From array expressions to predictable portable high-performance: foundations for no-code HPC on arrays

Lenore Mullin, Gaétan Hains

From array expressions to predictable portable high-performance: foundations for no-code HPC on arrays

Lenore Mullin, Gaétan Hains

From array expressions to predictable portable high-performance: foundations for no-code HPC on arrays

Lenore Mullin[1] and Gaétan Hains[2]

[1*] , University of Albany, (NY), USA.
[2] Université Paris-Est Créteil, 94000, France.

Contributing authors: lmullin@albany.edu; gaetan.hains@u-pec.fr;

Abstract

In an era of low-code and no-code technologies, when many Computer Science graduates never use MPI or CUDA, explicit parallel programming, compiling and performance tuning appears to become even more of the narrow specialist skill it always was. To improve on this situation we advocate the direct production of high performance executables from a declarative language of array expressions, and we present here the results of an extensive experiment on its basic building block and use-case: matrix multiplication. Dense array algorithms would seem to lack generality as application building blocks, but in fact they are the basis for most of scientific computing, much of today's machine learning and signal processing. Our previous work has shown that the array formalism we use can express all known numerical codes. Our experiments described here show how the choice of algorithm variants, mapping to architecture, vectorization-, multithread parallelization, compiler- options and program annotations can all blend into a mechanizable code-generation + compilation process to produce high and predictable performance. Once perfected and supported by appropriate tools, this methodology will increase productivity for experts and help non-experts make substantial performance gains while avoiding non-determinism and other sources of numerical errors.

Keywords: Efficient code generation, auto-tuning, and optimization for parallel programs. Domain-specific languages: design, implementation and applications.

1 Introduction

Predictable and portable *functional* behaviour of application source code is often taken for granted [1] at least for well-understood algorithms like those of dense numerical algebra, and assuming negligible numerical errors. So it would seem that such algorithms can be programmed once and for all in a high-level portable language, then reused as reliable building blocks in their many applications: scientific computing, machine learning, signal processing, real-time embedded computing etc.

One could argue that parallel execution time is inherently unpredictable, so that so-called parallel cost models are useless. But as every parallel programmer knows, a bad choice of algorithm or mapping to the architecture can lead to execution times that are asymptotically different from the best-known implementations. So execution time variations should be kept within a reasonable and predictable percentage, otherwise expensive parallel hardware and work-intensive parallel coding is wasted.

One attempt to simplify parallel programming and make performance predictable is the bulk-synchronous parallel (BSP) model of computing. In this model, [1, 2] one must define explicit parallel processes and use explicit structured communications (although a shared-memory version exists). BSP

[1]Despite being difficult to guarantee in many cases, as system verification is complex and costly.

programming usually produces deterministic output when the implementation of its structured communications avoids race conditions. That kind of predictability is very useful for parallel software debugging and an important step towards real software engineering. But the BSP model, like MPI, comes at the expense of requiring expertise in algorithmic engineering. This type of programming is appropriate for algorithm experts who want to develop portable parallel libraries (parallel- arrays, sorting, hashing, graphs etc.) with predictable performance. But it is too complex for application programming. BSP library development requires more algorithmic sophistication than classical (sequential) library coding.

In this research we investigate a methodology for parallel computing with similar objectives but higher-level programming and potentially much higher productivity. It is based on the Mathematics of Arrays (MoA) algebra of arrays [3, 4] which defines all potential transformations and compilations of linear-algebra algorithms. Our ultimate goal is to show that the algebraic exploration of equivalent forms of a given array algorithm is sufficient to produce all necessary information for compiling to scalable and predictable parallel code: initially vectorization and SMP multi-thread.

The shapes of arrays, and subarrays (blocks) are assumed sufficient to define the parallel operations, communications *and their mapping* to the parallel architecture because, despite heterogeneous hardware, its structures are mostly made of arrays of computation units. A summary of this approach could be "mapping array operation parts to hardware array parts automatically, based on size-shape information." It is viewing both the data and machines as shapes with indices denoting the data flow, location, and cost that makes this possible and MoA is a theory of shapes and indexing that has had success doing it. The scalar, it's shape, the empty vector, denoted by $<>$ or Θ makes this theory unique. It too has an index and number of components.

In this paper we explore the central notion of predictable high performance for MoA-based code by an exploration of matrix-multiplication variants producing vectorized and multi-threaded from "stereotyped" C code that is equivalent to MoA declarative expressions: nested for-loops whose bounds are defined by the arrays/blocks shapes and MoA operators.

We identify a sufficient "source" language (actually intermediate between array expressions and executable code) made of C loops, vectorization flags, and OpenMP directives. For a given initial "algorithm" (array expression) we can produce "variants" by algebraic transforms in MoA and each of them corresponds to a set of C for-loops in the source language. Exploring performance variations then reduces to exploring the source language programs and input data sizes for executing time. The result is a blueprint for a meta-compiler that will take declarative code and sufficient hardware information to produces reliably high performance on any architecture, present or future.

The next section summarize (a) the MoA description and transformation of the matrix multiplication naive algorithm into variants, using normal form and so-called dimension lifting (b) the target CPU architectures we experimented with (c) the timing experiments (d) the static information from our "source" language and its effect on performance (e) our conclusions and perspectives for future work.

2 Array expressions and the algorithm variants

Here we describe the fragment of MoA used to specify source programs for variants of dense matrix multiplication. The variants are produced from the naive algorithm by so-called *dimension lifting* which is a generalized form of row- and column-blocking for matrix algorithms as used for example in the Bisseling-McColl BSP algorithms [5]. Our current approach is not mechanized but the results support the development of an automatic MoA-to-C (and reverse) translation, automatic compiler directives and pragma generation and associated cost-model to predict and guarantee the most efficient execution within the expressive limits of our compilation and APIs.

The restriction to matrix multiplication (MM) is useful to concentrate on other dimensions of the problem, but the approach will most certainly generalize to other dense linear algebra operations. MM is a generalization of matrix-vector product and, can be generalized to include the Kronecker Product (KP), Hadamard Product (HP), and scalar operations[6] all core for many important algorithms. The reader is also informed that MoA can also express parallel-prefix like transformations of sparse arrays to dense representations, opening the door to sparse equivalents of the work described here.

2.1 Mathematics of Arrays: DNF to ONF to generic design

The C source programs, presented herein, come from the algebraic formulation of the MoA MM, that was then hand-derived (Ψ-Reduced) to a DNF (Denotational Normal), semantic form, that is defined in terms of cartesian coordinates. This proces allows us to systematically, and in the future automatically, produce corresponding C code for vector-parallel execution. Every full-length cartesian coordinate $\vec{i} \equiv < i_0, ..., i_{n-1} >$ where $n \equiv (\delta\xi)$, the dimensionality(δ) of an arbitrary array(ξ), and has a 1-1 correspondence to its layout in memory given shapes of data and architecture. By "layout in memory" we mean the general concept of mapping and distributing data, in this case an array's elements, onto the target hardware architecture which is assumed to be itself a multi-dimensional array of memory vectors. The base level of that *architecture topology* is a single processing unit, for example one CPU core, onto which a 1-dimensional array or vector is mapped. When C code is generated for processing that vector, the result is a for-loop traversing a (flat) C array. The next level in the hierarchy of architecture is a CPU, viewed as a vector of cores, onto which a two-dimensional array is mapped: the first dimension representing a core that will process a line assigned to it in the shared memory. So,

$$\forall \ \vec{i} \ \ s.t. \ \ 0 \leq^* < i_0, ..., i_{n-1} > <^* \ \ (\rho\xi)$$

$$\vec{i}\psi\xi \equiv (\text{rav } \xi)[\gamma_{row}(\vec{i}; \rho\xi)]$$

To the left of $\equiv$ we are in the domain of the Psi Calculus. When a full cartesian coordinacte $\vec{i}$ is used as the left argument to the ψ function it denotes how to obtain an arbitrary scalar using cartesian coordinates. This cartesian coordinate is "full" because it's length is the same size as the dimensionality of the array ξ. The ψ function is at the heart of the Psi Calculus. The bracket notation to the right of $\equiv$ denotes the transformation of a cartesian coordinate in the Psi Calculus, to where it is located in the computer. rav denotes the flatening of an array such that the offset would mean an offset from the address of the begining of the array mentioned. Here, we begin to relate abstract coordinates to their layout in memory. γ denotes a family of mapping functions: row, column, sparse, etc. The bracket notation with "for all" maps directly to the "for" statement in C. We note that C is used as an interface to HDLs[7]. For example For example MoA GEMM was easily implemented on FPGAs using the C programs generated from the ONF directly using Verilog HDL.

