Fine Tuning Matrix Multiplications on Multicore

Stéphane Zuckerman, Marc Pérache, and William Jalby

LRC ITACA, University of Versailles and CEA/DAM
{stephane.zuckerman,william.jalby}@prism.uvsq.fr, marc.perache@cea.fr

Abstract. Multicore systems are becoming ubiquitous in scientific computing. As performance libraries are adapted to such systems, the difficulty to extract the best performance out of them is quite high. Indeed, performance libraries such as Intel's MKL, while performing very well on unicore architectures, see their behaviour degrade when used on multicore systems. Moreover, even multicore systems show wide differences among each other (presence of shared caches, memory bandwidth, etc.) We propose a systematic method to improve the parallel execution of matrix multiplication, through the study of the behavior of unicore DGEMM kernels in MKL, as well as various other criteria. We show that our fine-tuning can out-perform Intel's parallel DGEMM of MKL, with performance gains sometimes up to a factor of two.

Keywords: BLAS, multicore, cache coherency.

1 Introduction

Dense linear algebra, being the first of Berkeley's seven dwarfs [1], is an important part of the scientific programmer's toolbox. BLAS (Basic Linear Algebra Subroutines), and in particular its third level, DGEMM (double general matrix multiplication), are widely used, in particular within dense or banded solvers. It is then no surprise that decades have been spent studying and improving this particular set of subroutines. Over time, theoretical complexity has been improved, while at the same time architecture-conscious algorithms for both sequential and parallel computations have emerged (cf for example Cannon's algorithm [2], Fox's algorithms [4], or more recently SRUMMA [7] and [3]).

There are some reservation to be asserted, however. First, numerous papers focused on the square matrix multiplication case, and not the truly general one. This is particularly damaging because for example the block version of the LU decomposition relies heavily on rank-k updates which are products of an $(N \times k)$ matrix by a $(k \times N)$ matrix with k, typically between 10 and 100, being much smaller than N (typically several thousands); [9] studies this matter extensively. Unfortunately, dealing with these rectangular matrices requires specific strategies fairly different from the standard, easier, square case.

Second, most of the algorithms proposed have a fairly high level view of the target architecture and their underlying model is much too coarse to get the best performance – in terms of gigaflops – of the recent architectures. More precisely, most of the practical algorithms relies on matrix blocking and spreading the block computations across the processors. However fine tuning (choosing the right block size) is still mandatory to get peak performance. This fine tuning process is fairly complex because many constraints have to be simultaneously taken into account: uniprocessor/core performance, including both ILP and locality optimization, has to be optimized, coherency traffic/data exchange between cores has to be minimized and finally the overhead of scheduling the block computations must remain low. In particular, a systematic methodology has to be developed to take into account all of these factors which might have major impact on overall performance. It should be noted that the simpler case of optimizing unicore performance of a matrix multiplication requires a fairly complex methodology (relying on experimental architectural characterization, cf. ATLAS[10]) to reach good performance.

In this paper, we try to develop a parallelization strategy for taking into account all of the architectural constraints of recent multicore architectures. Our contributions are twofold. First we experimentally analyze in detail all of the key factors impacting performance on two rather different multicore architectures (Itanium Montecito and Woodcrest). Second, summarizing our experimental study, we propose a parallelization strategy and shows its efficiency with respect to the well known MKL libraries.

This paper is structured as follows: section 2 describes a motivating example showing the difficulty in selecting the right block sizes, as well as our experimental framework. Section 3 presents experimental analysis of various blocking strategies. Section 4 presents our parallelization methodology and compares the resulting codes with Intel's parallel implementation of MKL.

2 Motivating Example

Experimental Setup

All the experiments shown in this paper have been carried out on the following architectures:

- A dual-socket Xeon Woodcrest (5130) board with dual-core processors, 2 GHz CPU (32 GFLOPS 4 cores peak performance), and 533 MHz FSB (i.e. $\approx 8.6\,GB/s$). Each dual-core processor has a 4 MB L2-cache shared by two cores. This machine will be denoted by "x86" in the remaining of this paper.
- A 4-way SMP node, equipped with dual-core Itanium 2 Montecito processors (with HyperThreading Technology deactivated[1]), with 1.6 GHz CPU (51.2 GFLOPS 8 cores peak performance). Each core has a private 12 MB L3-cache, 256 kB L2-cache and 667 MHz FSB (i.e., $\approx 10.6\,GB/s$). This machine will be denoted as "ia64" in the remaining of this paper.

