Fast recursive matrix multiplication for multi-core architectures

Gudula Rünger, Michael Schwind

Department of Computer Science, Chemnitz University of Technology, Germany

Abstract

In this article, we present a fast algorithm for matrix multiplication optimized for recent multicore architectures. The implementation exploits different methodologies from parallel programming, like recursive decomposition, efficient low-level implementations of basic blocks, software prefetching, and task scheduling resulting in a multilevel algorithm with adaptive features. Measurements on different systems and comparisons with GotoBLAS, Intel Math Kernel Library (IMKL), and AMD Core Math Library (AMCL) show that the matrix implementation presented has a very high efficiency.

1. Introduction

The performance of basic linear algebra kernels is crucial for the efficient implementation of scientific codes. One important linear algebra kernel is the dense matrix multiplication, which we consider in this article. The efficiency of sequential or parallel matrix multiplication strongly depends on the hardware architecture used and in the past decades many efficient parallel algorithms and implementations for matrix multiplication have been proposed for high performance computers. However, recent and future parallel platform will provide new architectural features, including multicore processors and memory hierarchies. Thus, it is required to revisit standard algorithms like matrix multiplication in order to provide new efficient implementations and libraries for this new generation of parallel hardware.

In this paper, we propose a new flexible parallel implementation for dense matrix multiplication resulting in very fast code compared with efficient libraries like GotoBlas [1], ACML [2], and IMKL [3]. The implementation is based on a recursive approach of dense matrix multiplication which stops at basic kernel operations for matrix blocks. The recursive algorithm induces a task structure of the matrix multiplication and a flexible computation order of the basic operations on matrix blocks. Special care has been taken to implement these basic kernel operations using specific data layouts and a software prefetching mechanism for overlapping computations and memory accesses.

The contribution of this article includes the following issues. The combination of recursive matrix multiplication and task oriented computations results in a flexible parallel implementation which can be adapted to a threaded execution for different hardware characteristics. A task handling library has been implemented which allows varying assignment schemes of tasks to cores of the systems; the library can be used for any block-based linear algebra code. Basic kernel operations for blocked matrix multiplication are provided in order to optimally exploit the characteristics
of the cores. A software prefetch mechanism for basic kernel operations leads to high efficiency of consecutive kernel operations on each single core.

The implementation of a parallel dense matrix multiplication presented in this paper is provided as a library written in C++ to be used in more complex application programs. Performance tests have been performed on three multicore systems, an AMD Opteron System (Egypt) without shared cache, an AMD Opteron System (Barcelona) where 4 cores share a level 3 cache, and an Intel XEON System (Cloverstown) where 2 cores share a second level cache. The tests and comparisons with popular efficient matrix multiplication, such as GotoBLAS, AMCL, and IMKL have shown, that the recursive task based matrix multiplication with block software prefetching has very high efficiency.

The rest of the paper is as follows. Section 2 describes the upper level recursive matrix multiplication and the task creation mechanism. Section 3 presents the task handler and scheduling library. The implementation of basic matrix operations and the block prefetching mechanism are explained in Section 4. Section 5 provides experiments and an evaluation of speedup and efficiency. Section 6 summarizes related work and Section 7 concludes.

2. Recursive Matrix Multiplication and Task Handling

For the task-based implementation, the matrix multiplication $C = A \cdot B$ of an $m \times k$ matrix $A$ and a $k \times n$ matrix $B$, (i.e. $c_{ij} = \sum_{k=1}^{k} a_{ik}b_{kj}$, $i = 1, \ldots, m$, $j = 1, \ldots, n$ for $C = (c_{ij})$, $A = (a_{ij})$, $B = (b_{ij})$) is structured in a recursive way. The matrix product is formulated in terms of submatrix operations rather than row or column operations. The input matrices $A$ and $B$ as well as the result matrix $C$ are partitioned into 4 submatrices each and the matrix computation is formulated as follows

$$
\begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix}
= 
\begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix}
\begin{pmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{pmatrix}
= 
\begin{pmatrix}
A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\
A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22}
\end{pmatrix}.
$$

(1)

This leads to eight recursive matrix multiplications of smaller submatrices $A_{11} \cdot B_{11}$, $A_{12} \cdot B_{21}$, $A_{11} \cdot B_{12}$, $A_{12} \cdot B_{22}$, $A_{21} \cdot B_{11}$, $A_{22} \cdot B_{21}$, $A_{21} \cdot B_{12}$ and $A_{22} \cdot B_{22}$, whose results are combined according to Formula (1). These matrix multiplications of submatrices are partitioned correspondingly such that a recursive algorithm results which stops at a certain level. For this recursive matrix multiplication, we propose a multithreaded implementation for multicore architectures (using Pthread, with C or C++) which realizes an adaptive, cache-aware and efficient implementation. The implementation uses several features including a suitable array layout in Z-Morton ordering, a blocked matrix version with basic blocks on the lowest level, software data prefetching, a task creation mechanism intertwined in the recursion, and a thread pool with a synchronization mechanism for shared memory writes. The features are described in the following subsections.