2.2 MoA Matrix Multiplication: MoAGeMM

The general matrix-matrix multiplication (`GEMM`) in MoA is a special case of the *inner product* for 2-D arrays (matrices). Define $\mathbf{A}$ as an $m \times n$ matrix, $\mathbf{B}$ as $n \times p$, and $\mathbf{C}$ as $m \times p$. Let the following notation denote the 2-d Matrix Multiplication.

$$\mathbf{C} = \mathbf{A} \bullet \mathbf{B} \tag{1}$$

In MoA notation, the shapes are:

$$\rho\mathbf{A} = \langle m, \, n \rangle \ \ \rho\mathbf{B} = \langle n, \, p \rangle \ \ \rho\mathbf{C} = \langle m, \, p \rangle \tag{2}$$

Next, define the valid indices of the matrices:

$$\forall \, i, j, k \ni \Big\{ 0 \leq i < m \quad 0 \leq j < p \quad 0 \leq k < n \tag{3}$$

$\mathbf{C} = \mathbf{A} \bullet \mathbf{B}$ is defined by the MoA psi expression, given the shapes above.

$$< i > \psi\mathbf{C} \equiv +_{red}(< i, \ k > \psi \ \mathbf{A} \times (< k > \psi B)) \tag{4}$$

When reduced to it's DNF, γ_{row} is applied thus producing the MoA Operational Normal Form (ONF) for `GEMM` and is given by the following "generic" program.

$$\mathbf{C}[(i \times p) + j] := \sum_{k=0}^{n-1} \mathbf{A}[(i \times n) + k] \times \mathbf{B}[(k \times p) + j] \tag{5}$$

Equation (5) is the generic code for a sequential program in MoA. In each of the ith rows of the resultant array **C**, each scalar-vector operation involving the column index j is independent of each other. The ith row of **C** is contiguously filled in by the summation of scalar-vector multiplications involving each matrix element at the ith row and kth column of **A** (the scalar) with each kth row of **B** (the vector) obtained by accessing the arguments contiguously. A row-wise sum reduction is then applied over the k index to yield the final answer as the ith row in **C**. The design is parallel at every step, see Figure 1.

- parallel row operations
- each n parallel row operation is doing m parallel scalar vector multiplies
- all n operations can be summed in $\log n$ steps (ideally).
- everything is accessed contiguously

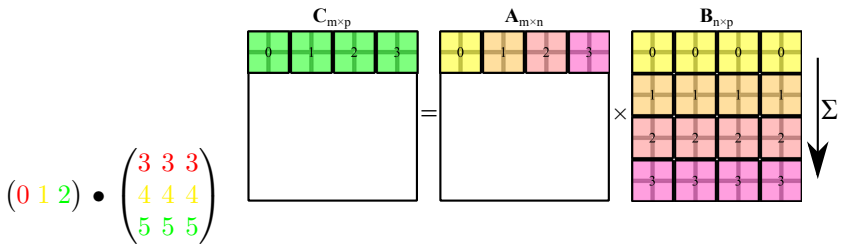


Fig. 1: Visualizing MoAGEMM

2.3 ONF to Generic Sequential Design to Psi Generic Parallel Design

Equation 5 represents the beginning of a predictive general design for the implementation of the MoA MM. What makes this algorithm different from all other MM designs, is that the MoA design always accesses arrays contiguously and when MoA is used to formulate an algorithm, it can be derived, and optimized mechanically.

The concept of **dimension-lifting** is defined by partitioning one or more dimensions into two. That is, given any shape $\vec{s}$, s.t. $\tau\vec{s} = \delta\xi$. The number of components in dimension i, i.e. s_i is partitioned into np parts, thus defining new shapes $s_i \rightarrow < np,\ s_i/np >$ or $< s_i/np,\ np >$. This means that every loop in the ONF can be partitioned into **one** or more loops to match components of the architecture chosen. Begin by partitioning the rows loop of **A** into a two parts: one loop that indexes the processors and the other loop defines how many rows are done sequentially within that processor. The next step is to map these loops to OpenMP menmonics that support the theoretical partitioning in a general way. Also it is essential that the mnemonics chosen provide performance scalability across architectures. Once that is done, the columns loop of **B** and **C** are partitioned into two loops, one that defines the vector register length, and the other, how many components of the columns must be loaded into the vector register. The C programs in Table 7 depicts the sequential program and the sequential program lifted over rows. The C programs in figure 8 depict lifting over columns as well as cache-blocked code.[2]

In order to mechanize and generalize mappings, the decision to use high-level mnemonics, e.g. OpenMP, was made. Identifying the simplest, most general and reliable ones was our challenge here.

2.3.1 Generalizing, Lifting and Adapting to Hardware: The Theory

The previous section discussed partitioning one shape component of an array into two to associate a **chunk** of sequential sections to a number of processors or registers. The rows of **A** were mapped to **np** processors and the columns of **B** and **C** were mapped to vector registers of length **rsize**. Consequently, the C code can be viewed as an MoA indexed expression. What differentiates the two expressions is that the implementation now knows how to index the hardware used, thus providing cost functions.

What we've done so far is to lift two shape components: first over the rows of **A** and **C** then over the columns of **B** and **C**.

[2] C interfaces with VHDL and Verilog. Thus, they are also possible as targets, making MoA an ideal interface to hardware design, costs, and verification.

$$\mathbf{A} \bullet \mathbf{B} = C \rightarrow \mathbf{A}' \bullet \mathbf{B}' = \mathbf{C}'$$

such that

$$\rho(\mathbf{A} \bullet \mathbf{B} = C) \rightarrow \rho(\mathbf{A}' \bullet B' = \mathbf{C}')$$

$$\rho\mathbf{A} \rightarrow \rho\mathbf{A}^{'} \quad \rho\mathbf{B} \rightarrow \rho\mathbf{B}^{'} \quad \rho\mathbf{C} \rightarrow \rho\mathbf{C}^{'}$$

$$< m,\ n > \rightarrow < np,\ m/np,\ n > \quad < n,\ p > \rightarrow < n,\ p/rsize,\ rsize >$$

$$< m,\ p > \rightarrow < np,\ m/np,\ p/rsize,\ rsize >$$

Thus, at any instant of time we know where every index is. Notice that **n** was not partitioned. Thus, $< m,\ n,\ p >$, defines the loop bounds that define the ONF that became a generic program written in C. Lifting and compressing dimensions is done easily with γ' and γ. Let np denote the number of partitions, e.g. processors, the divisor that gives the first coordinate, Let s denote the shape component to be lifted to $< \vec{s}_0 \vec{s}_1 >$ where $\vec{s}_0 = np$ and $\vec{s}_1 = s/np$ assuming $s/np \bmod np \equiv 0$

3 The target systems and source codes: algorithm variants

Having mathematically designed the high-level algorithm variants, we now had to take the MoA ONF to C programs, add OpenMP pragmas and use compiler flags to communicate with the compiler. Identifying pragmas and C flags that consistently worked across compilers and machines was a challenge. Although there were numerous flags, and OpenMP directives, most turned out to be merely "suggestions", hence, not useful as a general methodology.

