Automatic Generation of Tiled and Parallel Linear Algebra Routines

A partitioning framework for the BTO Compiler

Geoffrey Belter[1], Jeremy G. Siek[1], Ian Karlin[2], and E. R. Jessup[2]

[1] Department of Electrical, Computer, and Energy Engineering, University of Colorado
[2] Department of Computer Science, University of Colorado

Abstract. Exploiting parallelism in modern hardware is necessary to achieve high performance in linear algebra routines. Unfortunately, modern architectures are complex so many optimization choices must be considered to find the combination that delivers the best performance. Exploring optimizations by hand is costly and time consuming. Auto-tuning systems offer a method for quickly generating and evaluating optimization choices. In this paper we describe a data-parallel extension to our auto-tuning system, Build to Order BLAS. We introduce an abstraction for partitioning matrices and vectors and we introduce an algorithm to partitioning linear algebra operations. We generate code for shared-memory machine using Pthreads. Results from the prototype show that our auto-tuning approach is competitive with existing state-of-the-art parallel libraries. We achieve speedups of up to 2.7 times faster than MKL and speedups up to 6 times faster than our best-optimized serial code on an Intel Core2Quad.

1 Introduction

The trend in architecture design is an increase in the number of processing units and the complexity of the memory system. Fortunately, linear algebra code, such as the Basic Linear Algebra Subprograms (BLAS) [1], can be adapted to both of these changes by taking advantage of well understood optimizations, specifically tiling for cache locality and data parallelism. In each case, the linear algebra operation must be partitioned into separate sub-operations. For example the operation $y \leftarrow Ax$ can be partitioned such that a tile of A and the vector x can be used to compute a tile of y. This partitioning strategy improves temporal locality use by allowing the tile of y to remain in cache. Alternatively, each tile of y can be computed in parallel. In this paper we present a general purpose partitioner for linear algebra operations that enables the generation of cache tiled and data parallel code.

Another trend is that changes in computer architecture are occurring with greater frequency, reducing the time to adapt linear algebra codes to new architectures. The maintenance of linear algebra codes is costly and time consuming, making methods such as automatic code generation and auto-tuning more valuable. The Build to Order compiler (BTO) is a tool we are developing to automatically generate high performance linear algebra routines from a high level specification. BTO takes a mathematical description of a sequence of matrix and vector operations and explores optimization alternatives. The BTO selects the best performing alternative for the target platform and

then generates C code. The focus of our tool is memory bound (Level 1 and Level 2) operations with the primary optimization being loop fusion.

In this paper we present the new partitioning framework for the BTO compiler. The BTO uses the partitioning framework to automatically generate data parallel and cache-blocked code. Although the framework can generate partitions for cache tiling, this paper focuses primarily on data parallelism. We present preliminary data parallel results showing we can achieve speedups over BTO's best serial code of up to 6 times faster and over the Intel MKL library of up to 2.7 times faster on and Intel Core2Quad. This paper begins by examining related work. Section 3 explains the background of the BTO system and then describes new partitioning framework. Section 4 shows preliminary results and, finally, we conclude with plans for future work.

2 Related Work

This section examines research regarding the creation and maintenance of portable high performance parallel code for shared memory multi-core systems. In particular, we focus on research on data partitioning for the purposes of exploiting parallelism.

Tools Aiding in Manual Partitioning Both the Matrix Template Library [10] [6] and Hierarchically tiled arrays [3] offer features that let programmers nest array abstractions. These nestings can be used to create partitions in matrices and vectors. The main drawback with these tools is the lack of automation; if a programmer wants to try several different ways to partition an operation to find the best, he must do so manually.

Tools that Partially Automate Partitioning The FLAME tool takes programs in a high-level syntax and automatically generates parallel code [7]. FLAME identifies independent iterations of loops which can be performed in parallel. Currently FLAME does not identify task level parallelism in a set of operations. However, the programmer using FLAME is responsible for generating each algorithm they wish to evaluate.

The Falcon tool compiles MATLAB into Fortran code [5] and uses domain specific knowledge to extract parallelism. For example, using standard BLAS routines, *gemm* can be performed in parallel as multiple *gemv* operations. This approach is guided by the programmer, who must select the portion of an input program to optimize and select the optimizations to perform. If the programmer wishes to evaluate many implementations, this process must be repeated by hand each time.

Fully Automated Partitioning Tools ATLAS is an auto-tuning tool that generates high performance, parallel linear algebra routines [11]. ATLAS dynamically selects one-, two-, or three-dimensional partitions based on the input matrix sizes. The results are competitive with hand-tuned implementations on a range of systems. However, ATLAS uses a single matrix repacking strategy which can limit performance in the presence of CPUs with short vectors units.