2.1. Recursive task creation

The recursive matrix multiplication is implemented in a task-oriented algorithm, which consists of a task creation part and a task execution part. The task creation process is intertwined with the recursive matrix decomposition but can stop at a predefined unroll level unroll (see Algorithm 1). The task created is inserted into a task pool (addToPool), from which it is accessed for execution. Depending on the unroll level two different types of tasks are created, which are either matrix operations on basic blocks of size bs or larger matrix multiplications whose recursive decomposition may remain within the same task.

If the current size of the matrix block is larger than basic block size bs and the user-defined unroll levels is not reached, eight recursive calls to the task creation function rectaskCreate are performed with updated recursion level level. The new blocks for the recursive calls are computed from the current blocks with respect to bs. More precisely, the current blocks are divided into blocks of size bs in each dimension and the subblocks get approximately half of the blocks of size bs. If the current block size is not a multiple of bs, a last smaller block exists. The actual implementation just recalculates the sizes of the new submatrices and uses pointers to the matrix data. The recursion also computes a write tag wtag for the matrix C or of subblocks the matrix C, respectively. This tag is important for an implementation, in which the results of subcomputations are immediately written to the memory space allocated for matrix C. The
Algorithm 1: recTaskCreate

\[/* no execution necessary */
if (rows(A) = 0 or cols(A) = 0 or cols(B) = 0) then
    return;
/* recursion done until unroll or the matrices are smaller than bs */
if (rlevel = unroll or (rows(A) \leq bs and cols(B) \leq bs and rows(B) \leq bs)) then
    /* add task to taskpool */
    addToPool(A, B, C, rlevel);
return:
increment recursion level rlevel;
compute new subblock sizes with respect to basic block size bs;
decompose each matrix A, B or C into 4 submatrices;
recTaskCreate(A_{11}, B_{11}, C_{11}, wtag(C_{11}), rlevel);
...  
recTaskCreate(A_{22}, B_{22}, C_{22}, wtag(C_{22}), rlevel);

write tag can be used to organize the tasks such that conflicts are avoided, see routine addToPool in Algorithm 2. The second possibility, which we explore, is a synchronization mechanism with lock operations.

The recursive task creation leads to a recursive decomposition of the matrix multiplication according to (1) which forms a tree of computations, whose leaves are tasks to be executed by the threads of the multithreaded program. The user definable parameter unroll can be used to adapt the number of tasks to the number of threads available. For example the parameter unroll can be set such that the number of tasks corresponds to the number of threads.

2.2. Array layout and memory allocation

The matrices $A$, $B$, and $C$ are stored according to a two-level array layout which is suitable for the recursive decomposition as well as for the blocked character of the algorithm. First, a two-dimensional array $A$ of size $m \times k$ is decomposed into basic blocks of size $b_1 \times b_2$. Currently, the block size is $32 \times 32$, since experiments have shown the best efficiencies, but $b_1 \times b_2$ can be an arbitrary basic block-size. The basic blocks of matrix $A$ are ordered according to the Z-Morton Form shown in Figure 1. This upper level storage is suitable for the recursive decomposition into subblocks described in Section 2.1. If the number of rows $m$ or the number of columns $k$ is not a multiple of 32, the last basic blocks are fully allocated but only partially filled. If the number of basic blocks is not a power of 2, a Z-Morton ordering results as shown in Figure 1, where $6 \times 7$ basic blocks exist. In the second level of the array layout, the elements within a basic block are stored in column major order form used by the basic block implementation. This two level storage scheme supports spatial locality for accessing array elements by the blocked recursive matrix multiplication for sequential as well as for parallel execution.