Experiments were run using a single node of two machines provided by Sony Brook University's Ookami computer center: 1. Fujitsu's A64FX that we call simply "Ookami" and 2. Intel's Skylake that we call "Intel". We used three compilers: gcc on both machines, and fcc and icc on the A64FX and Skylake respectively. Table 1 describes the machine attributes. The A64FX was targeted due to it's success as the highest performing supercomputer. The Intel Skylake was targeted to build upon successes with previous experiments using cache blocked code[8, 9]. Ookami had multiple compilers. We initially chose the fastest based on initial tests of sequential code, e.g. fcc. We later chose gcc to compare results on two machines using the same compiler. The icc compiler was Intel's compiler and was chosen to compare with Fujitsu's fcc. A natural hypothesis was that the native (fcc, icc) compilers were written by the architecture's experts and should outperform gcc.

The A64FX processor was developed by Riken and Fujitsu for the Japanese path to exascale computing. Called Fugaku, it is currently the fastest computer in the world, and the first such computer outside of Japan. By focusing on crucial architectural details, the ARM-based, multi-core, 512-bit SIMD-vector processor with ultrahigh-bandwidth memory promises to retain familiar and successful programming models while achieving very high performance for a wide range of applications. It supports a wide range of data types and enables both HPC and big data applications. Skylake, is Intel's codename for its sixth generation Core microprocessor family. Skylake is a microarchitecture redesign using 14 nm manufacturing process technology .

All compilers used have numerous flags to choose from. We chose the flags that we believed would perform similar optimizations with the idea of scaling our designs with similar performance attributes automatically. We started with the C programs mentioned above. From them the only OpenMP pragma that was useful was the "parallel for". Although each loop could have been parallelized, multiple "parallel for" loops had no effect on performance. We also discovered that other optimizations, ideally done by the compiler had to be done by hand, e.g. eliminating common sub expressions, pre-fetching and loop unrolling, parameters that need further investigation to be fully automated. The programs listed in Figures 9, 10, 11, 12 illustrate this. We also used a blocked design that showed high performance in previous experiments on an Intel machine[8, 9]. This blocked design was based on cache-blocking and performance was consistent on the Intel Skylake but not on the Fujitsu A64FX.

4 The timing experiments

Building upon ideas of shapes and optimizing memory processor layouts[10], blocks in experiments fit the L1 Cache (64KiB/core), or 48 by 48 doubles. With two levels of memory we were able to predict

Table 1: Machines Used

Characteristics	Intel Skylake	Fujitsu A64FX
Architecture	x86_64	Arm.2−A+SVE
CPUs (cores)	36	48
Threads/CPU	1	1
CPU MHz	1764.320MHz	2000MHz
L1 cache	32KB	64KB

speedups using vector registers and multiple processors. We targeted one node with 4 Core Memory Groups(CMGs), each with 12 threads per core (48 threads in total).

One aspect of experiments was to use flags that were basically the same across platforms. Another set of experiments could identify what flags to extend given the plethora of flags available on each compiler. We use, typically, fast or O3 to vectorize, a flag to identify which architecutre, a flag to identify OpenMP. The only ones we used were prefetch and unroll. There would be a set of experiments to identify what the best prefetch size should be and the amount of loops to unroll. There are obviously related to the data and instruction cache sizes.

All of the experiments described in this paper target a single Node, which on the Ookami architecture has four Core Memory Groups (CMGs), with up to 48 CPUs per Node, 12 per CMG. Some of our current timing experiments target a mono-thread CPU and measure the efficiency of code vectorization. Others use multi-threaded execution to measure the efficiency of openMP parallelization combined with vectorization. Experiments were run using automated scripts to compile and run for varying sizes of input square matrices. The executables were then submitted via a batch mechanism (slurm) to obtain csv files used in our plots and analysis.

Plots in Figures 16b,16a illustrate the performance on both machines using the gcc and icc compilers on the cache blocked code. Figures 18b and 18a are closeups of these experiments. Futher experiments were not run due to licensing issues. The only experiments that were consistent with results obtained previously [8, 9], were the ones run on Intel's Skylake. The previous experiments were not run on large matrices. This time, we noticed, that as matrix sizes increased performance degraded as can be seen in Figure 16a and 18a. The plots in Figures 17a and 17b denote the experiments with rows sent to processors, i.e. dimension lifted over the first loop. Figure 13 depicts parallel performance using the fcc compiler on 2-48 processors.

5 Exploration of pre-execution decisions

We ran several thousands of similar experiments on the two available multicore architectures (that we call "Fujitsu" and "Intel" for short) measuring correct [3] execution speeds for increasing matrix sizes, aiming for the best possible Flop rates and parallel efficiency. That is a common parallel computing practice.

Despite testing a single general algorithm (matrix multiplication) our experiments are part of a methodology that is meant to be generalizable to any MoA-derived matrix code. Morevoer, we have used up to 3 variants of the algorithm, variants whose C source code can easily be generated from different but equivalent array expressions.

We validate a systematic approach to produce the best possible set of *static parameters* for our general algorithm. We explored and "solved" for the best choice of the following static parameters: **algorithm variant (choices of source code loops), compiler flags, compiler directives, and OpenMP pragmas**. Apart from the execution parameters, we also explored the usual "size" parameters: input matrix size and number of cores. Our current experiments only cover vectorization and multi-thread (multi-core) parallelization but will extend to multinode, GPU and other architecture levels, using static parameters that are specific to MPI, CUDA etc, and of course size parameters like the number of nodes or kernels etc.

The preliminary conclusions about execution times for each choice of static parameters are:

- Execution times approximately follow a cubic-time curve as algorithm complexity predicts
- Irregularities in the shape of the timing curves are:
  - localized around specific matrix sizes

[3] All matrix-multiplication results were checked to be numerically correct, which may not be the case for certain OpenMP runs that raise race condition warnings.

– dependent on the choice of algorithm variant
– appear on both architectures, but
– more frequent and larger on the Intel experiments that are faster
– appear related to cache sizes and blocking sizes

Our execution time prediction is currently limited to interpolation with small error percentages, but fails to predict the irregularities (e.g. specific treatment of cache effects in c.f. work of Pawlowski et al. [11]).

We systematically ran such experiments for every combination of the execution parameters. Then we summarized every experiment (curve) into its average and maximum execution speed (Flop count from the matrix multiplication algorithm divided by runtime) in GFlops/s. Then searched the space of static parameters separately on each architecture [4]. The preliminary conclusions about the effect of static parameters on average/maximum speed are:

- speed is monotonic along the dimension of almost every static parameter: one compiler is 95% of the time faster than the other one, vectorization is almost always faster than non-vectorized run, etc.
- it was possible to use a naive steepest descent along axes (static parameters) to find the fastest choice
- once the fastest choice of static parameters was found, it was put to parallel speedup analysis which measured scalability, comm-sync overheads etc.

On the Fujitsu A64FX node we ran 19 experiments, each one comprising from 9 to 100 runs of different matrix sizes. One experiment covers a 4-dimensional value of static parameters and the experiment set exhausts our search space. Because our experiments are exhaustive, we could select a favorable order on the four static parameters. An open problem remains to solve before converting our methodology into a (pre-)compilation method: how to select a good order on the static parameters to test, while minimizing the number of experiments. For each static parameter in sequence we proceeded as follows: (a) partition experiments by the values of that parameter and for each value obtain the average (or averages) execution speed for that subset of all experiments; (b) then observe that one of the values of that parameter has systematically equal or better performance than all others (this defines what we called "good order" and could always be found) and (c) dismiss all experiments with the other values of the given parameter (d) then repeat for the next parameter on the remaining subset of experiments etc (figure 3).