PLUTO is a source-to-source translator that uses the polyhedral model to analyze and restructure loops, including partitioning to generate parallel code [4]. The polyhedral model has proven successful, however one drawback is that it does not handle sparse matrices.

SPIRAL is a code generation system for the digital signal processing (DSP) domain [9]. SPIRAL uses a custom-built, high-level language specific to DSP. The system generates OpenMP code for shared memory systems and MPI for distributed systems.

3 Partitioning in the BTO Framework

Section 3.1 provides a brief review of the BTO compiler followed in Section 3.2 by a description of the BTO type system and how that enables data partitioning. The main contribution of the paper follows in sections 3.3 through 3.4. Section 3.3 describes the rules that govern partitioning data and operations in BTO. To ensure the generation of correct programs, BTO propagates partitioning decisions through the rest of the program, which is described in Section 3.4. Finally, in Section 3.6 we present a brief discussion of how the BTO internal representation is converted into C code.

3.1 The BTO Framework

BTO is a linear algebra optimization tool that automatically tunes a sequence of operations for a target platform. BTO takes as input a mathematical description of matrix and vector operations in a form similar to a subset of MATLAB. The programmer specifies the input and output variables to be either scalars, vectors, or matrices. An example input specification for a set of matrix-vector operations is shown in Figure 1 on the left.

A program is represented in BTO as a dataflow graph. In the dataflow graph, a node represents an input, output, or an operation. Edges represent the flow of data. Figure 1 shows an example program, consisting of two matrix vector products, and the corresponding dataflow graph. From this high-level description, the BTO framework decides how to implement the linear algebra operations by refining the data-flow graph into a more specific graph where matrix and vector operations are decomposed into abstract iterations over scalar operations. The abstract iterations are then mapped to sequential loops, threads, cache tiles, and vector units depending on the computer architecture. The refinement of the dataflow graph is driven by the BTO type system (discussed in detail in Section 3.2).

After all the program is refined into abstract iterations over scalars, the BTO framework enters an optimization phase. Currently the optimization phase enumerates combinations of loop fusion. The number of combinations is limited by a single heuristic: BTO considers a fusion profitable when two loops access a common piece of data. This optimization phase produces typically between one and 1000 versions of the program. The versions are fed through an analytic model based primarily on memory structure and bandwidth. The model is designed to quickly order the versions, identifying large differences in performance. The set of versions identified as best performing by the model are then empirically tested. The empirical testing identifies the overall best performing implementation. This portion of the tool is not discussed further here; see [2] for more details.

```
MATVEC2
in:
     x : vector, A : matrix
     z : vector
out:
     y : vector, w: vector
{
     y = A*x
     w = A*z
}
```

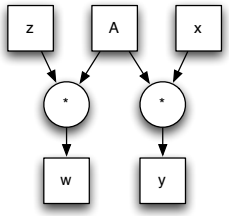


Fig. 1. Example BTO input of two matrix-vector operations on the left and corresponding dataflow graph on the right.

3.2 Type System

BTO assigns a type to each node in the dataflow graph. The type assigned to each node controls data layout in memory and is used to determine how to traverse the data to carry out a linear algebra operation. BTO also uses types to express multi-dimensional data partitions. The design of the type system is such that adding partitions does not require any changes to the BTO compiler with respect to how high-level dataflow graphs are refined into iterations over scalars. To understand how this is possible, it is necessary to understand how data is represented in BTO. Initially, a programmer assigns a keyword of matrix, vector, or scalar to all variables in the input program; BTO converts matrices and vectors into a hierarchical set of containers.

A container has an orientation associated with it, either *row* or *column*. The orientation encodes the container's layout in memory (row major vs column major) and ensures correct linear algebra operations (a row vector multiplied by a column vector is a different operation than a column vector multiplied by a row vector). Containers and scalars make up the types for which the grammars are

$$\begin{array}{ll} \text{orientations } O & ::= C \mid R \\ \text{types} \qquad\quad \tau & ::= O\!<\!\tau\!> \mid S \end{array} \tag{1}$$

where the container $O<\tau>$ represents a container with orientation O and containing type τ elements. A scalar (S) type represents the native type of the operation, for example a double or single precision value.

All data is represented using containers; so a column vector is represented as $C<S>$, and a column major matrix is represented as $R<C<S>>$. In the case of the matrix, the C container holds scalar values, telling BTO that elements in a given column of the matrix are stored contiguously.

