Parallel Matrix Multiplication based on Space-filling Curves on Shared Memory Multicore Platforms

Alexander Heinecke
Technische Universität München
Dept. of Informatics
80290 München, Germany
heinecke@in.tum.de

Michael Bader
Technische Universität München
Dept. of Informatics
80290 München, Germany
bader@in.tum.de

ABSTRACT

We present a parallel implementation of a cache oblivious algorithm for matrix multiplication on multicore platforms. The algorithm is based on a storage scheme and a block-recursive approach for multiplication, which are both based on a Peano space-filling curve. The recursion is stopped on matrix blocks with a size that needs to perfectly match the size of the L1 cache of the underlying CPU. The respective block multiplications are implemented by multiplication kernels that are hand-optimised for the SIMD units of current x86 CPUs. The Peano storage scheme is used to partition the block multiplications to different cores. Performance tests on various multicore platforms with up to 16 cores and different memory architecture show that the resulting implementation leads to better parallel scalability than achieved by Intel's MKL or GotoBLAS, and can outperform both libraries in terms of absolute performance on eight or more cores.

Categories and Subject Descriptors

G.4 [**MATHEMATICAL SOFTWARE**]: Algorithm design and analysis; Parallel and vector implementations

General Terms

Algorithms; Performance

Keywords

Matrix multiplication, multicore, parallelisation, space-filling curves, cache oblivious algorithms

1. INTRODUCTION

Dual- and quad-core processors have already become the standard CPUs for both workstations and high performance computers. In the near future, multicore and manycore (i.e. with more that 16 cores) processors will dominate the field and generate an urgent need for algorithms and implementations that scale well on such shared memory systems. With the parallel processor power increasing at a much faster pace than latency and bandwidth of main memory, it is to be expected that the access to memory will turn into the most important bottleneck. Hierarchical cache structures but also different concepts of connecting main memory to the processor cores will be designed to bridge this performance gap between memory and CPU. As a consequence, application software and numerical libraries need to be tuned in a cache-aware way to achieve satisfying performance. Inherently cache-efficient (cache oblivious) data structures and algorithms will be required – to avoid re-implementation and re-tuning of software and libraries for each new multicore platform and, more important, because recent results[8] indicate that already data structures and underlying algorithms need to be inherently cache-efficient to gain optimal performance on multi- and manycore hardware.

In this article, we study the performance of an inherently cache-efficient algorithm for matrix multiplication on various shared-memory multicore systems, and compare it to the `dgemm` implementation of established BLAS libraries – GotoBLAS[4, 5] and Intel's MKL[9]. Cache efficient approaches for matrix operations are usually based on block-structured and block-recursive techniques, as for example presented in [7], or, in combination with automated optimisation, in the ATLAS project[11]. A review of cache-aware and cache-oblivious approaches was given in [3]. In a recent paper, Gunnels et al.[6] compared the performance of cache-aware and cache-oblivious matrix multiplication, and showed that performance critically depends on the optimisation of the respective multiplication kernels.

In the present paper, we use a block-recursive approach that is based on Peano curves and which inherits excellent cache-efficiency from the space-filling curve's strong locality properties[1, 2]. The parallel implementation is based on OpenMP and distributes the work load according to the space-filling curve following an owner-computes principle. We studied the performance on various shared memory platforms with up to 16 cores (4×quad-core) and with different memory layouts. These include an Intel-Xeon-based workstation, which follows a UMA-based approach, where the multicore processor communicates with main memory via the so-called front side bus by using the main board chip sets. We compared the results to those on an AMD Opteron machine that uses a NUMA memory layout, where each CPU (socket) has its own private memory located very near to the socket.

First results show that TifaMMy [10] – our implementation of the Peano-curve based cache oblivious matrix mltiplication – shows better scalability than the respective implementations in GotoBLAS and MKL. On machines with 8 or 16 cores, TifaMMy even outruns GotoBLAS in absolute performance. Hence, TifaMMy seems to be able to deal with the gap between CPU and memory performance better than the other libraries.

2. CACHE EFFICIENT MATRIX MULTIPLICATION USING PEANO CURVES

The Peano-curve based algorithm for matrix multiplication was introduced in [1]. It derives from a block recursive numbering of the matrix elements that is based on 2D *iterations* of the Peano curve. Figure 1 illustrates the recursive structure of this element order, which can be defined via four block-numbering patterns: P, Q, R, and S. Matrix blocks are stored contiguously according to the order given by the recursive numbering patterns. The recursion is stopped on matrix blocks with a size that depends on the size of the level-1 (L1) cache (cf. section 3). On these *L1 blocks*, simple row-major order is used. To store matrices of arbitrary size, zero-padding both within the L1 blocks and to complete the 3×3 recursive scheme is used. Zero L1 blocks are neither stored nor processed during the later multiplication. However, for simpler implementation operations are currently performed on entire L1 blocks, even if they contain zero-padding.