This procedure amounts to a steepest descent for execution time in the space of static parameters, with the simplification of moving along a "Manhattan topology". Once the best vector of static parameters has been selected, we measured its parallel performance: runtime vs number of cores, parallel acceleration (monocore runtime / runtime) and parallel efficiency (acceleration / number of cores). Those values are displayed in the 3 graphs of figure 3. The lower graph of the last graphic is the remainder for 100of efficiency, that is the percent of acceleration that is lost to communication and synchronization.

For the Fujitsu architecture, the absolute speeds measured ranged from 0.3 to 5.6 GFlops/s/core which is still far from the published peak rates above from 50 GFlops/s/core. We have found no simple explanation for this if only that the peak rates are either theoretical (with no guarantee of correct numerical results in the presence of dependencies) or published from hand-written assembly code values. Nevertheless the (relative) parallel accelerations are impressive and (relatively) better than those observed on the Intel architecture: 95% effiency when using 12 cores and still 80% efficiency when using 48 cores, twice as many as the Intel architecture.

For the Intel architecture, the absolute speeds measured ranged from 0.3 to 7.8 GFlops/s/core and often 2 to 3 times faster than similar experiments on the Fujitsu architecture. But the reader is reminded that cross-archictecture comparisons are very delicate.

Our parameter selection process was applied to our large (333) set of experiments on the Intel architecture. An ordering was chosen on the four static parameters selected and then successive tests allowed us to conclude that one value was systematically no worse than the other for that parameter. The final selection favored the native icc compiler, while gcc has beaten Fujitsu's native fcc compiler for our other experiments.

The (relative) Intel parallel accelerations are good but not as high as the ones for Fujitsu, moreover only 24 cores were available instead of 48: 75% efficiency when using 24 cores, and as the graph in figure 6 shows, there are unexplained irregularities in the parallel efficiency curve. Part of the reason for those could be the finer time scale at which they are measured, often 2-3 times faster than the Fujitsu

[4] comparing similar static parameters across architectures is to us an open problem because the compilers vary and the effects of parameters on different hardware appears unpredictable. In other words, we treat hardware as a black box.

experiments. But since all values come from averages of averages, we postulate that they exhibit some of the architecture's quantitative feature. Such irregularities are usually visible in a single experiment's timing curve as the array size varies.

The experiments and parameter analyses were designed as a proof of concept for an pre-compilation methodology that can automate the tuning and porting of parallel C codes that were themselves generated automatically from an array algebra expression.

The measures presented here also help compare the two architecture nodes that were tested: better average Flops rate for the Intel node but more nodes and more scalability for the Fujitsu node. But that is not our main concern. We have begun to demonstrate that a mixture of performance testing, and small-scale optimization can lead to mostly predictable, scalable and automatic execution of declarative array expressions.

6 Conclusions and future work

A goal of MoA research is to identify common algorithms, e.g. matrix multiplication (MM), Kronecker product (KP), and Hadamar product (HP), then mechanically produce scalable, high-performance parallel algorithms using high-level constructs, e.g. OpenMP, MPI, OpenACC and CUDA.

In this set of experiments we've shown how to use *dimension lifting* to break up 2 and 3 loops using OpenMP thus producing 2 or 3 variants of the same matrix multiplication algorithm. Regarding multicore execution, what we've discovered is that although OpenMP has numerous ways to provide parallelization, these are just suggestions (e.g. using Static or Collapse pragmas) may or may not improve performance. Consequently, using them is not scalable (at this time) except the only reliable pragma that has been found: **parallel for**. It produces reliable improvements across two different architectures, the A64FX and Intel Skylake. For example, OpenMP collapse caused race conditions producing incorrect results. This unreliable behaviour is similar to unstructured use of MPI where non-deterministic behaviour can be produced by hand-written programs.

In the dimension of scalability, we were also able to determine that more processing elements did not improve the performance for 5164 by 5164 matrices and reached a maximum at 24 threads on both machines. It is like that as the data size increased that more threads were beneficial. That was validated by one of our experiments where we tried over 20000 by 20000. Execution times and queue time limits kept us from running these large size experiments, thus limiting them to 5164 by 5164. The most relevant question here would be to determine the maximum matrix size where performance begins to degrade on A64FX or Skylake.

Finally, programming tools will be designed to automate the already systematic generation of C source code from MoA array expressions. Dimension lifting is just one of the rich set of transformations to be experimented with. Given a simple-minded definition of algorithm from MoA primitives (array constructors, transformers like transposes and dim-lifting, reductions like dot products) it is possible to explore a whole set of equivalent algorithm variants, each one entering the optimization process that we described above so as to produce the best possible execution time for the target architecture. For example an open question is to discover a sequence of MoA transformations from naive MM to a Stassen-like block-recursive algorithm, or prove that the algebra is not expressive enough.

The next steps in this research should determine whether our designs scale to multiple nodes using MPI, to explore their behaviour on very large multicore nodes (such as the 8 CPU Intel Cooper lake, 224 cores, 6To RAM "SMP" node available at CRIANN in Normandy) and finally design a GPU and then hybrid adaptation of our approach which is currently restricted to vectorization + multicore parallelization. Another open problem is: how close do we get from 'optimal' performance for a given architecture? This is unknown at this point because it involves first of all defining what optimal speed is, possibly an architecture- and source-code dependent maximal speed beyond which race conditions make the numerical results unsafe.

Acknowledgments

The Ookami computing center is supported by the US NSF grant #1927880. The authors gratefully acknowledge the generous support of the Ookami community. In particular, we thank Eva Siegemann, Tony Curtis, Ben Michalowicz, David Carlson for their many programming suggestions on Ookami's Slack Channel, and others who spent their time talking to us during office hours.

References

[1] Valiant, L.G.: Why BSP computers?(bulk-synchronous parallel computers). In: [1993] Proceedings Seventh International Parallel Processing Symposium, pp. 2–5 (1993). IEEE

[2] McColl, W.F., Tiskin, A.: Memory-efficient matrix multiplication in the BSP model. Algorithmica **24**, 287–297 (1999)

[3] Thomas, S., Mullin, L., Swirydowicz, K.: Improving the performance of DGEMM with MoA and cache-blocking. Technical report, National Renewb. Energy Lab.(NREL), Golden CO (USA) (2022)

[4] Hains, G., Mullin, L.M.R.: Parallel functional programming with arrays. The Computer Journal **36**(3), 238–245 (1993)

[5] Bisseling, R.H.: Parallel Scientific Computation: a Structured Approach Using BSP and MPI. Oxford University Press, Oxford (2004)

[6] Grout, I.A., Mullin, L.: Realization of the kronecker product in vhdl using multi-dimensional arrays. In: 2019 7th International Electrical Engineering Congress (iEECON), pp. 1–4 (2019)

[7] Grout, I.A., Mullin, L.M.: Realizing mathematics of arrays operations as custom architecture hardware-sofwate co-design solutions. Information 2022 **13**(11) (2022)

[8] Thomas, S., Mullin, L., Swirydowicz, K., Khan, R.: Threaded multi-core gemm with moa and cache-blocking: Preprint

[9] Thomas, S., Mullin, L., Świrydowicz, K.: Improving the performance of dgemm with moa and cache-blocking: Preprint. Proceedings of ARRAY '21, 20-26 PLDI ACM June 2021 (2022)