Containers represent all data, including inputs, outputs, results and any temporary data structures. We have discovered that a set of rules describing container operations can express linear algebra. This knowledge is encoded in a linear algebra knowledge base, a portion of which is shown in Table 1. These rules describe the introduction of loops based on the types of the containers involved in an operation. Consider the

Algo	Op and Operands	Result Type
`add`	$O<\tau_l> + O<\tau_r>$	$O<\tau_l + \tau_r>$
`s-add`	$S + S$	S
`trans`	$O<\tau>^T$	$O^T<\tau^T>$
`s-mult`	$S \times S$	S
`rr-mult`	$R<\tau_l> \times R<\tau_r>$	$R<R<\tau_l> \times \tau_r>$
`cc-mult`	$C<\tau_l> \times C<\tau_r>$	$C<\tau_l \times C<\tau_r>>$
`dot`	$R<\tau_l> \times C<\tau_r>$	$\sum(\tau_l \times \tau_r)$
`outer1`	$C<\tau_l> \times R<\tau_r>$	$C<\tau_l \times R<\tau_r>>$
`outer2`	$C<\tau_l> \times R<\tau_r>$	$R<C<\tau_l> \times \tau_r>$
`scale`	$S \times O<\tau>$	$O<S \times \tau>$

Table 1. Sample of the linear algebra knowledge base used by BTO

first rule in the table; the algorithm expressed is an *add* of containers. The *Op and Operands* column is used to match the operation and operand types from the program representation with a rule from this table. We can explain the notation used in the *Result Type* column by considering this operation in terms of vector-vector addition. A vector-vector addition is the sum of corresponding operand elements. We see from the *Result Type* column that the result of container addition needs to be a container whose elements are the sum of the two operand containers. The summation of each element tells BTO that a loop is required.

The combination of utilizing containers to represent data and expressing operations as container operations enables reasoning about iteration space with any number of nestings within a container. This feature is exploited to represent partitions using the existing types.

3.3 Rules Controlling Data and Operation Partitioning

Using containers to represent data enables partitioning of data without significant changes to the remaining BTO framework. Additionally, the nature of containers provides an abstract method for introducing data partitions. However, more than the ability to partition data is required to safely partition a program. Program partitioning requires the ability to partition operations.

Data To explain partitioning, we introduce a new notation for a *type list* given as,

$$[O_n, \ldots, O_1] \text{ given the type } O_n<\ldots<O_1 < S>>, \tag{2}$$

where the list describes the ordered set of containers in a given type. Our notation for appending type lists is a comma and is overloaded to handle appending a single container or scalar type to a type list. In these examples

$$\begin{array}{c} R, [C] \quad \text{gives} \quad [R, C] \\ [R], [C] \quad \text{gives} \quad [R, C] \end{array} \tag{3}$$

we demonstrate the creation of column major matrices.

Using this notation we can describe the single rule required to determine legal data partitioning. Given some type, if the type list l contains a container with orientation O, it is legal to append a container with the same orientation to the type list, written as O, l. Consider a row vector $[R]$, the only legal method for partitioning the vector is to create a row of rows $[R, R]$. Figure 2 helps to visualize this partitioning, showing on the right a partitioned row vector. We can see from Figure 2 that we have introduced an additional row container to represent the partition.

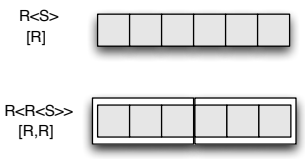


Fig. 2. A row vector on the top and a partitioned row vector on the bottom.

Additional containers can be added to any type; the number and location in the type list of added containers will specify the number and dimensionality of the partition.

Operations BTO represents linear algebra operations as series of binary and unary container operations. This level of the program is where partitioning decisions are made. Consider a simple case of a binary vector operation, $y \leftarrow x + z$. If BTO decides to only partition x, we can think of this as needing two loops to access x, but only a single loop to access y and z. This presents a problem when reasoning about the iteration space of the operation. BTO does not allow this situation to occur by making data partitioning decisions based on operations and updating all containers involved in the operation.

Deciding when to partition data based on operations ensures that all types of an operation are updated correctly which ensures iteration spaces will remain consistent. Similar to the set of rules that controls the reasoning of operation iteration spaces (Table 1), there is a set of rules that describes when a linear algebra operation can be partitioned. These rules describe how to express a partition by modifying the types involved in the operation. Table 2 shows the partition rules. The first row shows how to partition addition and subtraction operations. First, the operation must match the description given in the *Operation* column; for addition this states that both operand and return types all must match. The remaining three columns describe how each type can be updated. For the `add/sub` algorithm there is only one unique type τ; and if that can be written as the set of appended lists and container described in the *List* column, then the operation can be partitioned. The *Partition* column describes the modification to the given type. In this example we can add a new container with orientation O to the outermost level of the type.