The 3×3 block recursion within the numbering scheme motivates a block recursive approach for the matrix multiplication:

$$\begin{pmatrix} P_{A0} & R_{A5} & P_{A6} \\ Q_{A1} & S_{A4} & Q_{A7} \\ P_{A2} & R_{A3} & P_{A8} \end{pmatrix} \begin{pmatrix} P_{B0} & R_{B5} & P_{B6} \\ Q_{B1} & S_{B4} & Q_{B7} \\ P_{B2} & R_{B3} & P_{B8} \end{pmatrix} = \begin{pmatrix} P_{C0} & R_{C5} & P_{C6} \\ Q_{C1} & S_{C4} & Q_{C7} \\ P_{C2} & R_{C3} & P_{C8} \end{pmatrix}. \tag{1}$$

Here, each matrix block is named with respect to its numbering scheme and indexed with the name of the global matrix and the position within the storage scheme. A blockwise matrix multiplication then needs to execute block operations such as $P_{C0} \mathrel{+}= P_{A0}P_{B0}$, $Q_{C1} \mathrel{+}= Q_{A1}P_{B0}$, $P_{C2} \mathrel{+}= P_{A2}P_{B0}$, etc. In [1], the inherently local execution order shown in figure 2 was introduced for these block multiplications.

Recursive extension of this scheme leads to a block recursive algorithm that combines eight nested recursive procedures – one for each of the eight occuring combinations of numbering patterns. In [1], the following strong locality properties were shown for this algorithm:

- The element access to the matrices A, B, and C can be achieved entirely by increment and decrement operations on the indices: after a matrix element is accessed, the next access will be either to itself or to its direct left or right neighbour – as can be observed in equation (2).
- Any sequence of k^3 floating point operations is executed on only $\mathcal{O}(k^2)$ *contiguous* elements in each matrix. Vice versa, on any memory block of k^2 contiguous elements, at least $\mathcal{O}(k^3)$ operations will be performed.

As a result, the algorithm will cause at most $\mathcal{O}\left(n^3/(L\sqrt{M})\right)$ cache misses on a machine with a cache memory consisting of M cache lines of L elements each, which is asymptotically optimal (cf. [1]).

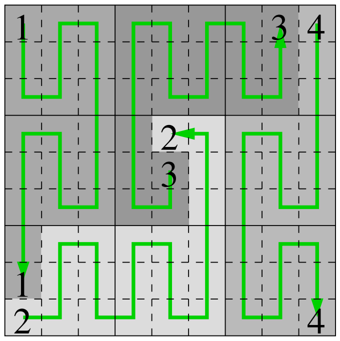

Figure 3: Partitioning of the result matrix according to the Peano element order; here, a 9×9 block matrix is partitioned into four contiguous clusters of L1 blocks.

3. PARALLEL IMPLEMENTATION

In TifaMMy[10], our implementation of this Peano-based algorithm, the recursion is stopped on the L1 blocks. Optimal cache and absolute performance is achieved, if the size of these L1 blocks is tuned to the size of the L1 caches of the underlying CPU cores. We experienced that exactly two L1 blocks need to fit into the L1 cache, which seems to result from the special reuse property of matrix blocks in the Peano algorithm: from one L1 block multiplication to the next, two L1 blocks are always reused. For example, for the Xeon processor used in the performance tests in section 4, blocks of size 52×52 proved to be optimal. In addition, a highly efficient implementation of the respective multiplication kernel, i.e. the multiplication of those L1 blocks, is required. In the present work, we used a hand-optimised assembler implementation of this kernel, which is described in detail in [2].

The parallelisation of the algorithm for multicore platforms follows a shared-memory approach and was implemented using OpenMP. All threads of the implementation run through the entire recursive scheme, and decide for each L1 block multiplication separately whether they need to execute it. Technically, this is achieved by the following implementational trick:

```
// start 'numThreads' threads, which
// all execute the given code block
#pragma omp parallel num_threads(numThreads)
{
   // store thread number in (syntactically
   // global) variable thread
   // 'thread' in thread local storage
   thread = omp_get_thread_num();
   // all threads: start block recursion:
   peanomult_PPP(...);
}
```

`peanomult_PPP` generates a cascade of recursive calls which all end in an L1 block mutiplication. The variable `thread` is then used to let only one thread execute a given L1 block operation. The required mapping of block operations to threads uses an *owner-computes* strategy. Each L1 block of the result matrix is assigned to one specific thread, which will execute all block multiplications that write to this block. The respective partitioning of the L1 blocks is illustrated in figure 3; based

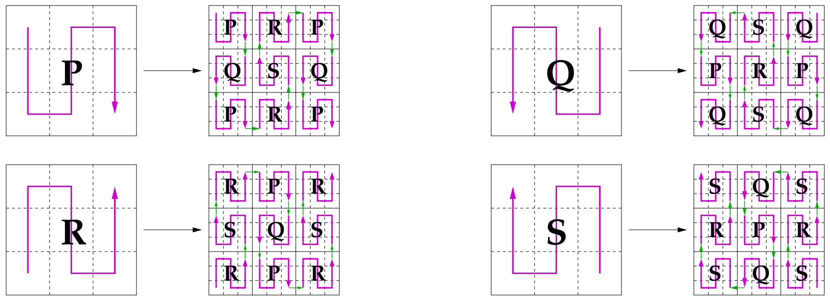

Figure 1: Recursive construction of the Peano numbering scheme; the patterns P, Q, R, and S define the numbering scheme for the matrix subblocks.