[10] Mullin, L.M.R.: From array algebra to energy effiiency on GPUs. Technical report, HLPP2023: Cluj-Napoca (RO) (2023). https://doi.org/10.48550/arXiv.2306.11148

[11] Pawłowski, F., Uçar, B., Yzelman, A.-J.: A multi-dimensional morton-ordered block storage for mode-oblivious tensor computations. Journal of Computational Science **33**, 34–44 (2019)

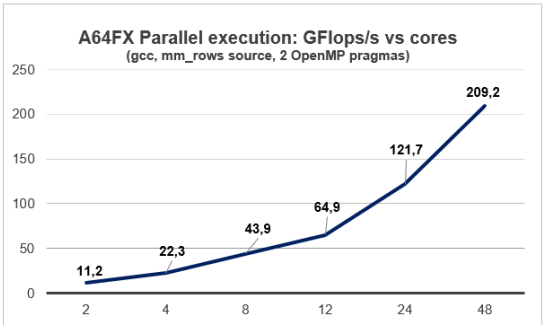
Choosing the static parameters : Fujitsu [compiler ; vectorized/not ; OMP pragmas ; algorithm variant]

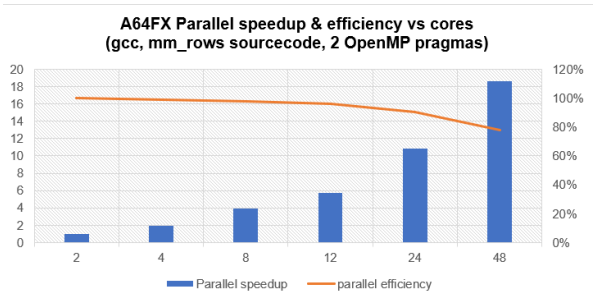
Comparison	vectorized	Source code	Compiler	OMP pragmas	Speedup
Fujitsu-cc vs gcc	Y/N	48x48 blocks	fcc/gcc	common-subexpressions removed by hand	3,2
Fujitsu-cc vs gcc	Y	48x48 blocks	fcc/gcc	1 OMP pragma	1,6
Fujitsu-cc vs gcc	Y	48x48 blocks	fcc/gcc	2 OMP pragmas	1,5
Fujitsu-cc vs gcc	Y	48x48 blocks	fcc/gcc	1 OMP pragma	1,2
Fujitsu-cc vs gcc	Y	48x48 blocks	fcc/gcc	2 OMP pragmas	1,1
Vectorized vs not vect.	Y/N	48x48 blocks	fcc/gcc	common-subexpressions removed by hand	3,2
2OMP vs 1OMP	Y	48x48 blocks	gcc	2 OMP pragmas / 1 OMP pragmas	1,1
2OMP vs 1OMP	Y	48x48 blocks	fcc	2 OMP pragmas / 1 OMP pragmas	1,0
2OMP vs 1OMP	Y	48x48 blocks	gcc	2 OMP pragmas / 1 OMP pragmas	1,0
2OMP vs 1OMP	Y	48x48 blocks	fcc	2 OMP pragmas / 1 OMP pragmas	1,0
mm_rows vs 48x48 blocks	Y	mm_rows/48^2blocks	gcc	2 OMP pragmas	14,6
mm_rows vs 48x48 blocks	?/Y	mm_rows/48^2blocks	gcc	2 OMP pragmas	16,8
mm_rows vs 48x48 blocks	Y	mm_rows/48^2blocks	gcc	2 OMP pragmas	14,6
mm_rows vs 48x48 blocks	Y	mm_rows/48^2blocks	gcc	2 OMP pragmas	5,2

Best static parameters (Fujitsu):

vectorized	Source code	Compiler	OMP pragmas
Y	mm_rows	gcc	2 OMP pragmas

Fig. 2: Static parameters selection: Fujitsu architecture





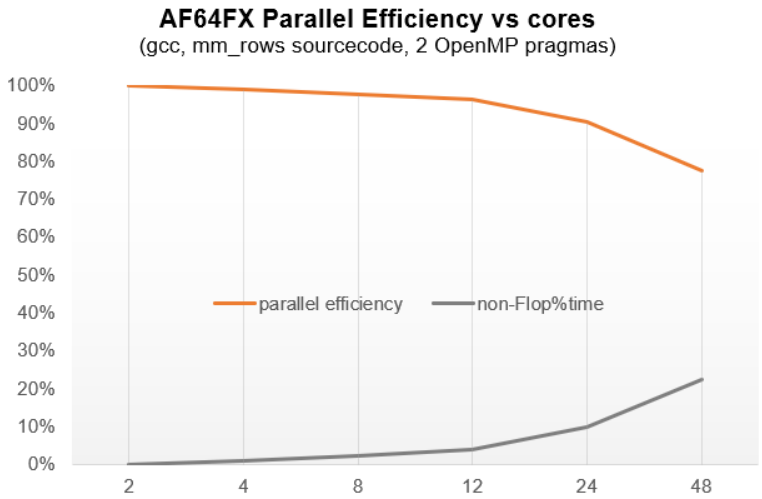


Fig. 3: Parallel execution time, speedup and efficiency (speedup/cores): Fujitsu architecture

Choosing static parameters (Intel) : [compiler ; common subexpression elim./not ; algorithm variant, ; prefetching/not]

Comparison	vectorized	Source code	Compiler	com-subex remv by hand	Prefetching	Threads	Avg Gflops/s	Avg GFlops/s/core	statistics
Compiler?	Y	24B/48B/rows	icc	Y/N	Y/N	2-24	61,10	5,21	Avg +/- sdev
							+/- 33,01	+/- 1,44	
			gcc				4,69	0,43	
							+/- 2,23	+/- 0,27	
Comparison	**vectorized**	**Source code**	**Compiler**	**com-subex remv by hand**	**Prefetching**	**Threads**	**Avg Gflops/s**	**Avg GFlops/s/core**	
com-subex rem ?	Y	24B/48B/rows	icc	Y	Y/N	2-24	62,27	5,21	Avg +/- sdev
							+/- 32,35	+/- 1,44	
				N			5,92	5,92	
							+/- 1,23	+/- 1,23	
Comparison	**vectorized**	**Source code**	**Compiler**	**com-subex remv by hand**	**Prefetching**	**Threads**	**Avg Gflops/s**	**Avg GFlops/s/core**	
Source code ?	Y	24x24 blocks	icc	Y	Y/N	2-24	67,50	5,61	Avg +/- sdev
							+/- 31,08	+/- 0,64	
		48x48 blocks					76,93	6,51	
							+/- 33,69	+/- 0,92	
		mm_rows					42,39	3,48	
							+/- 20,05	+/- 0,32	
Comparison	**vectorized**	**Source code**	**Compiler**	**com-subex remv by hand**	**Prefetching**	**Threads**	**Avg Gflops/s**	**Avg GFlops/s/core**	
Prefetching?	Y	48x48 blocks	icc	Y	Y	2-24	78,55	6,48	Avg +/- sdev
							+/- 32,50	+/- 0,89	
					N		75,38	6,54	
							+/- 34,69	+/- 0,95	

Best static parameters : Intel

Acceleration	vectorized	Source code	Compiler	com-subex remv by hand	Prefetching
Best configuration	Y	48x48 blocks	icc	Y	Y