Partitioning an operation in isolation from the remaining program provides the potential for incorrect loop generation and can limit optimization potential. We handle

Algo	Operation	Type List		Partition
`add/sub`	$\tau = \tau + \tau$	τ	u, O, l	O, u, O, l
`mult-bc`	$\tau_c = \tau_a \times \tau_b$	τ_b	u_b, R, l_b	R, u_b, R, l_b
		τ_c	u_c, R, l_c	R, u_c, R, l_c
`mult-ac`	$\tau_c = \tau_a \times \tau_b$	τ_a	u_a, C, l_a	C, u_a, C, l_a
		τ_c	u_c, C, l_c	C, u_c, C, l_c
`mult-ab*`	$\tau_c = \tau_a \times \tau_b$	τ_a	u_a, R, l_a	R, u_a, R, l_a
		τ_b	u_b, C, l_b	C, u_b, C, l_b
`scale`	$\tau = S \times \tau$ $\tau = \tau \times S$	τ	u, O, l	O, u, O, l
`trans`	$\tau_t = \tau_n$	τ_t	u_t, O_t, l_t	O_t, u_t, O_t, l_t
		τ_n	u_n, O_n, l_n	O_n, u_n, O_n, l_n
			where $O_t \neq O_n$	

Table 2. Linear algebra partitioning rules where algorithms marked with '*' will require a reduction operation.

this by requiring the propagation of a partition decision made using a single operation throughout the rest of the program.

3.4 Propagation of Partition Decisions Throughout a Program

As mentioned in Section 3.3, whenever a partition decision is made at the operation level, that decision must be propagated throughout the rest of the program. The partition decision is propagated to all connected operations in the dataflow graph. The partitioning rules that control legal partitioning of operations (Table 2) are used to determine type updates for any containers involved in the operations that are connected to the original partition decision.

Consider the following set of operations:

$$\begin{aligned} w &\leftarrow Az \\ y &\leftarrow Ax. \end{aligned} \tag{4}$$

The dataflow graph for these operations are shown in figure 3, with a partition decision made at the operation $w \leftarrow Az$ as shown in the figure with lines through A and w. To propagate this decision throughout the rest of the program, all operations connected to this decision must be considered for update. In this example, the operation $y \leftarrow Ax$ is connected through the matrix A and so is updated with a partition of vector y.

Currently the BTO compiler requires an update to be performed in all cases. This is not necessary when partitioning a program but is a design decision to ensure loop fusion potential remains after introducing partition decisions. Consider again the operations above and their dataflow graph in Figure 3.

In this example the two matrix-vector multiples can be fused to reduce the traffic of matrix A. However, if we partition a single matrix-vector multiplication and not the other, as shown in Figure 3 with lines through matrix A and vector w, we have lost the ability to perform loop fusion on the set of operations. BTO can still perform loop fusion if the partition decision is propagated through the rest of the program. This need to propagate partitions affects programs that have many connected operations, most often

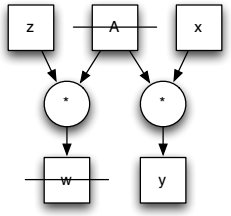


Fig. 3. Example dataflow graph of the operation $y \leftarrow Ax$ and $w \leftarrow Az$. A partition of the operation $w \leftarrow Az$ is represented with lines through A and w.

caused by common data elements. Programs that have sets of independent operations will not be affected by this propagation requirement. In practice, this required propagation has not degraded performance.

After a program is completely updated with all partition decisions, the dataflow graph moves on to the optimization phase. We next describe how different partitioning versions fit into the existing optimization process.

3.5 Partitioning and the Optimization Search Space

The optimization phase of the BTO compiler is designed to generate many optimized implementations of the input program. Similarly, we generate many partitioned implementations of the input program. The generation of partitioned versions occurs before the existing optimization phase. Each partitioned version is then fed to the optimization phase, which is unchanged from the description in Section 3.1.

The algorithm for generating partitioned versions is similar to that used in the optimization. The process begins with the single version specified by the programmer. The algorithm visits each node in the graph once only taking action at nodes that are operations. When an operation node is found, the rules describing operation partitioning are used to generate versions for each legal partitioning. In each version, the partition information is propagated throughout the graph. This process repeats until all nodes in the graph have been visited. At this point versions representing single dimension partitioning have been generated. If more levels of partitioning are desired, the process can be repeated on any or all versions.