$$\begin{tikzcd}
P_0 \mathrel{+}= P_0P_0 \arrow[d] & R_5 \mathrel{+}= P_6R_3 \arrow[r] & R_5 \mathrel{+}= R_5S_4 \arrow[d] & P_6 \mathrel{+}= R_5Q_7 \arrow[r] & P_6 \mathrel{+}= P_6P_8 \arrow[d] \\
Q_1 \mathrel{+}= Q_1P_0 \arrow[d] & S_4 \mathrel{+}= Q_7R_3 \arrow[u] & S_4 \mathrel{+}= S_4S_4 \arrow[d] & Q_7 \mathrel{+}= S_4Q_7 \arrow[u] & Q_7 \mathrel{+}= Q_7P_8 \arrow[d] \\
P_2 \mathrel{+}= P_2P_0 \arrow[d] & R_3 \mathrel{+}= P_8R_3 \arrow[u] & R_3 \mathrel{+}= R_3S_4 \arrow[d] & P_8 \mathrel{+}= R_3Q_7 \arrow[u] & P_8 \mathrel{+}= P_8P_8 \\
P_2 \mathrel{+}= R_3Q_1 \arrow[d] & P_2 \mathrel{+}= P_8P_2 \arrow[u] & R_3 \mathrel{+}= P_2R_5 \arrow[d] & P_8 \mathrel{+}= P_2P_6 \arrow[u] & \\
Q_1 \mathrel{+}= S_4Q_1 \arrow[d] & Q_1 \mathrel{+}= Q_7P_2 \arrow[u] & S_4 \mathrel{+}= Q_1R_5 \arrow[d] & Q_7 \mathrel{+}= Q_1P_6 \arrow[u] & \\
P_0 \mathrel{+}= R_5Q_1 \arrow[r] & P_0 \mathrel{+}= P_6P_2 \arrow[u] & R_5 \mathrel{+}= P_0R_5 \arrow[r] & P_6 \mathrel{+}= P_0P_6 \arrow[u] &
\end{tikzcd}$$

Figure 2: Inherently local block multiplication scheme for the block matrix multiplication given in equation (1); the indices A, B, C have been left away for better readability.

on the element order given by the Peano space-filling curve, the blocks of the result matrix are distributed into equally sized partitions.

The owner-computes approach has two important advantages, which both result from the fact that only one thread will ever write to a specific L1 block: we do not require any time-consuming synchronisation of these write accesses; and, even more important, there will be no cache conflicts between threads as no cache synchronisation is required.

4. PERFORMANCE ANALYSIS

In this section, we compare the performance of TifaMMy (v. 1.3.2, available for download[10]) with that of the library functions `dgemm` provided by GotoBLAS (release 1.19) and Intel MKL (v. 10.0.1.014). For these performance comparisons (in sections 4.1 to 4.3), we used the following two machines with 8 and 16 cores, respectively:

- A 4-way (16 core) Intel Xeon Server with four Intel Xeon X7350 quad-core processors (Tigerton, 2.93 Ghz) with 64 GB FB-DIMM memory (667 MHz) connected via one front side bus per socket.
- A 2-way (8 core) AMD Opteron Server with two AMD Opteron 2347 processors (Barcelona, 1,9 GHz, TLB enabled) with 8 GB DDR2 memory (667 MHz) connected via AMD's NUMA technology.

They represent two different concepts of connecting CPU cores to memory. The Intel Xeon server uses a classical UMA architecture, where the processor communicates with main memory through the so-called front side bus using the main board chip sets. Hence, all sixteen CPU cores have to transport their data in parallel via the front side bus, which therefore forms a potential bottleneck. In contrast, AMD uses NUMA memory layout for their Opteron processors. Each CPU (socket) has its own private memory, which allows faster access. Access to "remote" memory regions requires additional inter-processor communication (Hypertransport 3.0).

In sections 4.4 and 4.5, we study the effect of the connection between CPUs and main memory. We used two machines with the same processor equipment – two Intel Xeon X5355 quad-core processors (Clovertown, 2,66 GHz) – but with different memory configuration: one *dual channel* workstation, where the 6 GB of FB-DIMM memory are connected to the memory controller hub via two parallel channels, and a *quad channel* server with 8 GB FB-DIMM connected via four channels, which doubles the memory bandwidth.

4.1 Performance on a 4-Way Xeon Platform (up to 16 Cores)

Figures 4 and 5 show the achieved MFlops/s rates on the 4-way Intel Xeon Server. TifaMMy proves to be faster than Intel's MKL from one up to 16 cores. MKL shows a particular performance deficit for 16 cores; there the execution times are not much faster than when using eight threads with TifaMMy. Compared to GotoBLAS, TifaMMy is slightly slower for the one and two thread tests; when using four threads on this system, the performances of TifaMMy and GotoBLAS are about equal. For 8 or 16 threads, TifaMMy is faster than GotoBLAS. Note that in figures 4 and 5 TifaMMy's performance was determined for the case where the matrix dimensions match the size of the L1 blocks (52×52) and no padding is required within the L1 blocks. The difference between best and worst case with respect to padding leads to slight performance differences of up to 100 MFlop/s per core (see also [2]).

In contrast, figure 6 shows the maximum and average performance *per core* for each library, which also shows how much of the theoretically available performance is achieved. The average (and maximum) was taken for matrix dimensions from 520 up to 5000 rows/columns (in steps of 10, including matrix sizes that require padding in TifaMMy). TifaMMy shows the smallest gap between average and maximum performance, which is also because TifaMMy reaches its peak performance already for smaller matrices.

As in the previous plots, we observe a distinctive drop of performance when all 16 cores are used. This might indicate a scalability problem of the underlying UMA architecture: all 16 cores have to transfer the accessed L1 blocks from main memory via the main board chip sets. As all libraries suffer from this performance drop in a similar way, the connection between chip set and memory seems to be the bottleneck in this test. As MKL already shows a noticeable decrease for 8 cores and because the decrease is a bit less for TifaMMy, this would also indicate slightly different characteristics of the memory access for the three codes.