Fig. 4: Static parameters selection: Intel architecture

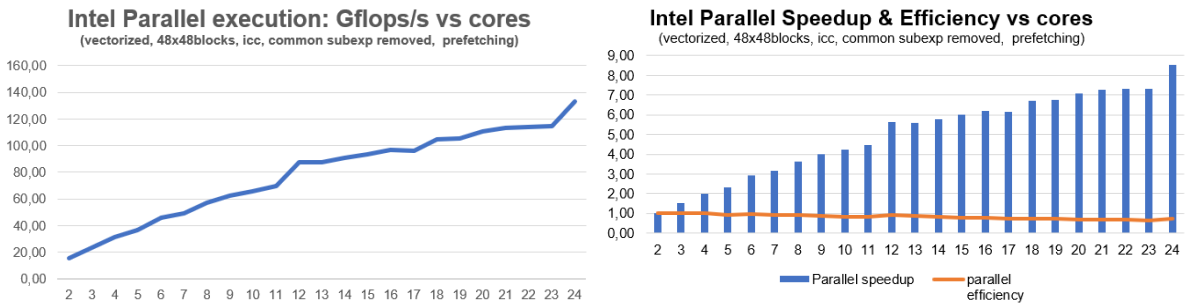


Fig. 5: Parallel execution time and speedup vs cores: Intel architecture

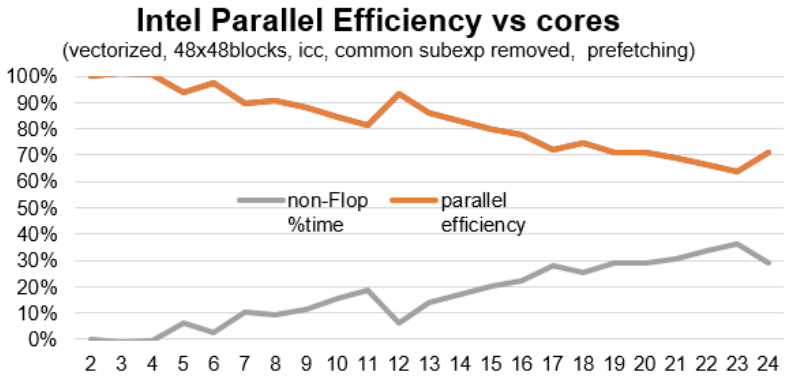


Fig. 6: Parallel efficiency (speedup/cores): Intel architecture

```
 #include <stdio.h>
 #include <sys/time.h>

  void ip(double *C, double *A, double *B,
      int sizel, int sizer, int sizeres, int np, int shr0)
  {
    int i,j,sigma;
   for (i=0;i<sizel;i++)
     {for (sigma=0; sigma<shr0; sigma++)
       {for (j=0;j<sizer;j++)
         {
           C[j+i*sizer]=C[j+i*sizer]+A[(i*shr0)+sigma]*B[(sigma * sizer)+j];
         }
       }}}

#include <stdio.h>
#include <sys/time.h>
#include <omp.h>
void mm_rows_pragmaomp (double *C, double *A, double *B,int m, int n, int p,
 int np)
 {
   int i,j,sigma;
   {
          for (i=0;i<m;i++)
        {
          {
             for (sigma=0; sigma<n; sigma++)
           {
             for (j=0;j<p;j++)
          {
               C[j+i*p]=C[j+i*p]+A[(i*n)+sigma]*B[(sigma * p)+j];
          }
           }}}}}
```

Fig. 7: Original MoA MM without then with dimension lifted over rows

```
#include <stdio.h>
#include <sys/time.h>

 void ip_cols(double *C, double *A, double *B,
     int sizel, int sizer, int sizeres, int np, int shr0,int rsize)
 {
   int i,j,jp,kp,sigma;
  for (i=0;i<sizel;i++)
    {for (sigma=0; sigma<shr0; sigma++)
    { for (jp=0;jp<(sizer/rsize);jp++)
        {for (kp=0;kp<rsize;kp++)
        {
          C[((jp*rsize)+kp)+i*sizer]=C[((jp*rsize)+kp)+i*sizer]+A
           [(i*shr0)+sigma]*B[(sigma * sizer)+((jp*rsize)+kp)];
        }
        }}}}

#include <stdio.h>
#include <sys/time.h>
#include <omp.h>
 void mm_block(double *C, double *A, double *B, int m, int n, int p, int mb,
  int nb, int pb)
 {
     int i,k,l,ip,jp,kp,sigmap;
     int id;
     {
     for (ip=0;ip<(m/mb);ip++)
       {for (sigmap=0;sigmap<(n/nb);sigmap++)
       for (jp=0;jp<(p/pb);jp++)
            {for (k=0;k<mb;k++)
         {
             {for (l=0;l<nb;l++)
             {for (kp=0;kp<pb;kp++)
                  {
               #pragma prefetch
               #pragma ivdep
               #pragma unroll(24
                     C[((jp*pb)+kp)+(ip+(m/mb)*k)*p]=C
                      [((jp*pb)+kp)+(ip+(m/mb)*k)*p]+A
                      [((ip+((m/mb)*k))*n)+((sigmap+((n/nb)*l)))]*B
                      [((sigmap+((n/nb)*l)) *p)+((jp*pb)+kp)];
                  }
             }
         }
       }
       }}
     }
 }
```

Fig. 8: MoA MM dimension lifted over columns, without & with blocking

```
#include <stdio.h>
#include <sys/time.h>

 void mm_rows_pragmaomp(double * restrict C,
              double * restrict A,
              double * restrict B,
            int m, int n, int p, int np)
{
    int i,j,k,sigma;
//#pragme omp parallel for shred (A,B,C)
     for (unsigned k=0;k<np;k++) {
       const unsigned int t1 = (m/np)*k;
         for (unsigned ip=0;ip<(m/np);ip++){
             const unsigned int t2 = ip+t1;
             const unsigned int t3 = t2*p;
             const unsigned int t6 = t2*n;
             for (unsigned sigma=0; sigma<n; sigma++){
                 const unsigned int t4 = t6+sigma;
                 const unsigned int t5 = sigma * p;
//#pragma prefetch
//#pragma ivdep
//#pragma unroll (8) //cacheline(skylake=64) = 64 size/double
//#pragma unroll (32) //cacheline(a64fx=256) = 256/double
                 for (unsigned j=0;j<p;j++) {
                   const unsigned int a_index = t4;
                   const unsigned int b_index = t5+j;
                   const unsigned int c_index = j+t3;
          C[c_index]=C[c_index]+A[a_index]*B[b_index];
                 }}}}}
```

Fig. 9: mm_rows with common sub expressions removed

```
#include <stdio.h>
#include <sys/time.h>

 void mm_rows_pragmaomp(double * restrict C,
              double * restrict A,
              double * restrict B,
            int m, int n, int p, int np)
{
#pragma omp parallel for shared (A,B,C)
     for (unsigned k=0;k<np;k++) {
       const unsigned int t1 = (m/np)*k;
         for (unsigned ip=0;ip<(m/np);ip++){
             const unsigned int t2 = ip+t1;
             const unsigned int t3 = t2*p;
             const unsigned int t6 = t2*n;
             for (unsigned sigma=0; sigma<n; sigma++){
                 const unsigned int t4 = t6+sigma;
                 const unsigned int t5 = sigma * p;
//#pragma prefetch
//#pragma ivdep
//#pragma unroll (8) //cacheline(skylake=64) = 64 size/double
//#pragma unroll (32) //cacheline(a64fx=256) = 256/double
                 for (unsigned j=0;j<p;j++) {
                   const unsigned int a_index = t4;
                   const unsigned int b_index = t5+j;
                   const unsigned int c_index = j+t3;
          C[c_index]=C[c_index]+A[a_index]*B[b_index];
                 }}}}}
```