This algorithm is designed to generate partitioned versions of the program. Decision making about which decisions are good are intentionally left out of this process. This will eventually be controlled by a set of rules that can easily be modified to ensure portability and extensibility.

This algorithm has the potential to generate many duplicate versions. This is avoided by observing that the dataflow graph remains unchanged throughout the partitioning process, only type information is changed. We avoid duplication by creating representative strings from the type information in the graph. These strings provides a mechanism for quickly identifying and eliminating duplicates.

At the end of the partitioning process, we have many versions of the program. Each of these moves onto the existing optimization phase. The only difference being that now BTO is starting with many versions as opposed to the one version without partitioning. This has the potential to explode the optimization search space. For example if before partitioning we were generating ten versions and BTO can find one way to partition that problem, we can expect on the order of 20 versions. In practice with a single partitioning pass we generate between three and 100 partitioned versions increasing our search space significantly. On top of this, we have introduced partitions that require a size to be selected, for example number of threads or cache tile size. So if we have 100 partitioned versions and we need to try 1,2,3, and 4 threads, we have to consider 400 versions.

We are in the process of improving the mechanism that handles this search space. We plan to improve our analytic model and to look at pruning heuristics reduce this search space.

3.6 Mapping Iteration Space Representations to Code Generators

The BTO code generator only required small modifications to handle the partition information and to generate C code for cache tiles and utilize the Pthreads library for data parallelism. In this section we describe how the internal program representation expresses loop information and we describe how the code generator uses this representation to generate appropriate code.

Iteration Space Representation Section 3.1 shows how BTO represents a program with a dataflow graph. Iteration spaces are represented in the dataflow graph as subgraphs, where each subgraph contains nodes for any get, store and arithmetic operations as well as other subgraphs that represent inner loops. Figure 4 shows the dataflow graph of a matrix-vector multiplication where the iteration spaces are represented as subgraphs which in this case are shown as sequential for loops. We can see that there is a subgraph for each loop required to perform the operation. The outermost loop gets a row from the matrix A and an element from the vector y. The inner most loop performs a dot product with the row of A and the vector x, summing into the element of y.

As mentioned in Section 3.1 iteration spaces can map to sequential, parallel, or cache tiling language features. With subgraphs representing iteration spaces, a partitioned program will have a similar dataflow graph to the unpartitioned version, the main difference being the addition of subgraphs for each partition that was introduced. The code generator can convert any subgraph to either serial loops, parallel loops, or cache tiles, while only being concerned with the depth of a subgraph. The partitioning phase tells the code generator how many subgraphs have been introduced for parallelism and how many for tiling. Any remaining subgraphs are assumed to be serial loops. The code generator can then generate the correct C code based only on subgraph nesting depth.

C Code and Pthreads Currently only a Pthreads backend is implemented. The modularity of the code generator and the representation of the program means other parallel implementations can be generated by writing additional backends. This will allow for generating code for distributed systems, GPGPUs, and vector units simply by adding

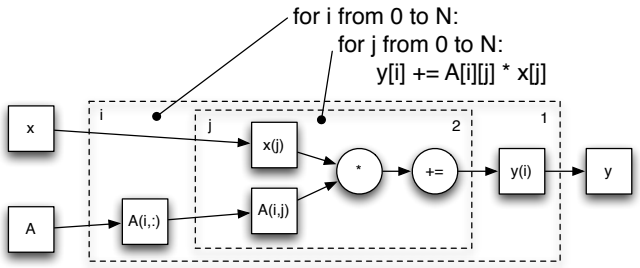


Fig. 4. Example dataflow graph of the operation $y \leftarrow Ax$ with loops expressed as subgraphs.

other code generation modules. This design was selected to ensure extensibility and portability, the decision to use Pthreads is not hard-coded into the BTO tool.

4 Results

In this section we present preliminary results for a representative set of routines. The partitioning framework has been implemented and can automatically introduce partitions for data parallelism. The code generator produces C code using Pthreads to implement the data parallel partitions. Currently the compiler generates most combinations of loop fusion and partitioned operations. This brings the number of unique implementations that BTO must evaluate from many hundred to many thousand. Additionally, the implementations that have a partition in them have at least one parameter that needs searching, specifically number of threads. This grows the search space even larger to many tens of thousands. The mechanism in place evaluating unique implementations can not yet handle this many versions and can not automatically search those implementations with parameters. This limits the results presented in this section.

4.1 Number of Implementations Generated and Search Time

The time to identify the best implementation is a function of the number of versions BTO creates. Here we show that even with a significant increase the number of versions generated, we still have usable search times.

