j2] += rmul1[k2] * rmul2[j2];
```

This code appears to be quite complex. To some extent it is, however, only because it incorporates some tricks that can be expressed in MoA e.g. contiguous array access. The most visible change is that now there are six nested loops. The outer loops iterate with intervals of SM (the cache line size divided by `sizeof(double)`). This breaks up the multiplication into several smaller problems which exhibit better cache locality. The inner loops iterate over the missing indices of the outer loops. There are, once again, three loops. The only difficulty here is that the `k2` and `j2` loops are in a different order. This is done because, in the actual computation, only one expression depends on `k2` but two depend on `j2`.

The rest of the complication here results from the fact that compilers are not proficient when it comes to optimizing array indexing. The introduction of the additional variables `rres`, `rmul1`, and `rmul2` optimizes the code by pulling common expressions out of the inner loops, as far down as possible. The default aliasing rules of the C and C++ languages do not help the compiler making these decisions (unless `restrict` is used, all pointer accesses are potential sources of aliasing).

The input matrices can be arbitrarily large as long as the result matrix fits into memory as well. This is a requirement for a more general solution which has now been achieved. Most modern processors include special support for vectorization. Pipelined vector instructions allow processing of 2, 4, 8, or more values at the same time. These are SIMD (Single Instruction, Multiple Data) operations, augmented by others to get the data in the right form. The SSE2 instructions provided by Intel processors can handle two double values in one operation. The instruction reference manual lists the intrinsic functions which provide access to these SSE2 instructions. Advanced vector extensions AVX-2 instructions process four 64-bit double-precision floating point numbers. AVX-2 instructions utilize 256-bit registers for the vectors, which can be streamed to the vector units.

The matrix multiplication has been optimized through the use of the loaded cache lines. All bytes of a cache line are always used and they are accessed before the cache line is evacuated. It should be noted that, in the last version of the code, there are still cache problems with `mul2`; prefetching may not work. However this cannot be solved without transposing the matrix. Perhaps the cache prefetching units will improve and recognize the access patterns, then no additional change would be needed. An alternative approach is discussed below for the Intel Xeon processor.

The latest generation Intel Xeon and IBM Power-9 processors provide vector instructions. For example, the AVX-2 vector instructions from Intel. These generally work with vectors stored as cache lines or special registers and are employed in our experiments reported in the sequel.

4 INTEL MKL DGEMM ON XEON SKYLAKE

Our performance on the Intel Xeon SkyLake processor was further improved by treating the `mul2` array differently in the code given below. In particular, this array is not addressed using pointer arithmetic but rather with array indexing as in the inner-most loops for the other arrays. The `restrict` keyword in 'C' is employed to indicate to the compiler that aliasing will not occur.

In order to load the Intel `icc` compiler and associated libraries along with the lapack library and BLAS, the following commands were employed

```
module load intel-parallel-studio/cluster.2019.1
module load netlib-lapack/3.8.0
```

Cache pre-fetching was enabled in our code with the `pragma prefetch`. In addition, the inner-most loop was unrolled to a depth of 16, which is twice the recommended value for this architecture. Furthermore, our cache line size parameter SM was set to 16 doubles. Both of these choices lead to higher execution rates from the AVX-2 vector instructions associated with the inner-most loop of our MoA based matrix-multiply kernel. The Intel `icc` compiler options are given below and the resulting executable was run on the NREL Eagle Supercomputer. Note that vector SSE and AVX-2 instructions were enabled for these tests. The `restrict` flag informs the compiler to avoid cache aliasing. These parameters are meant to ensure that the majority of memory references are within the current L1d data cache line.

```
icc -restrict -Ofast -xSSE4.2 -axAVX,CORE-AVX2  -o
    transpose  transpose.c
```

Intel provides a fast CBLAS DGEMM matrix multiply kernel in the math kernel library (MKL). For comparison, a driver for the Intel MKL DGEMM was compiled and compared against our implementation on square matrices ranging in size up to $N = 2500$. The compile options were specified as given below.

```
icc -Ofast  geMMlapack.c `pkg-config --libs --cflags mkl-
    dynamic-ilp64-seq` -Ofast -o geMMlapack
```

The computational complexity of the matrix multiply is $O(N^3)$ for square matrices, with two floating-point operations (flops) appearing in the inner-most loop as a multiply-add. The results of our comparison are displayed in Figure 1, where our cache-blocked MoA based code achieves a 25% faster execution rate versus the Intel MKL DGEMM. The execution rate increases up to 15 GigaFlops/sec, at which point the curve flattens. Further analysis would likely indicate that the available memory bandwidth on the Eagle nodes with two 18-core sockets has been reached.