For the implementation on a multicore system as considered in this article, there is a second aspect associated with memory allocation. The multicore systems are shared memory architectures consisting of several multicores accessing the same shared memory. For some of these systems, the shared memory is fragmented into parts which are associated with the multicore processors but can be accessed by all multicore processors of the system. An example are AMD multicore systems, for which such multicore processors are called numa nodes. For the AMD systems

Algorithm 2: addToPool

\[input: A, B, C, wtag, rlevel\]
create task $T (A, B, C, rlevel)$;
if (there exists a task $\bar{T}$ with the same wtag) then
    mark Task $T$ as waiting;
    set $T$ as subsequent task of $\bar{T}$;
else
    mark task $T$ as ready;
investigated in our experiments, we exploit two different memory allocation schemes. The first is the allocation of all matrix elements on the memory parts of one multicore processor; this is the standard mode of memory allocation. The second memory allocation scheme exploits the numa characteristics and stores parts of the array in all memory fragments of the system. This memory allocation format can be achieved by using specific operating system calls. The Z-Morton block order layout of the array remains unchanged. Since the matrix data are closer to the single core and multicore, the second memory allocation may lead to a higher bandwidth. The experiments section shows the results of this comparison.

3. Task Administration Library TAL and Basic Block Operations

In our approach, the task administration and execution is separated from the task creation in the application algorithm, i.e. the recursive matrix multiplication. For the task administration, we have designed and implemented a task handling and scheduling library TAL. This library can be considered as a specific task pool library with additional features for an effective task mapping and scheduling on multicore systems with memory hierarchy. The library is responsible for storing tasks created by the application algorithm and for scheduling and mapping task according to the precedence constraints specified for the tasks.

The execution of tasks is performed by threads running on the multicore system. The number of threads corresponds to the number of cores. Thus, the number of threads remains constant during the execution of the application, independent from the recursion level of the matrix multiplication or the number of ready tasks. This leads to a low overhead for thread creation. Also, the same recursive algorithm as described in Sect. 2 can be used to get a sequential program or a parallel program for different numbers of cores. Program execution is initiated by the thread, which asks an associated task scheduler object (tso) of the library to assign a suitable task. The pseudocode in Algorithm 3 illustrates the code performed by a thread.

**Algorithm 3:** thread pseudo code

```plaintext
while (there are tasks in the system) do
    t = getNextTask; /* ask leaf tso for a new task */
    if (successful assignment) then
        execute task t; /* sequential execution */
        for all tasks s depending on t: mark s as ready; delete task t;
```

The library TAL contains a set of task scheduler objects (tso), which are organized as a hierarchical tree structure, see Fig. 2. This tree structure is independent from the recursive task structure of the application algorithm. Each leaf tso of the tree is associated with one or more specific threads executing the application. Tasks of the application are
put into the task scheduler objects and threads asks their associated tso for tasks. Internally, a tso has a lifo queue to store tasks to be executed. Tasks accessing the same data are inserted into the same hierarchy level and, thus, they are executed by the same thread or set of threads leading to data locality.

The hierarchical tree structure of tsos is used to organize the task assignment, when the task scheduler object of a specific thread is empty. If the task scheduler object tso associated to a thread is empty, the thread asks the parent task scheduler tso∗, which can ask his parent tso∗∗, if tso∗ is also empty. Otherwise sibling scheduler objects are asked. In every step of this recursive structure the task provided by the parents tso are decomposed into smaller (more fine grain) tasks. The task decomposition is a property of the application program. For the matrix multiplication, we use a recursive decomposition. This structure can be used to exploit the memory hierarchy of the multicore systems effectively. The specific task assignment strategy is illustrated in Algorithm 4.

```
Algorithm 4: getNextTask
1   if (LIFO of tso not empty) then
2       return task t;
3   else
4       if (iam is a child tso with parent tso) then
5           ask parent tso for task t;
6       if (task t from parent tso is assigned) then
7           if (task t can be decomposed into subtaks) then
8               decompose t into subtasks t₁, ..., tₙ;
9               enqueue t₁, ..., tₙ into tso;
10              goto Label 1; /* goto line 1 */
11           else /* task cannot decomposed into subtasks */
12              return task t;
13       else /* task t from parent tso is not assigned */
14           ask sibling tso for tasks; /* task stealing */
15           if (task stealing successful with task t) then
16               if (task t can be decomposed into subtaks) then
17                   decompose t into subtasks t₁, ..., tₙ;
18                   enqueue t₁, ..., tₙ into tso;
19                   goto Label 1; /* goto line 1 */
20               else /* task cannot decomposed into subtasks */
21                  return task t;
22           else /* no task assigned */
23               return NIL;
24   else
25       return NIL;
```