The number of unique implementations and the corresponding search time is shown in Table 3. The number of unique implementations represents serial versions with combinations of loop fusions and unique methods for parallelization where each of those implementations then has combinations of different loop fusions. The search time represents the time from beginning of compilation until the best performing implementation is found. The empirical tests dominates this time.

As previously mentioned BTO cannot automatically search optimizations with parameters. Table 3 represents search times for a single number of threads equal to the number of available cores only. Additionally this data represents the case when BTO

only introduces a single dimension partition to a given problem. Allowing anything else for these two features would significantly increase the number of implementations created and the search times.

For an auto-tuning tool to be useful it must produce a result in a reasonable amount of time. BTO uses a hybrid analytic machine model and empirical testing to identify the best performing implementation. Unfortunately the analytic model is not yet capable or predicting performance for parallel implementations. This temporarily forces a search to be completely empirical. Future improvements to the model will greatly reduce the search times.

Kernel	Specification	Unique Versions Total (parallel)	Search Time (seconds)
MADD	$C \leftarrow A + B$	2 (1)	1.6
MMUL	$C \leftarrow AB$	40 (39)	29.1
AATX	$y \leftarrow AA^T x$	6 (4)	2.5
BiCGK	$q \leftarrow Ap$ $s \leftarrow A^T r$	11 (8)	4.4
GEMVER	$B \leftarrow A + u_1 v_1^T + u_2 v_2^T$ $x \leftarrow \beta B^T y + z$ $w \leftarrow \alpha B x$	8456 (7808)	2825.4
GESUMMV	$y \leftarrow \alpha Ax + \beta Bx$	64 (52)	95.5
DSCAL	$x \leftarrow \alpha x$	2 (1)	1.5
VADD	$x \leftarrow w + y + z$	5 (3)	9.5
WAXPBY	$w \leftarrow \alpha x + \beta y$	13 (9)	19.9

Table 3. Number of unique implementations and search time.

4.2 Performance Results

The results in this section show extracting data parallelism from certain kernels can provide significant speedups. We show the results produced by BTO for the serial and parallel case as compared to existing state-of-the-art libraries.

The results presented in this section are from an Intel Core2Quad running at 2.44 GHz with 4 GB of memory. The Intel compiler version 10.1 was used with the following flags used: *-O3 -msse3*. Comparison is against the Intel MKL library version 10.2.5 with four threads used. All data is gathered in 32 bit mode including use of the 32 bit MKL library. All parallel BTO implementations use four threads. Space limitations prevent results from other systems being presented, however results are similar on an Intel Core2Duo and a two socket, four core AMD Opteron.

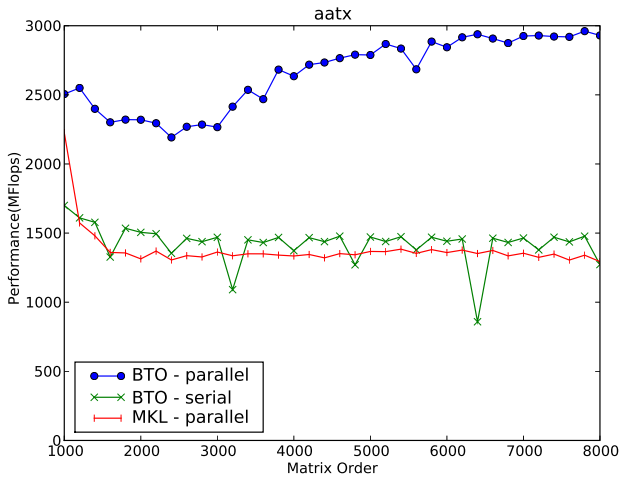
The majority of operations examined here are memory bound operations and will benefit most from improvements in memory traffic. BTO achieves this with loop fusion, which is performed in both the serial and parallel results presented. For fused memory