4.2 Performance on a 2-Way Barcelona Platform (up to 8 Cores)

In figures 7 and 8 we compare the performance of TifaMMy, GotoBLAS and MKL on the AMD platform, which in contrast to the Intel machine uses a NUMA architecture. Figure 7 shows the measured MFlop/s rates for increasing matrix size using four and eight threads. Again, we use matrix dimensions that are exact multiples of the L1 block size (here 68×68, due to bigger L1 caches in the Barcelona chip) in figure 7, whereas figure 8 compares the average and maximum performance (including average- and worst-case matrix sizes) of the three codes running on 1, 2, 4, and 8 cores, respectively.

Interestingly, the MKL performance, relative to GotoBLAS and TifaMMy, is better on the Opteron than what we observed on the Xeon platform. The performance gap between MKL and the others is noticeably smaller, but MKL still scales worse than GotoBLAS and TifaMMy. The most eye-catching difference between TifaMMy and the other two libraries is that TifaMMy's performance for 8 threads is nearly uniform for growing matrix size, while both MKL and GotoBLAS show strong variations in performance: MKL shows a general zig-zag behaviour, which is also observed on the Xeon platform (though much more regularly) and might result from an underlying blocking strategy. In contrast, GotoBLAS shows certain plateaus of better performance, which are not observable on the Xeon platform. Hence, these might be an effect of the NUMA architecture.

4.3 Parallel Efficiency

In this paragraph, we compare the scalability of the three codes on the Xeon and the Barcelona platform. For that purpose, we compute the parallel efficiency $E(p) = F(p)/(p \cdot F(1))$, when using p threads, where $F(1)$ denotes the average (or maximum) performance in terms of MFlop/s measured for a single thread, p is the number of threads, and $F(p)$ is the average (or maximum) performance on p threads.

Figure 9 shows the computed parallel efficiency for the Barcelona (2 to 8 cores) and for the Xeon machine (2 to 16 cores). MKL shows the worst scalability with the largest difference between average and maximum performance. TifaMMy clearly shows the best scalability values on the Intel machine. For the Barcelona machine, GotoBLAS shows super-linear speedup for the best case, which is probably a result of the observed plateaus of higher performance.

4.4 Performance on Systems with Quad Channel vs. Dual Channel Memory

In this section, we compare TifaMMy's with GotoBLAS' performance on a dual and a quad channel memory system in order to identify dependencies on the connection between processor and memory. Figures 10 and 11 show the MFlop/s rates achieved by GotoBLAS and TifaMMy on the two 8-core Xeon machines.

In the GotoBLAS plots, we observe that the performance approaches a nearly uniform limit for the dual and the quad channel machine. It seems that GotoBLAS can not fully exploit the available higher memory bandwidth and is more limited by the memory latency, which is identical for the two machines. In contrast, the plots for TifaMMy (figure 11) show that the performance for the quad channel machine is better by a nearly constant value of 4000 MFlop/s. This indicates that TifaMMy is less affected by the latency, and can thus exploit the available higher bandwith in a better way. Note also that TifaMMy is as fast on the dual channel machine as GotoBLAS on the quad channel server.

4.5 Profiling using Hardware Performance Counters

To examine the memory access characteristics of the three multiplication codes in detail, we used Intel's VTune to measure certain hardware events, such as cache misses in the L1, L2, and DTLB (data translation lookaside buffer) cache, but also bus utilisation and stall cycles. Note that instead of the standard hit ratios provided by VTune, which relate the cache misses to the total number of instructions retired, we computed in table 1 the ratios between L1 misses (counter `L1D_CACHE_LD.I`) and total accesses to the L1 cache (counter `L1D_CACHE_LD.MESI`), which emphasizes the pure cache performance. In the same way, we computed the DTLB hit rate (`DTLB_MISSES.ANY` vs. `L1D_CACHE_LD.MESI`). To obtain the L2 hit rate, we related the number of L2 line fetches (`L2_LINES_IN.BOTH_CORES.DEMAND`) to the number of L2 requests (`L2_RQSTS.BOTH_CORES.DEMAND`) – in that way we disregard L2 misses caused by hardware prefetching. The measurements were started after a suitable time offset in order to measure the multiplication routines, only, i.e. to mask out the effect of any preprocessing steps, such as matrix initialisation or transposition. We used the 2-way Intel Xeon workstation with dual channel memory to compute the multiplication of square matrices of dimension 5200 on all eight cores. The results of the VTune measurements are given in table 1.