[1] HTT is mainly useful when dealing with I/O-bound programs, much less with compute-bound ones. Limited testing showed no improvement by using HTT in our computations, while increasing risks of cache-thrashing.

ICC v10.0 and MKL v10.0 were used to make our benchmarks. Two versions of MKL were used: MKL Parallel denotes the orginal parallel version provided by Intel, MKL Unicore (or Sequential) refers to the MKL specially tuned for unicore/sequential use. The operating system is Linux in both cases, with a 2.6.18 kernel.

It should be noted that the MKL Sequential was used as a “black box”. It is a very high performance library, on both x86 and IA64 architectures. It shows extremely good results on monocore systems. Thus, aside from the parallel strategy we describe in section 4, a fair amount of tiling, copying and so forth is being performed by the MKL sequential functions. For the remaining of the paper, we will compare our parallelized version of DGEMM based on Sequential MKL kernels with MKL Parallel.

All of the arrays are stored following the “row major” organization, as we used C for our programs. Although the original BLAS library is implemented in FORTRAN, all matrices are stored in a unidimensional array. Experiments show no significant differences between row- and column-major storage strategies in the MKL/BLAS library.

Instead of using OpenMP or directly POSIX threads, we used a performant M:N threading library, Microthread, which was developed internally, and served as a basis for MPC’s [8] OpenMP runtime. It relies on a fork-join approach as OpenMP does, but allows for more flexibility – for example by permitting us to chose to which processor we want to assign a given sub-DGEMM, while reducing thread handling complexity inherent to classic POSIX threads. Moreover, thread creation and destruction overheads are kept minimal. However, in terms of performance, the gain offered by Microthread over a solution based on OpenMP remains limited: between 5 and 10% when block computations are small and less than 5% when blocks are large. However, on truly small kernels, where the amount of data makes it difficult to find enough ILP per core, the overhead of Microthread becomes too large (just like any OpenMP runtime). Of course, this is a case where parallelizing a task might prove less beneficial than running a sequential job.

Notations/General Principles of Our Parallelization
For the remainder of this paper, we will look at the simplest form of DGEMM, which performs the following task : $C_{N_1,N_3} = A_{N_1,N_2} \times B_{N_2,N_3}$. We denote NB_i the number of blocks resulting from the partitionning of *i-th* dimension.

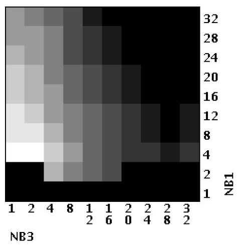
Our parallelization strategy relies on a standard decomposition of the three matrices in blocks ([4, 7]). All of the block computation on a unicore are performed using the MKL library which achieves very good performance on a unicore when the blocks fit in the cache. It should be noted that the blocks resulting from our decomposition are not necessarily square (they can have arbitrary rectangular shape) and second our parallelization strategy is not limited to having a number of block computation exactly equal to the number of available cores. We allow to have much more block computations than cores, i.e. overloading of cores is used.

A Simple Performance Test

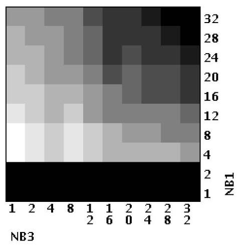
Figure 1 describes performance variations of various partitionning strategies for a simple parallel $C_{N,N} = A_{N,10} \times B_{10,N}$: the X axis (resp. Y axis) refers to the number of blocks (NB_3) along the third dimension (resp. the first dimension (NB_1)). Instead of showing absolute performance numbers, relative performance (with respect to the best performance) is displayed: the whiter areas corresponds to best partitionning strategies (i.e. white means between 95 % and 100 % of the best performance), while the darker areas identify poor choices of partitionning parameters.