Fig. 10: mm_rows, common sub expr. removed with OMP

```
 void mm_block(double * restrict C,
          const double * restrict A,
          const double * restrict B,
          int m, int n, int p, int mb, int nb, int pb)
 {
 // #pragma omp parallel for shared(A, B, C)
     for (unsigned int ip = 0; ip < (m / mb); ip++) {

         for (unsigned int sigmap = 0; sigmap < (n / nb); sigmap++) {

             for (unsigned int jp = 0; jp < (p / pb); jp++) {
                 const unsigned int t0 = jp * pb;

                 for (unsigned int k = 0; k < mb; k++) {
                     const unsigned int t1 = ip + ((m / mb) * k);
                     const unsigned int t3 = t1 * p;
                     const unsigned int t4 = t1 * n;;

                     for (unsigned int l = 0; l < nb; l++) {
                         const unsigned int t2 = sigmap + ((n / nb) * l);
                         const unsigned int t5 = t2 * p;
                         //#pragma prefetch
                         //#pragma ivdep
                         //#pragma unroll (8) //cacheline(skylake=64) =
                          64/double=8
                         //#pragma unroll (32) //cacheline(a64fx=256) =
                          256/double=8
                         for (unsigned int kp = 0; kp < pb; kp++) {
                             const unsigned int a_index = t4 + t2;
                             const unsigned int b_index = t5 + t0 + kp;
                             const unsigned int c_index = t0 + kp + t3;

                             C[c_index] += A[a_index] * B[b_index];
                         }
                     `
```

Fig. 11: mm_block common sub expr. removed

```
void mm_block(double * restrict C,
         const double * restrict A,
         const double * restrict B,
         int m, int n, int p, int mb, int nb, int pb)
{
 #pragma omp parallel for shared(A, B, C)
    for (unsigned int ip = 0; ip < (m / mb); ip++) {

        for (unsigned int sigmap = 0; sigmap < (n / nb); sigmap++) {

            for (unsigned int jp = 0; jp < (p / pb); jp++) {
                const unsigned int t0 = jp * pb;

                for (unsigned int k = 0; k < mb; k++) {
                    const unsigned int t1 = ip + ((m / mb) * k);
                    const unsigned int t3 = t1 * p;
                    const unsigned int t4 = t1 * n;;

                    for (unsigned int l = 0; l < nb; l++) {
                        const unsigned int t2 = sigmap + ((n / nb) * l);
                        const unsigned int t5 = t2 * p;
                        //#pragma prefetch
                        //#pragma ivdep
                        //#pragma unroll (8) //cacheline(skylake=64) =
                         64/double=8
                        //#pragma unroll (32) //cacheline(a64fx=256) =
                         256/double=8
                        for (unsigned int kp = 0; kp < pb; kp++) {
                            const unsigned int a_index = t4 + t2;
                            const unsigned int b_index = t5 + t0 + kp;
                            const unsigned int c_index = t0 + kp + t3;

                            C[c_index] += A[a_index] * B[b_index];
                        }
                    `
```

Fig. 12: mm_block common sub expr. removed, OMP

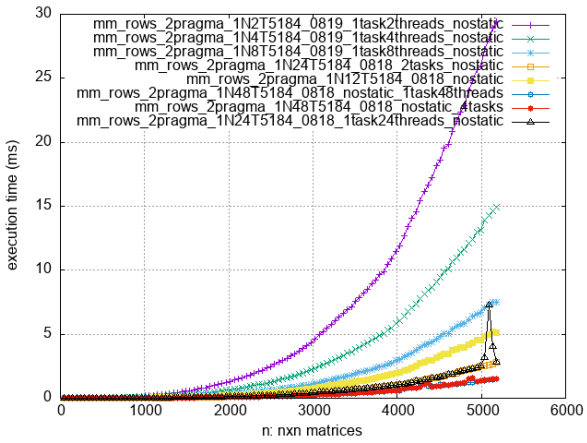


Fig. 13: Scalable OpenMP for 2-48 cores

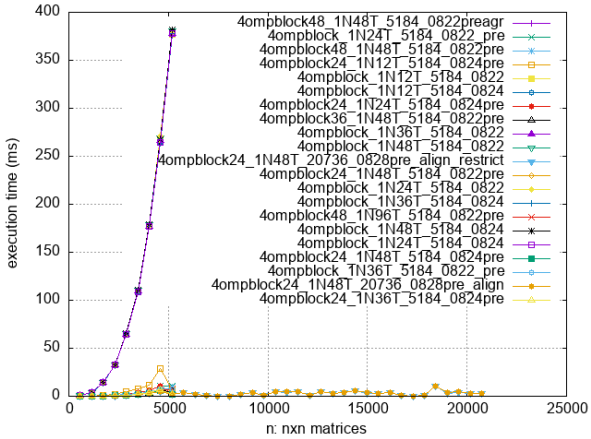


Fig. 14: Collapse(5) versus Static: Static executes sequentially

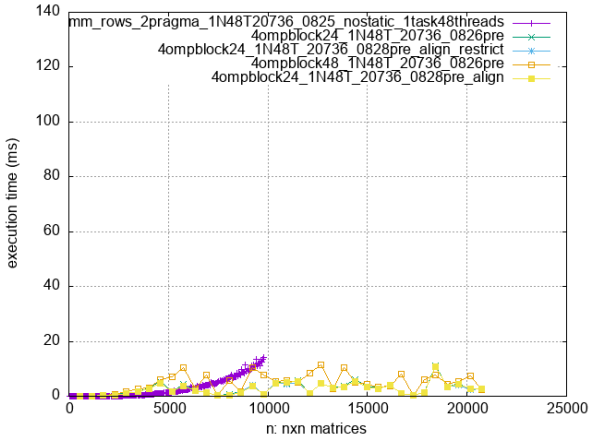
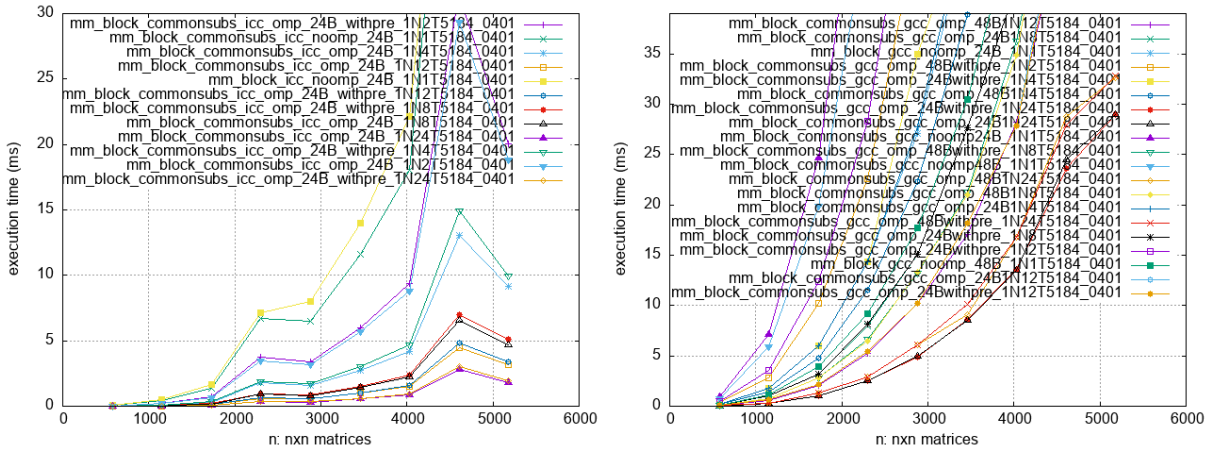