TifaMMy's cache oblivious approach proves to compete well with the cache aware approaches of GotoBLAS and MKL. TifaMMy has the highest L1 hit rate and, due to its owner computes strategy, a distinctively lower number of cache inconsistency events (`CMP_SNOOP.BOTH_CORES.INVALIDATE`). The rather low L1 hit rate measured for GotoBLAS can be explained by GotoBLAS' approach to stream the matrix A directly from the L2 cache into the registers[4]. We also observe that TifaMMy's blocked data structure and cache oblivious algorithm are sufficient to minimise DTLB misses,

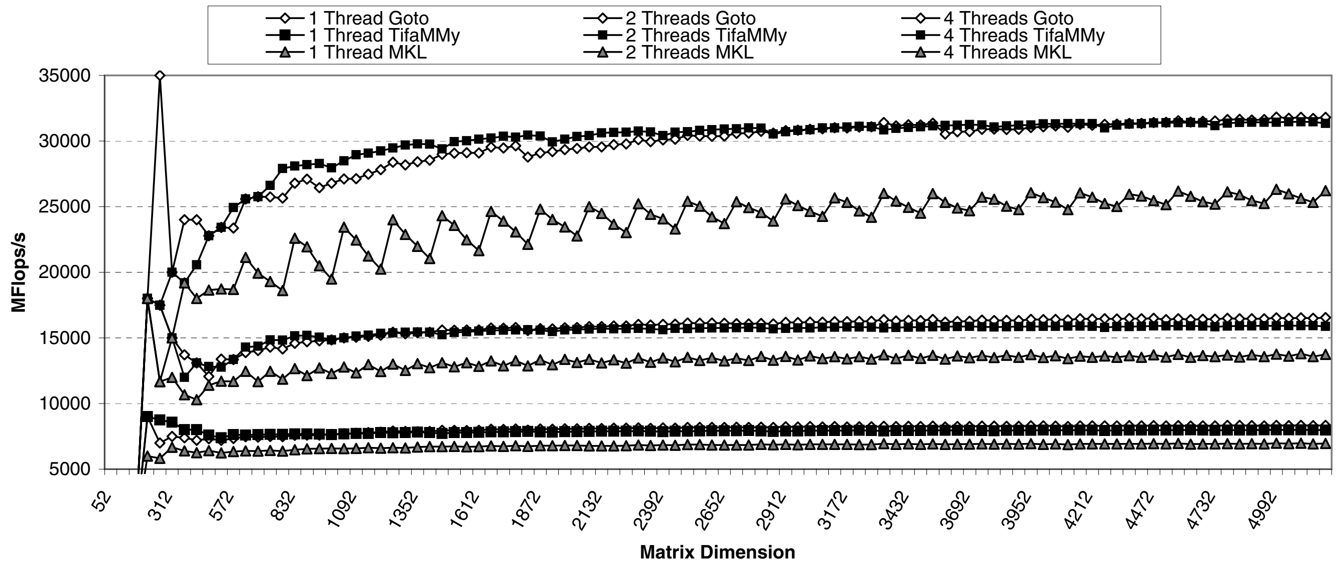

Figure 4: Performance of TifaMMy, GotoBLAS and MKL on the 4-way Xeon server (1 to 4 cores)

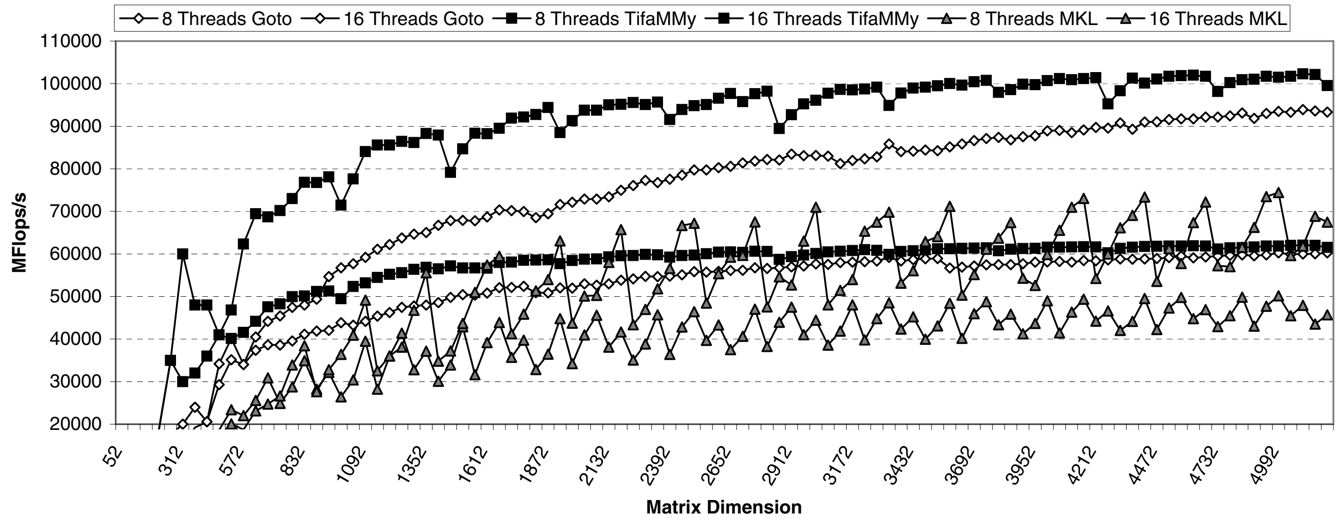

Figure 5: Performance of TifaMMy, GotoBLAS and MKL on the 4-way Xeon server (8 to 16 cores)