In the upper three plots (1(a), 1(b), 1(c)) displayed, the size of the matrices are such that they entirely fit in the L2 (resp. L3) cache of the x86 (resp. ia64). Now for these three cases, the white area is much narrower: only one or two partitionning strategies achieve top performance.

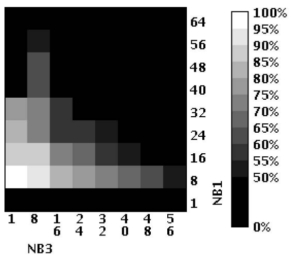
In the lower three plots (1(d), 1(e), 1(f)) displayed, the size of the matrices exceed the L2 (resp. L3) cache size of the x86 (resp. ia64). For these three cases, the white areas are fairly large, meaning that many partitionning strategies allow to reach close to the best performance. Now which is much more difficult to predict is the shape of the white area and why the shapes are so different



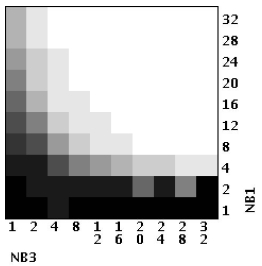
(a) x86 2 × 2-core $A_{500,10} \times B_{10,500}$



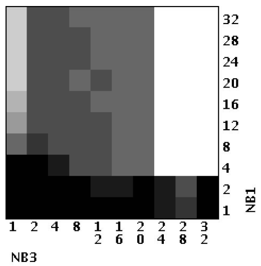
(b) ia64 2 × 2-core $A_{1000,10} \times B_{10,1000}$



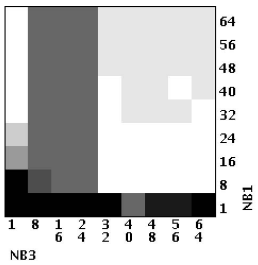
(c) ia64 4×2-cores $A_{1000,10} \times B_{10,1000}$



(d) x86 2 × 2-core $A_{2000,10} \times B_{10,2000}$



(e) ia64 2 × 2-core $A_{3000,10} \times B_{10,3000}$



(f) ia64 4 × 2-core $A_{4000,10} \times B_{10,4000}$

Fig. 1. Figure 1(a) (resp. Fig. 1(b), 1(c)): the size of the matrices is such that they fit entirely within the L2 (resp. L3) cache of the x86 (resp. ia64). For figures fig. 1(d) (resp. Fig. 1(e), 1(f)), the size of the matrices is such that they exceed the L2 (resp. L3) cache of the x86 (resp. ia64).

between x86 and ia64. Furthermore, it is a bit surprising that the NB_1 and NB_3 parameters do not have a similar effect on the ia64.

Our Approach
Our goal is to develop a strategy allowing to identify quickly what are good choices for the block values NB_1, NB_2 and NB_3. By "good" we mean within 10% of the best performance achievable when varying arbitrarily block sizes.

To achieve that goal, 3 subproblems have to be carefully taken into account:

1. The block computation running on a unicore must be close to top speed. If the block is too small, there is not enough ILP to get the best performance of the unicore, loop overhead becomes the main reason for slowdowns. If the block exceeds the L2/L3 cache size, the blocking method used by MKL might not be adequate.
2. The number of blocks must be carefully chosen first to achieve a good load balancing and second to keep a low parallelization overhead.
3. The scheduling of block computations to different cores might induce coherency traffic between the cores. For example if a row of C is spread across several cores, each core will write part of the row, some cache lines being shared between two cores (cf Section 3.1).

Finally, it is important to note that we are aiming at the best 10 % as far as performance is concerned, which is symbolized by white or light-grey colors on all our figures.

3 DGEMM Performance Analysis

3.1 Limiting Cache Coherency Traffic

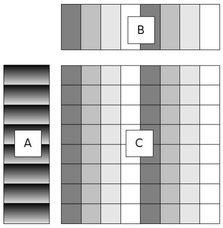
The amount of coherency traffic will depend how blocks are allocated to different cores. We will use two opposite strategies: Write Columns versus Write Rows.