```
#define SM  16 // 16 (64 / sizeof (double))
#define L2  32 // 32 (64 / sizeof (double))
#define L3  8 //   8 (64 / sizeof (double))


int main(int argc, char** argv)
{
   long long int i, i2, j, j2, k, k2;
   long long int ii, ij;

   long long int N = atoi(argv[1]);
   double res[2*N*N] __attribute__ ((aligned (64)));
   double mul1[2*N*N] __attribute__ ((aligned (64)));
   double mul2[2*N*N] __attribute__ ((aligned (64)));

   double *__restrict__ rres;
   double *__restrict__ rmul1;
   double *__restrict__ rmul2;

   for (i = 0; i < N; i += L2)
      for (j = 0; j < N; j += SM )
         for (k = 0; k < N; k += L3)
            for (i2 = 0, rres = &res[i*N+j],
                 rmul1 = &mul1[i*N+k];
                 i2 < L2; ++i2, rres += N, rmul1 += N)
               for (k2 = 0; k2 < L3; ++k2)
                  #pragma prefetch
                  #pragma ivdep
                  #pragma unroll (16)
                  for (j2  = 0; j2 < SM; ++j2)
                     rres[j2] += rmul1[k2] *
                                 mul2[k*N+j+j2];
}
```

5 IBM ESSL DGEMM ON POWER-9

The Power-9 core comes in two variants, a four-way multi-threaded SMT-4 and an eight-way SMT-8. The SMT-4 and SMT-8 cores are similar, in that they consist of a number of so-called slices fed by common schedulers. A slice is a rudimentary 64-bit single-threaded processing core with load store unit (LSU), integer unit (ALU) and a vector scalar unit (VSU, doing SIMD floating point). A super-slice is the combination of two slices. An SMT-4core consists of a 32 KB L1 cache (1 KB = 1024 bytes), a 32 KB L1d data cache, an instruction fetch unit (IFU) and an instruction sequencing unit (ISU) which feeds two super-slices. An SMT-8 core has two sets of L1 caches and, IFUs and ISUs to feed four super-slices. The result is that the 12-core and 24-core versions of the Power-9 each consist of the same number of slices (96 each) and the same amount of L1d cache.

A Power-9 core, whether SMT-4 or SMT-8, has a 12-stage pipeline (five stages shorter than its predecessor, the Power-8), but aims to retain the clock frequency of around 4 GHz. It is the first processor to incorporate elements of the Power ISA v.3.0 that was released in December 2015, including the VSX-3 instructions.

The IBM `xlc` compiler and associated libraries along with the lapack library and BLAS, are loaded using the following commands

```
module load essl/6.2.1-0
module load xl/16.1.1-4
module load netlib-lapack/3.8.0
```

The IBM `xlc` compiler options are given below and the resulting MoA DGEMM executable was run on the ORNL Summit Supercomputer. Vector instructions were enabled for these tests. The `restrict` flag informs the compiler to avoid cache aliasing.

```
cc -O3 -qalias=restrict -qarch=pwr9 -mcpu=power9
   -qhot -qsimd -qprefetch=aggressive
   -qtune=pwr9  -o transpose transpose.c
```

IBM provides a fast DGEMM matrix multiply kernel in the ESSL library. For comparison, a driver for the IBM ESSL DGEMM was compiled and compared against our implementation on square matrices ranging in size up to $N = 2500$. The compile options were specified as given below.

```
  cc -Ofast -mcpu=power9 -mtune=power9 -qarch=pwr9
     -qhot -qprefetch=aggressive
     -qsimd -qtune=pwr9  -qalias=restrict
     -o geMMlapack geMMlapack.c
     -DUSE_MASS -lessl -L$OLCF_ESSL_ROOT/lib64
```

The performance on the IBM Power-9 processor was improved significantly by using array indexing instead of pointer arithmetic within the inner-most loop in the code given below. Another difference between the IBM and Intel implementations is the smaller cache line size specified as $SM = 8$ for the Power-9. Loop unrolling was applied once again, however, only to a depth of eight on this machine. The `restrict` keyword in 'C' is employed once again to indicate to the compiler that aliasing should not occur.

The execution rates achieved on the ORNL Summit Supercomputer are plotted in Figure 2. A single processor core was employed within one of the Summit nodes. Each node of Summit has two 22-core IBM Power-9 CPUs and six NVIDIA Volta 100 GPUs. The performance increases rapidly as the matrix dimensions are increased and then plateau as the maximum acheivable memory bandwidth is reached on a Summit node.

```c
#define SM  8
#define L2  8
#define L3  8


int main(int argc, char** argv)
{
   // ... Initialize mul1 and mul2
   long long int i, i2, j, j2, k, k2;
   long long int ii, ij;

   long long int N = atoi(argv[1]);
   double *__restrict__ res ;
   double *__restrict__ mul1 ;
   double *__restrict__ mul2 ;

   res = malloc(N*N*sizeof(double));
   mul1 = malloc(N*N*sizeof(double));
   mul2 = malloc(N*N*sizeof(double));

   double *__restrict__ rres;
   double *__restrict__ rmul1;
   double *__restrict__ rmul2;