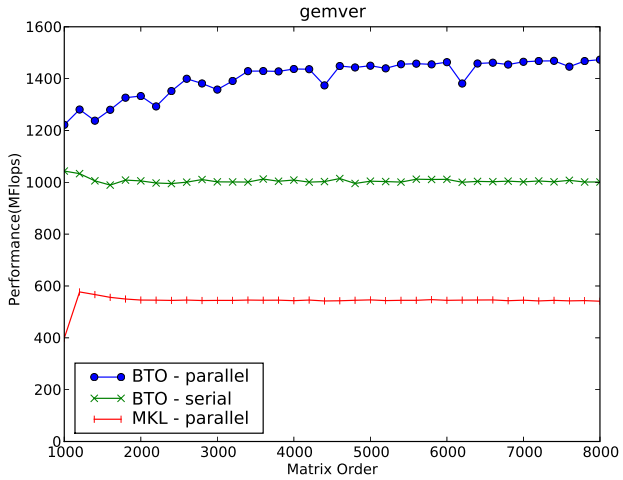
bound operations, we cannot expect linear scaling. Fusion of memory bound operations decrease memory-to-processor data movement resulting in an operation bound on cache-to-processor data movement. On the computer presented here, utilizing additional cores increases the available bandwidth between cache and processors. This pushes the bottleneck back to a memory-to-processor problem. Based on memory bandwidth, calculations show that with operations typical of those presented here we expect an upper bound on performance improvements of approximately 3x from four cores. Routines with higher memory traffic to operation ratios have lower expectations.

Figure 5 shows the performance in MFlops on the y-axis (higher is better) and a range of matrix orders on the x-axis for the AATX, GEMVER, and MMUL kernels. The lines labeled *BTO - serial* represent the best performing serial implementation produced. For AATX and GEMVER this represents a heavily loop fused implementation with reduced memory traffic as compared to the MKL approach. Operations such as these tend to be performance restricted completely by the memory bandwidth and reducing memory traffic generally provides the best opportunity for performance increases. In AATX and GEMVER, the reduction in memory traffic enables the BTO performance on a single core to outperform MKL when given four cores. In the case of MMUL, there is not opportunity to reduce memory traffic through fusion, and MKL is able to perform optimizations such as data repacking that provide a significant advantage over BTO.

The lines labeled *BTO - parallel* represent the best performing parallel implementation. Again, for AATX and GEMVER these represent heavily loop fused implementations running in parallel. The performance difference between the BTO parallel case and the BTO serial case we can attribute to data parallelism. In Figure 5 we see the AATX kernel achieves slightly less than a 2x performance increase in the best case. The GEMVER kernel sees approximately a best of a 65% speedup. MMUL improves by as much as 6 times for a range of approximately 1000 to 2000. In this range, the data parallelism improves cache utilization by providing additional cache space from other cores and reducing the storage requirements of two of the matrices. For larger sizes this effect stops occurring and we achieve approximately a 4x speedup.

Table 4 shows a summary of the performance as measured in MFlops. The first two kernels are matrix-matrix operations, the next four kernels are matrix-vector operations and the last three kernels are vector-vector operations. The matrix-matrix operations improve performance from BTO serial to BTO parallel by 3% minimum for MADD and a maximum of 6x for MMUL with a matrix order of 1000. For the matrix-vector operations, the speedups from the serial version of BTO to the parallel version range from a 10 percent slow down for GESUMMV with a matrix order of 1,000 to a 2.4 times speedup with BiCGK for a matrix order of 8,000. With the exception of small orders for BiCGK, GESUMMV, and all cases of MMUL, the BTO parallel version is able to outperform the MKL implementations, in the best case showing a speedup of 2.7 times faster. It is worth noting that approximately 66 percent of the overall speed up over MKL is caused by optimizations to the serial case leaving the parallel implementation responsible for approximately 33 percent of the improvements. BTO does not employ optimizations that target Level 3 operations such as MMUL. Cache tiling will help our





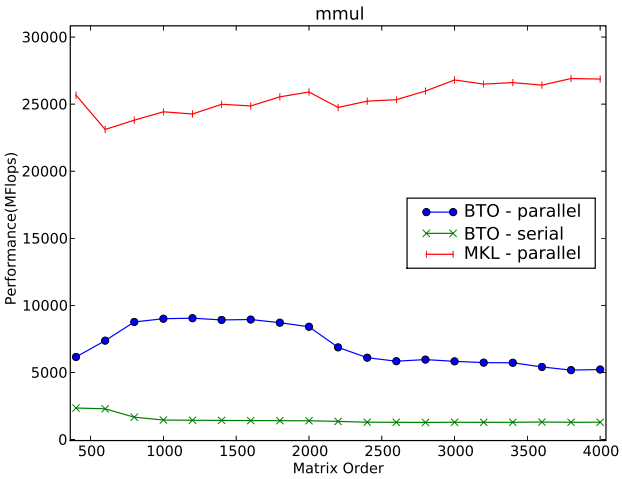


Fig. 5. Performance results for AATX, GEMVER, and MMUL kernel.

implementation of MMUL, however without employing data repacking and generating our own vector code it is unlikely BTO will be competitive with MKL in these cases.