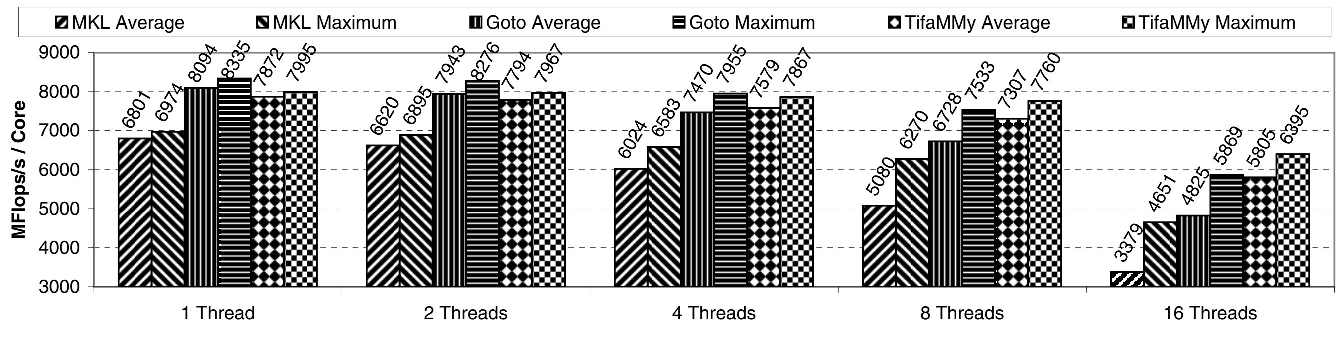

Figure 6: Relative performance of TifaMMy, GotoBLAS and MKL on the 4-way Xeon server (average and maximum values in MFlop/s *per core*)

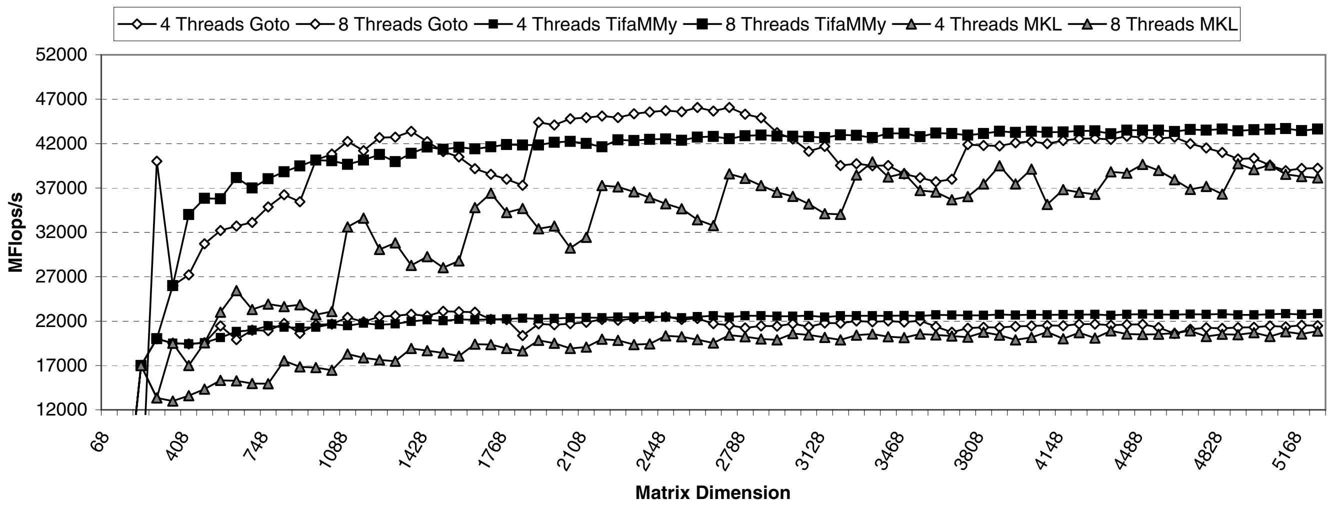

Figure 7: Performance of TifaMMy, GotoBLAS and MKL on the 2-way Opteron server (4 and 8 cores)

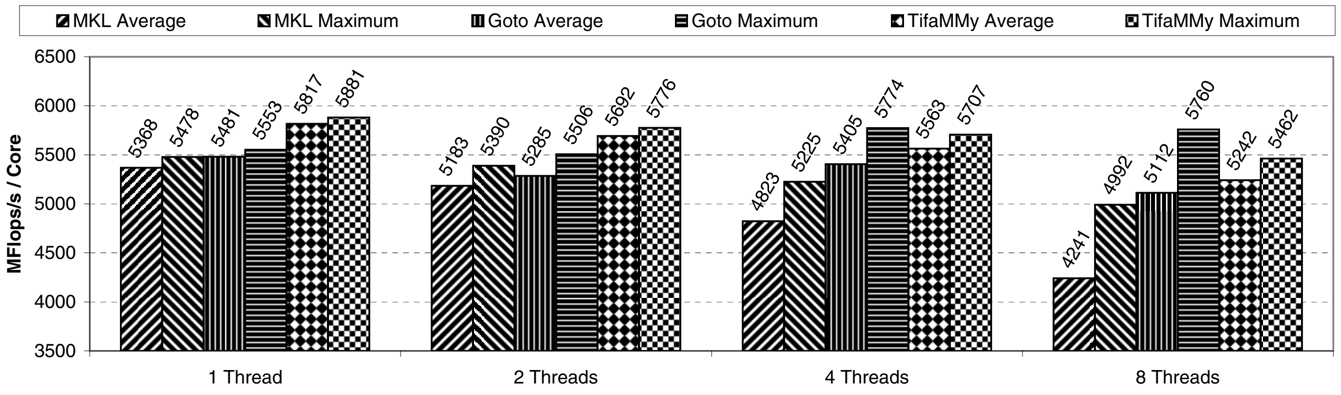

Figure 8: Relative performance of TifaMMy, GotoBLAS and MKL on the 2-way Opteron server (average and maximum values per core)