   for (i = 0; i < N; i += L2)
      for (j = 0; j < N; j += SM )
         for (k = 0; k < N; k += L3)
            for (i2 = 0; i2 < L2; ++i2 )
               for (k2 = 0; k2 < L3; ++k2)
                  #pragma prefetch
                  #pragma ivdep
                  #pragma unroll (8)
                  for (j2  = 0; j2 < SM; ++j2)
                     res[i*N+j+j2] +=
                     mul1[i*N+k+k2] * mul2[k*N+j+j2];
}
```

6 CONCLUSIONS

In this paper, the mathematics of arrays paradigm was applied to the BLAS DGEMM matrix multiplication kernel. DGEMM is a widely used algorithm and plays a control role in numerical linear algebra and AI/ML applications. With the advent of low-precision FP-16 tensor-cores, mixed-precision algorithms can now achieve higher speeds with the same level of accuracy.

For both the Intel Xeon and IBM Power-9 architectures, cache-blocking strategies were combined with vector instructions to achieve significant performance improvement. For example, the sustained execution rates were at least 25% faster than the Intel MKL DGEMM on the NREL Eagle Supercomputer. Differences in the ‘C’ implementations were notable. A mixture of pointer arithmetic and array indexing was best for the Intel, whereas pure array indexing performed best on the IBM Power-9 CPU. Presumably, this is related to the compiler and also the vector instructions.

We have identified a core algorithm that is important in numerical linear algebra, especially for iterative and direct solvers. Our studies to accelerate these algorithms will continue and include many-core architectures such as GPUs. It is notable that the MoA inner-product matrix-multiply in 2D, is defined using the outer product. That said, our methodology also supports the Kronecker product, and this is useful for AI and machine learning and thus requires further investigation.

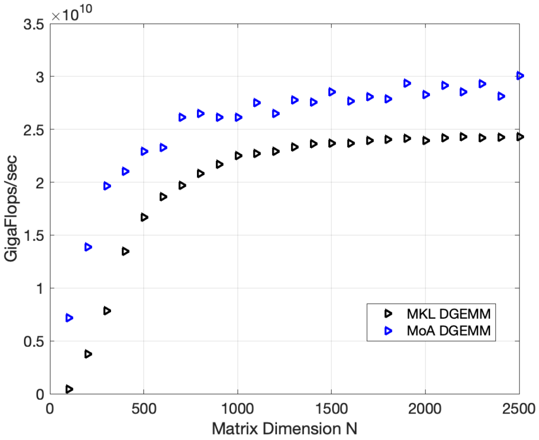

Figure 1: MoA versus Intel MKL DGEMM. NREL Eagle Xeon Skylake

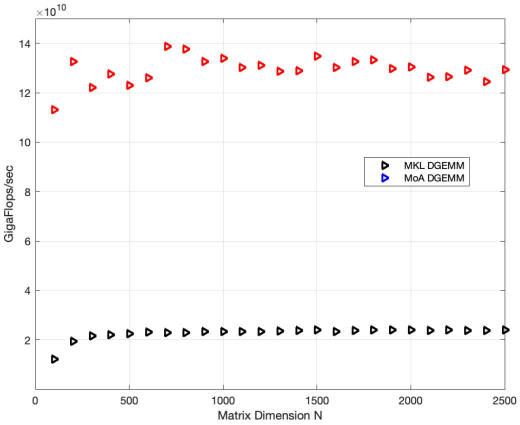

Figure 2: MoA versus IBM ESSL DGEMM. ORNL Summit IBM Power-9

REFERENCES

[1] 2009. Future Directions in Tensor-Based Computation and Modeling. DOI: 10.13140/2.1.4040.4807.

[2] Ahmad Abdelfattah and Hartwig Anzt et. al. 2021. A Survey of Numerical Linear Algebra Methods Utilizing Mixed Precision Arithmetic. *International Journal of High-Performance Computing and Applications* (2021).

[3] A. Church. 1941. *The Calculi of Lambda-Conversion*. Princeton University Press.

[4] U. Draper. 2007. What every programmer should know about memory. (2007). https://people.freebsd.org/~lstewart/articles/cpumemory.pdf

[5] John L. Gustafson and Lenore M. Restifo Mullin. 2017. Tensors Come of Age: Why the AI Revolution will help HPC. *CoRR* abs/1709.09108 (2017). arXiv:1709.09108 http://arxiv.org/abs/1709.09108

[6] Xiaoye S. Li. 2005. An Overview of SuperLU: Algorithms, Implementation, and User Interface. 31, 3 (Sept. 2005), 302–325. https://doi.org/10.1145/1089014.1089017

[7] L. Mullin and M. Zahran. 2020. A case for hardware support for mathematics of arrays. *IEEE Computer Architecture Letters* (2020), 1–4.

[8] L. M. R. Mullin. 1988. *A Mathematics of Arrays*. Ph.D. Dissertation. Syracuse University.

[9] Kasia Świrydowicz, Julien Langou, Shreyas Ananthan, Ulrike Yang, and Stephen Thomas. 2020. Low synchronization Gram-Schmidt and generalized minimal residual algorithms. *Numerical Linear Algebra with Applications* 28 (2020), 1–20.