The basic block operation used in block based recursive dense matrix multiplication are matrix multiplication for basic blocks of size 32 × 32. The basic kernel operations are implemented as assembler kernels to be computed sequentially on one core. As mentioned before, the tasks mapped to cores can be such basic kernels or bigger tasks consisting of a matrix multiplication for larger blocks. The sequential code for larger blocks continues the recursive decomposition until the basic block size is reached and a sequence of basic kernel operations is performed on the same core. Since the kernel operations are produced by the same parent operation and the array data are stored in the blocked Z-Morton order, there are good spatial locality conditions. Temporal locality can also be achieved because of the specific recursive decomposition. Additionally, we exploit a software prefetch mechanism for basic data blocks in
which entire blocks of size $32 \times 32$ are prefetched. The prefetched blocks are from matrices $A$ and $C$ (on Egypt and Cloverstown) but a prefetching of blocks from $A$ and $B$ can also lead to good performance (for Barcelona). A data block for the next kernel operation is fetched just before the actual computation of the current kernel operation starts. This is done according to a list of successor tasks maintained for each basic kernel operation. Several basic blocks fit into the Level 1 Cache so that cache locality is exploited. The reuse of data takes place between several kernel operations executed sequentially on one core.

4. Experiments and Evaluation

The efficiency of the blocked recursive algorithm has been tested on three different multicore systems with characteristics described in Table 1; the names are given according to the codenames of the manufactures. Figure 3 presents performance results for the multiplication of quadratic matrices on the test systems with the maximum number of cores available, i.e. 16 cores on Barcelona or Egypt and 8 cores on Cloverstown. The diagrams for Barcelona or Egypt compare the performance of the matrix multiplication presented in this article (recursive) with the AMD specific ACML and with GotoBLAS. For ACML and GotoBLAS two implementation versions are considered, a standard one (named acml and goto) and a version, in which the data are distributed across the memory parts associated with processors (named acml numa and goto numa). The recursive matrix multiplication uses one tso object and a recursion level 2, which is the best variant shown by measurements tests. On the system Barcelona 91.88 GFlop/s, which is 75.86% of the peak performance, and on the system 63.62 GFlop/s, which is 86.44% of the peak performance, are achieved. The diagram show strong variations for the numa version which may be due to unbalanced access to the memory controller. Also the zigzag line of the measurements for the recursive implementation may be caused by concurrent accesses to the same controller in some cases. The numa version of memory layout is not possible for the system Cloverstown and the diagram presents only three lines. It can be observed that the recursive variant outperform the other for matrix sizes smaller 2500.

Figure 4 shows different versions of the blocked recursive algorithm presented in this article. A recursive version (no optimization) is compared with a numa version (numa), a version using the block prefetch (prefetch), and a version using both optimizations (numa+prefetch) on the systems Barcelona and Egypt. The first two diagrams show that the prefetch version is only useful together with the numa distribution of data; this is because the memory controller in

<table>
<thead>
<tr>
<th>Figure</th>
<th>Barcelona</th>
<th>Egypt</th>
<th>Cloverstown</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min</td>
<td>max</td>
<td>avg</td>
</tr>
<tr>
<td>4 numa+prefetch</td>
<td>0.18</td>
<td>91.88</td>
<td>78.09</td>
</tr>
<tr>
<td>numa</td>
<td>0.20</td>
<td>82.08</td>
<td>70.74</td>
</tr>
<tr>
<td>prefetch</td>
<td>0.15</td>
<td>61.30</td>
<td>51.32</td>
</tr>
<tr>
<td>no optimization</td>
<td>0.22</td>
<td>60.57</td>
<td>50.22</td>
</tr>
<tr>
<td>6 locks</td>
<td>0.16</td>
<td>89.42</td>
<td>70.81</td>
</tr>
<tr>
<td>dependencies</td>
<td>0.18</td>
<td>91.88</td>
<td>78.09</td>
</tr>
</tbody>
</table>

Table 2: Minimum, Maximum and Average GFlops rate from Fig. 4 and Fig. 6 calculated over all matrix sizes.
Figure 3: Comparison: Vendor-BLAS, Goto-Blas, and recursive implementation; shown is the GFlops rate for different matrix sizes.