In the Write Columns scheme, every core is computing and writing into different sets of columns of the result matrix C. In this scheme, the A matrix will be read by all cores while each core will read different sets of B columns. Since C is stored row-wise, some cachelines (containing C values) can be shared by different cores leading to coherency traffic. The resulting performance, depending on various blocking strategies are shown in figure 2(b) and 2(c).

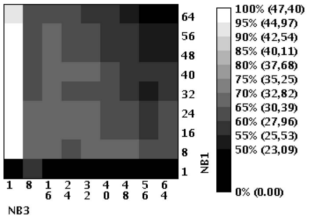
In the Write Rows strategy, every core is computing and writing into different sets of rows of the result C matrix. In this scheme, the B matrix will be read by all cores while each will read different sets of rows of the A matrix. In this case, very few cachelines of C are shared between different cores. The resulting performance, depending on various blocking strategies are shown in figure 3(b) and 3(c).

The two strategies are illustrated in figure 2(a) and 3(a).

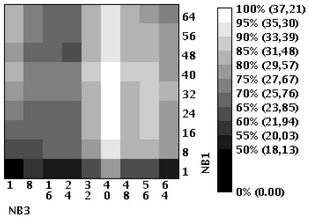
Although best performance between the two strategies is comparable, one (the Write Columns one) produces a much narrower area of good values for the good block values. On the other hand, the Write Rows strategy gives us an



(a) Write Columns allocation strategy

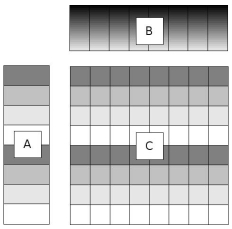


(b) MC 4×2 cores $A_{1000,100} \times B_{100,1000}$

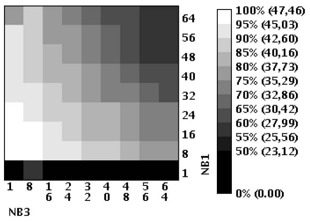


(c) MC 4×2 cores $A_{4000,100} \times B_{100,4000}$

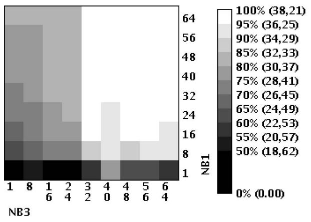
Fig. 2. $A_{N,k} \times B_{k,N}$ blocking with a Write Columns strategy



(a) Write Rows allocation strategy



(b) MC 4×2 cores $A_{1000,100} \times B_{100,1000}$



(c) MC 4×2 cores $A_{4000,100} \times B_{100,4000}$

Fig. 3. $A_{N,k} \times B_{k,N}$ blocking with a Write Rows strategy

advantage: the "good" areas encompass the ones in the Write Columns strategy, but are much larger, hence allowing for a bigger blocking factor without hurting performance.

This behavior is clearly due to false-sharing of cachelines: when using the Write Columns strategy, one creates many "frontiers" where a set of cache lines may be shared between two cores. By ensuring that a single core writes for the longest possible time in a same set of rows in C, we reduce these "frontiers" to a minimum. This works because we are in a row-major setup; the strategy would give inverse results in a column-major one.

3.2 DGEMM Analysis

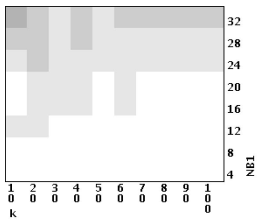
In this section, we will study three extreme cases of matrix multiplication of rectangular matrices, which allows us to uncover most of the key problems in matrix multiplication parallelization. A large set of experiments were carried out. Only the most impressive ones are shown and analyzed. Moreover, we observed a continuous performance behavior when varying parameters such as for example k (where k is the number of columns of A). More precisely, a performance behavior for $k = 20$ can be easily interpolated from the behavior of $k = 10$ and $k = 30$.