The last two kernels in Table 4 are vector-vector operations. With this type of operation we observed considerably less performance change when parallelism is extracted. Comparing serial BTO to parallel BTO the performance changes range from a three percent slow down for WAXPBY for a vector size of 1,000,000 to a nine percent increase for DSCAL with a vector size of 1,000,000. These vector-vector operations are completely bound by memory bandwidth, something on this computer, parallelism can do little for. When benchmarking our test system with STREAM [8] we observed about a 5% increase in usable bandwidth from memory to the processor when using all cores instead of one which approximately matches the performance changes we observe. When the parallel BTO results are compared to MKL there are performance increases ranging up to 2.2 times faster for WAXPBY for a vector size of 10,000,000. These performance changes are dominated by optimizations done in the serial case to reduce memory traffic and unneeded temporary data structures.

Kernel	Size	BTO - serial	BTO - parallel	MKL - parallel
MADD	1,000	167	173	117
	8,000	168	179	113
MMUL	1,000	1455	9004	24429
	4,000	1290	5219	26869
AATX	1,000	1705	2550	2240
	8,000	1277	2925	1280
BiCGK	1,000	1845	2090	2525
	8,000	1293	3115	1304
GEMVER	1,000	1035	1224	415
	8,000	1015	1470	540
GESUMMV	1,000	1360	1220	1495
	8,000	1350	1640	1330
DSCAL	1,000,000	311	340	316
	10,000,000	272	326	270
VADD	1,000,000	283	292	204
	10,000,000	282	288	155
WAXPBY	1,000,000	710	690	388
	10,000,000	710	733	328

Table 4. Summary of performance results. Performance measured in MFlops and Size represents matrix order for first six and vector length for last three kernels.

5 Conclusions and Future Work

This paper has presented a framework for automatically introducing partitions to linear algebra operations to introduce data parallelism. The performance results from the work show that this approach has a great deal of promise. However, there is more work required to efficiently handle the many thousands of versions that must be evaluated to find the best performing implementation.

As mentioned, the BTO uses a hybrid analytic and empirical search strategy. We plan to extend our analytic model to determine when tiling for cache or extracting data

parallelism will be beneficial and to provide an appropriate tile size range that can be empirically searched. Additionally, we plan to heuristically prune versions and parameter sizes based on hardware information. For example, when searching for the number of threads that achieves the best performance, BTO can probe for the number of processors limiting the range that must be evaluated. Similarly, we plan to probe for cache sizes to limit tile size searches.

Acknowledgments

This research is funded by NSF grant 0846121, *CAREER: Bridging the Gap Between Prototyping and Production*.

References

1. Netlib blas. http://www.netlib.org/blas/index.html
2. Belter, G., Jessup, E.R., Karlin, I., Siek, J.G.: Automating the generation of composed linear algebra kernels. In: SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. pp. 1–12. ACM, New York, NY, USA (2009)
3. Bikshandi, G., Guo, J., Hoeflinger, D., Almasi, G., Fraguela, B.B., Garzarán, M.J., Padua, D., von Praun, C.: Programming for parallelism and locality with hierarchically tiled arrays. In: PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming. pp. 48–57. ACM, New York, NY, USA (2006)
4. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. SIGPLAN Not. 43(6), 101–113 (2008)
5. DeRose, L., Gallivan, K., Gallopoulos, E., Marsolf, B.A., Padua, D.A.: Falcon: A MATLAB interactive restructuring compiler. In: Proc. 8th International Workshop on Languages and Compilers for Parallel Computing. pp. 269–288 (1995)
6. Gottschling, P., Wise, D.S., Adams, M.D.: Representation-transparent matrix algorithms with scalable performance. In: ICS '07: Proceedings of the 21st annual international conference on Supercomputing. pp. 116–125. ACM, New York, NY, USA (2007)
7. Low, T.M., van de Geijn, R.A., Van Zee, F.G.: Extracting SMP parallelism for dense linear algebra algorithms from high-level specifications. In: PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming. pp. 153–163. ACM, New York, NY, USA (2005)
8. McCalpin, J.D.: Stream: Sustainable memory bandwidth in high performance computers. http://www.cs.virginia.edu/stream/ (April 2010)
9. Puschel, M., Moura, J., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R., Rizzolo, N.: Spiral: Code generation for dsp transforms. Proceedings of the IEEE 93(2), 232–275 (Feb 2005)
10. Siek, J.G., Lumsdaine, A.: The matrix template library: A unifying framework for numerical linear algebra. In: ECOOP ?98: Workshop on Object-Oriented Technology. pp. 466–467. SpringerVerlag (1998)
11. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimizations of software and the ATLAS project. Parallel Computing (27), 2001