Figure 4: Comparison: Prefetch and Numa; shown is the GFlops rate for different matrix sizes.
the non-numa case (all data are stored at one processor) cannot satisfy all prefetch accesses which leads to higher latency. For the system Egypt, the prefetch non-numa version gets faster for matrices of size 3000 and higher. This is because the matrices are too large to be stored at one processor and, thus, several memory controllers are used, which leads to the same performance advantage as in the case numa+prefetch. The differences between the optimized and the non-optimized versions are higher on Barcelona than on Egypt. This can be explained with a higher peak performance of Barcelona and a smaller memory bandwidth than Egypt, which means that the memory controller on Egypt get requests more frequently. On the system Cloverstown, only the optimization prefetch should be used. It can be seen that, the prefetch optimization leads to much higher performance. The performance results of Figure 4 are summarized in Table 2.

Figure 5 shows experiments concerning the task pool variant used. A hierarchical task pool (hierarchy aware) is compared with a flat task pool (flat) for the cases prefetch and numa+prefetch. Only slight differences can be observed for smaller matrix sizes. A small performance improvement is measured for the system Cloverstown.

Figure 6 shows experiments with the two variants proposed for synchronizing the write accesses to the results matrix C. The basic version is the fast numa+prefetch implementation. On all three systems, the solution using write tags is faster than the program version with the Pthread locking mechanism. Table 2 summarizes the performance results of Figure 6. Finally, Figure 7 presents speedup values for the best blocked recursive matrix multiplication which shows almost optimal speedup for high matrix size and a good exploitation of the multicore systems. For the system Barcelona, superlinear speedup can be observed due to cache effects.

Figure 8 shows performance histograms of the basic block operations from a parallel matrix multiplications with 2048×2048 square matrices. For the histograms the range from 0 GFlop/s to the maximum available single processor performance was divided into 100 intervals. The performance of each basic block operation was measured and classified according to the interval to which it belongs. The histograms show the number of basic block operations in each interval divided by the total number of basic-block operations. The Barcelona platform shows a much more complex behavior than the Cloverstown platform. The implementations without the numa optimization show a concentration of basic block operations with a performance in between 1 GFlop/s and 3 GFlop/s while the implementation with numa has a concentration of basic block operations between 3.2 GFlop/s to 5.2 GFlop/s. This shows that the memory access after misses in the implementation without numa optimization takes longer as a result of the shared bandwidth of the memory controller.
Figure 6: Comparison: Locks vs. Dependencies; shown is the GFlop rate for different matrix sizes.

Figure 7: Speedup of best parallel recursive implementation vs. sequential recursive implementation for different matrix sizes.

Figure 8: Performance histogram of basic block operations of a $2048 \times 2048$ parallel square matrix multiplications on Barcelona and Cloversquad.
5. Related Work

Due to the importance of matrix multiplication for numerical software there are numerous articles about sequential and parallel algorithms and implementations. A good survey on recursive blocked algorithms for dense linear algebra including BLAS operations $C = C + A^T \cdot B$ for matrices $A, B, C$ is given in [4]. The article also presents performance results for blocked algorithms on single processors and small SMPs with up to 4 processors (of older processor models due to the publication date of that article). The results for an explicitly tuned implementation of the operation given above already indicate a good efficiency of recursive blocked algorithms compared to ATLAS [5] and a performance of more than 50% of the peak performance on an IBM PowerPC 604 platform.

Recursive data layouts for recursive matrix multiplication are studied intensively in [6] in the Cilk system. Specific storage layouts have been presented in [7]. A more recent experimental investigation for sequential recursive algorithms concerning cache behavior is given in [8]. Exploiting the memory hierarchy is also the focus of [9] and [10]. In [11], memory efficiency of matrix multiplication is studied in the context of irregular applications. A framework for matrix multiplication with optimized low level kernels is given in [12]. The newer multicore or multithreaded architectures are addressed in [13], [14], and [15]. The specific investigations presented in our paper, i.e. adaptive mapping techniques to cores or specific memory layouts in a multicore system, are not covered so far.

6. Conclusion

In this article, we have presented a block based recursive matrix multiplication implemented in an adaptive way with a task pool mechanism. The task pool mechanism has specifically been implemented for handling subtasks of blocked recursive algorithms and mapping them to cores of a multicore system. Basic block operations are provided by a library which also has been designed and implemented. Prerequisites of the implementation are specific data and memory layouts and a software prefetch mechanism for basic blocks. The experiments on AMD and Intel platforms show that the implementations reaches about 85% of the peak performance.

References