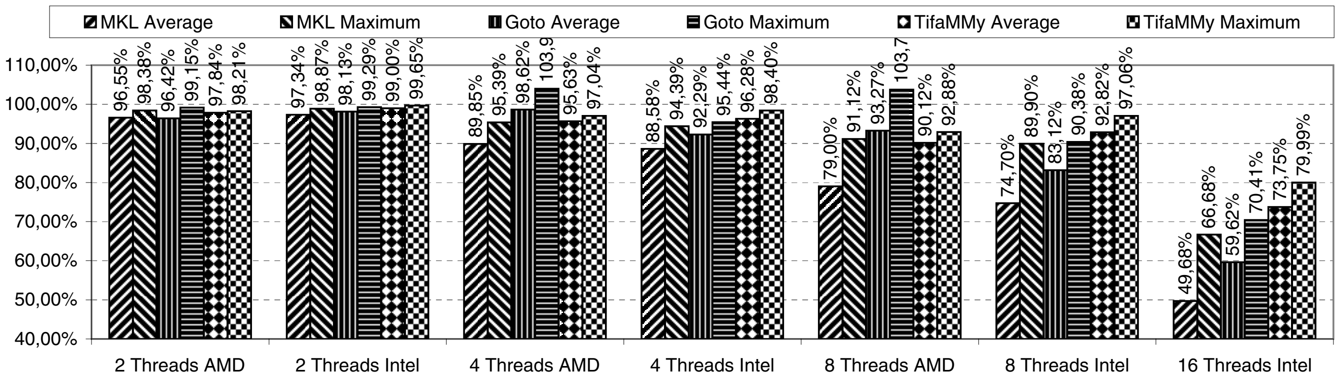

Figure 9: Parallel efficiency $E(p)$ for TifaMMy, GotoBLAS and MKL (computed for average and maximum performance, respectively)

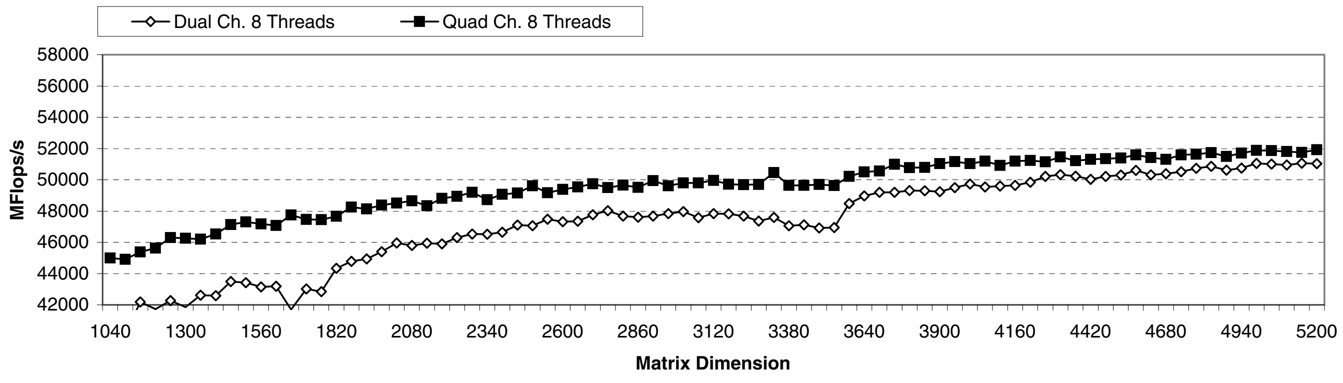

Figure 10: GotoBLAS running on Clovertown with dual & quad channel memory interface

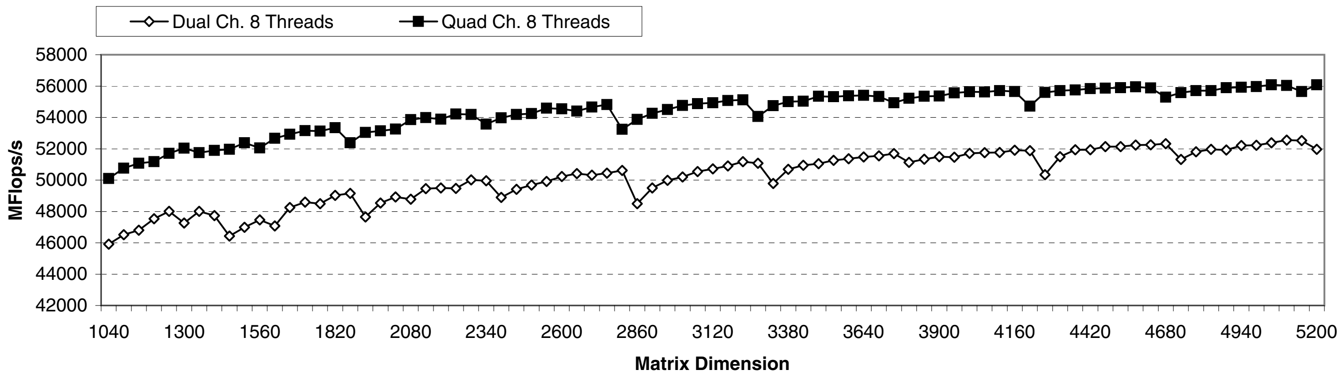

Figure 11: TifaMMy running on Clovertown with dual & quad channel memory interface