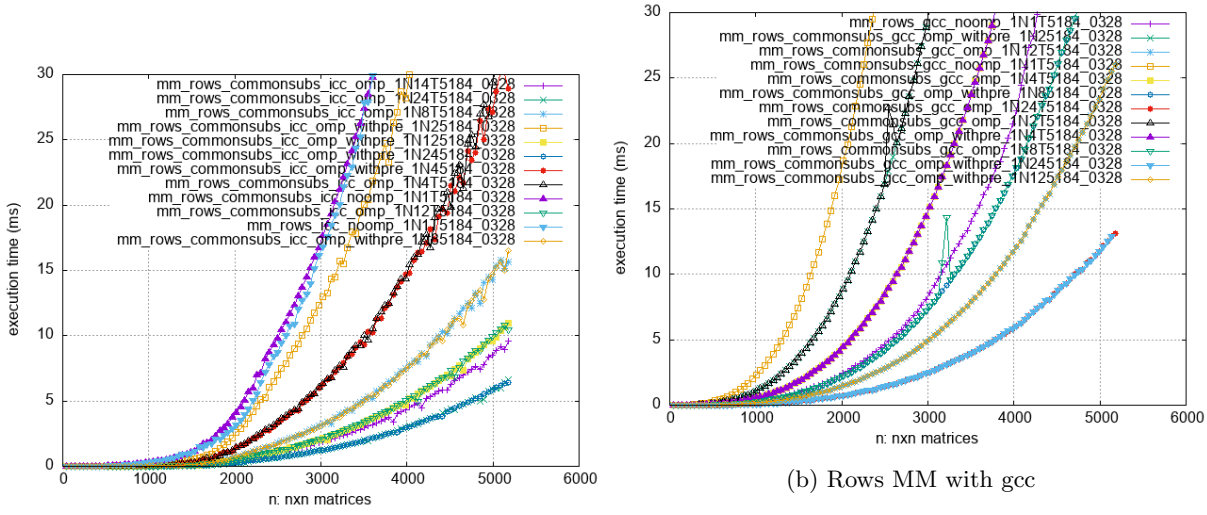
Fig. 15: Collase(5) versus Two Pragmas using Row Partitioning: Close Up



(a) Blocked MM with icc compiler

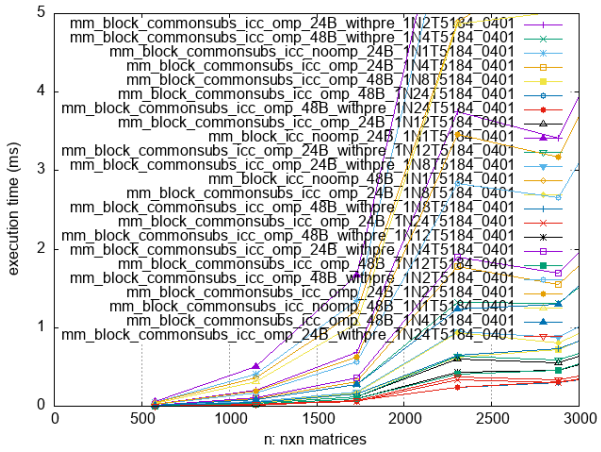
(b) Blocked MM with gcc

Fig. 16: Blocked Performance

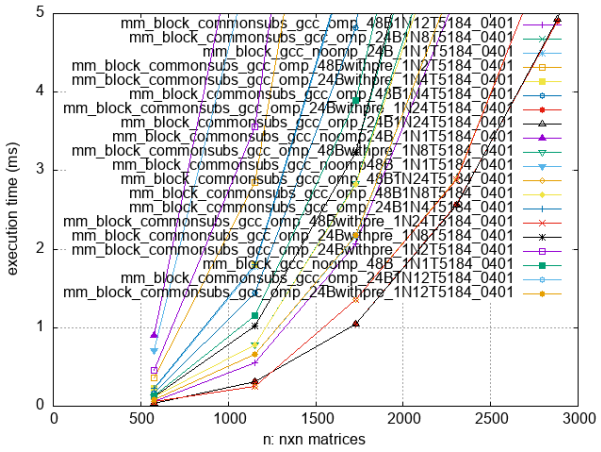


(a) Rows MM with icc

(b) Rows MM with gcc

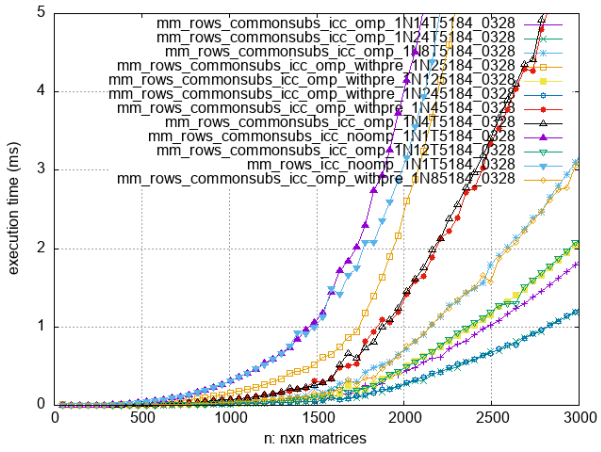


(a) Blocked MM with icc compiler:closeup

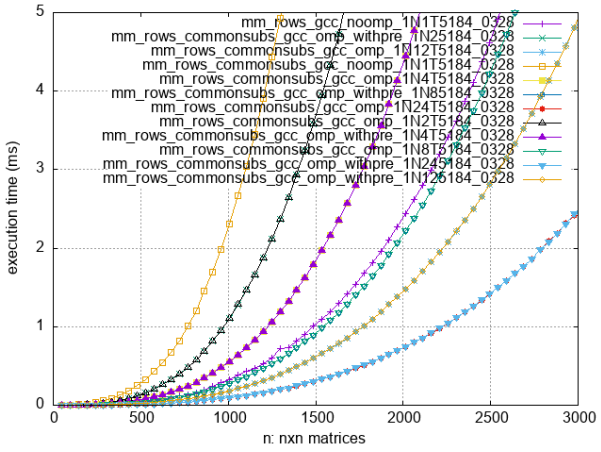


(b) Blocked MM with gcc:closeup

Fig. 18: Blocked Performance:closeup



(a) Rows MM with icc:closeup



(b) Rows MM with gcc:closeup

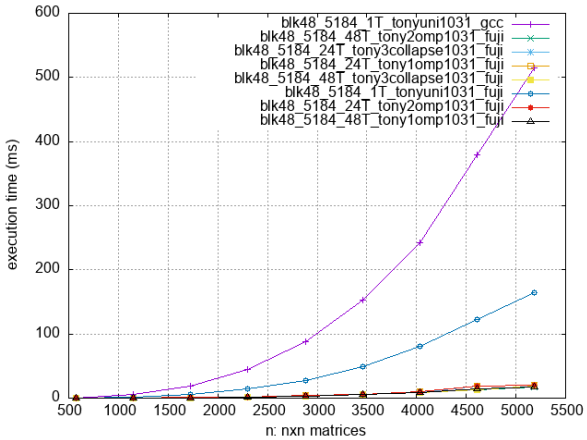


Fig. 20: Optimized MoA Blocked on 1, 24, and 48 processors.

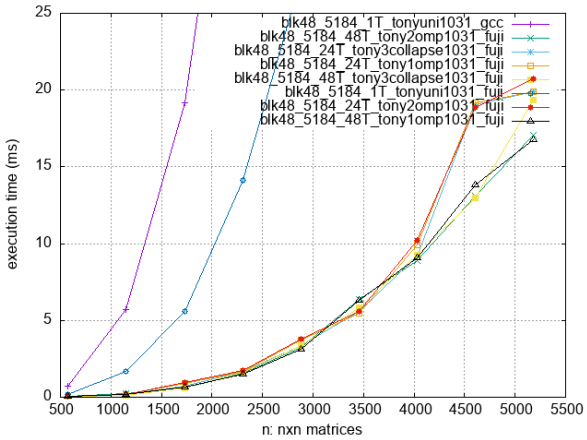


Fig. 21: Optimized MoA Blocked on 1, 24, and 48 processors: Close Up