Ratios/Event	TifaMMy	GotoBLAS	MKL
L1 Hit Rate	98.74%	63.27%	92.49%
L2 Hit Rate	95.41%	99.69%	99.57%
L2 Lines demanded	$49.3 \cdot 10^6$	$30.2 \cdot 10^6$	$18.1 \cdot 10^6$
L2 Lines prefechted	$185.8 \cdot 10^6$	$97.4 \cdot 10^6$	$152.1 \cdot 10^6$
Cache invalid	$1.8 \cdot 10^6$	$13.4 \cdot 10^6$	$92.8 \cdot 10^6$
DTLB Hit Rate	99.99%	99.98%	99.99%
Data Bus Utilisation	12.74%	14.36%	15.16%
Stall cycles ratio	0.13	0.12	0.21
Clocks per Instruction	0.452	0.467	0.586

Table 1: Measured hardware events and cache hit rates on the 2-way Clovertown dual channel workstation

as well. Regarding L2 performance, it is somewhat contradictory that for TifaMMy we measure a larger number of L2 lines loaded from memory (both on demand and due to hardware prefetching), but on the other hand TifaMMy has the lowest data bus utilisation. Hence, considering all cache results, we have to assume that TifaMMy gains it scalability advantage over GotoBLAS from the better L1 hit rate and the much fewer cache inconsistencies. The overall lower performance of MKL shows up in the higher average number of clock cycles per retired instruction and in the larger ratio of stall cycles. The latter was measured as the ratio of wasted micro operations (`RS_UOPS_DISPATCHED.CYCLES_NONE`) vs. all unhalted CPU cycles (`CPU_CLK_UNHALTED.CORE`).

5. CONCLUSION

We demonstrated that a straightforward parallelisation of our cache oblivious algorithm for matrix multiplication based on Peano space-filling curves is a powerful alternative to today's extremely optimised hardware oriented implementations of BLAS routines. TifaMMy's cache oblivious approach is even able to outperform established libraries, such as GotoBLAS and MKL, on various multicore platforms, which is achieved without any further modifications of TifaMMy's memory access scheme. In order to tune TifaMMy for different hardware, only the comparably simple assembler kernel has to be optimised for the given processor. In particular, the size of the L1 blocks has to be chosen according to the size of the L1 cache. We demonstrated that a specific assembler kernel can be effective for a larger range of CPUs (here, Intel Xeon and AMD Opteron) offering similar instruction set implementations. Besides this point of hardware awareness, especially the parallelisation is straightforward and easy to expand for many-core systems (>16 cores). Hence, future work will include the evaluation of TifaMMy's performance on multicore-based parallel platforms ranging from workstations and servers up to HPC clusters. Especially for HPC clusters, the influence of the memory architecture (shared vs. distributed memory, etc.) will need to be studied, which will help to predict TifaMMy's performance for future manycore CPUs with 100 cores or more.

Acknowledgements

All multicore platforms used for measurements in this paper were provided by the chair for parallel computer architectures (LRR) at the TU München. We want to thank our colleagues at LRR, most of all Josef Weidendorfer and Michael Ott, not only for providing access to their machines, but especially for their valuable support and most helpful contributions in setting up the measurements.

6. REFERENCES

[1] Bader, M. and Zenger, C.: Cache oblivious matrix multiplication using an element ordering based on a Peano curve. Linear Algebra Appl. 417 (2–3), 2006.

[2] Bader, M., Franz, R., Günther, S., and Heinecke, A.: Hardware-oriented Implementation of Cache Oblivious Matrix Operations Based on Space-filling Curves, Proceedings of the PPAM 2007, LNCS 4967, 2008 (in print).

[3] Elmroth, E., Gustavson, F., Jonsson, I., and Kågström, B.: Recursive blocked algorithms and hybrid data structures for dense matrix library software. SIAM Review 46 (1), 2004.

[4] Goto, K., and van de Geijn, R.A.: Anatomy of a High-Performance Matrix Multiplication. Accepted for publication in ACM Transactions on Mathematical Software 34 (3), 2008, preprint: `http://www.cs.utexas.edu/users/flame/pubs/openflame.pdf`.

[5] GotoBLAS, Texas Advanced Computing Center, `http://www.tacc.utexas.edu/resources/software/`

[6] Gunnels, J.A., Gustavson, F.G., Pingali, K., Yotov, K.: Is cache-oblivious DGEMM viable? Proceedings of the PARA 2006, LNCS 4699, p. 919–928, 2007.

[7] Gustavson, F. G.: Recursion leads to automatic variable blocking for dense linear-algebra algorithms. IBM Journal of Research and Development 41 (6), 1997.

[8] Kurzak, J. and Dongarra, J.: Implementation of the Mixed-Precision High Performance LINPACK Benchmark on the Cell Processor. LAPACK Working Note 177, 2006.

[9] Intel math kernel library, 2007. `http://intel.com/cd/software/products/asmo-na/eng/perflib/mkl/`

[10] TifaMMy (TifaMMy isn't the fastest Matrix Multiplication, yet), `http://tifammy.sourceforge.net`, version 1.3.2.

[11] Whaley, R.C., Petitet, A., and Dongarra, J.J.: Automated empirical optimization of software and the ATLAS project, Parallel Computing 27 (1–2), 2001.