Performance counters were not used for this parallel analysis, because efficient tools that give correct and fine measurements in a multicore environment are almost inexistant. You can find good sequential measurement tools such as Perfmon or Intel VTune. Of course, the sequential behavior of a given kernel can help to fine-tune its parallel counterpart (for example, a kernel that already saturates the main memory bandwidth is going to be trouble in parallel). But nothing can be said about cache coherency, and additional bus contention due to several cores trying to write to main memory, for example. However, we do use performance counters while evaluating unicore performance (cf. section 4).

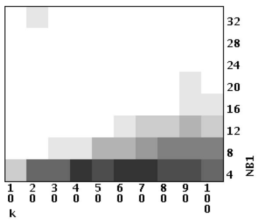
Both the $A_{N,k} \times B_{k,k}$ and $A_{k,k} \times B_{k,N}$ kernels (studied below) behave well in a sequential, unicore environment: performance counters tell us that there is no bandwidth shortage, nor real performance issues.

3.3 Performance Analysis of $C_{N,k} = A_{N,k} \times B_{k,k}$(Fig. 4)

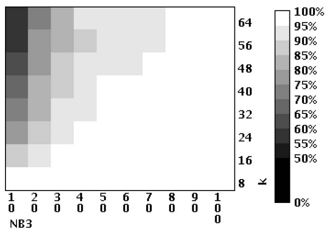
Since k is small, the only opportunity for parallelization lies in partitioning along the first dimension. Each core has its own copy of B, and only relevant rows of A are read. Moreover, writing to C is done row-wise, which prevents most false-sharing from occurring. In figure 4(a) (x86 4 cores) the three matrices fit within the cache and minimizing the partitioning on A i.e. $NB_1 = 4$ or 8 is a fairly good strategy. On the other hand, in fig. 4(b), when we exceed the cache size, larger partitioning degrees of A are required. In figure 4(c) where the three arrays fit again in cache, a minimum number of blocks of A is a very good strategy.



(a) x86 2 × 2-core $A_{2000,k} \times B_{k,k}$



(b) x86 2 × 2-core $A_{6000,k} \times B_{k,k}$

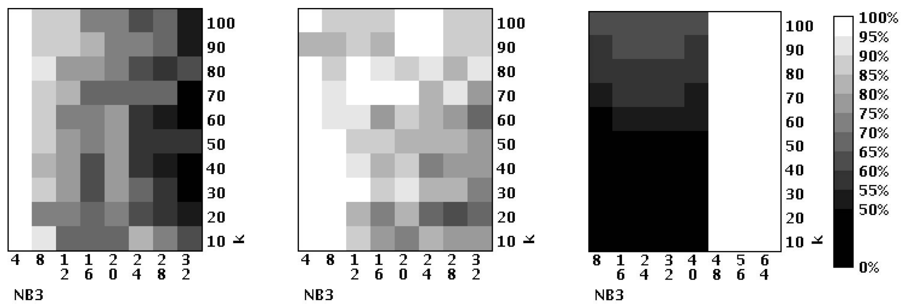


(c) ia64 4×2-core $A_{6000,k} \times B_{k,k}$

Fig. 4. Figures 4(a), 4(b) (resp. 4(c)) present performance variations of the primitive $C_{N,k} = A_{N,k} \times B_{k,k}$ on x86 (resp. ia64). The Y axis refers to the number of horizontal blocks used for partitioning A and B, while the X axis refers to different values of k. For each value of k, performance numbers have been normalized with respect to the best performance number obtained for this value of k. For Figure 4(a) (resp. Fig 4(c)), the size of the matrices is such that they fit entirely within the L2 (resp. L3) cache of the x86 (resp. ia64) while for Fig 4(b), the size of the matrices exceed the L2 x86 cache size.

3.4 Performance Analysis of $C_{k,N} = A_{k,k} \times B_{k,N}$(Fig. 5)

This is the symmetrical counterpart of the previous case. In theory, it should behave exactly the same way, but in practice, there is a huge performance gap. Several factors explain this. The first one is that the performance behaviour of the



(a) x86 2×2-core $A_{k,k} \times B_{k,2000}$ (b) x86 2×2-core $A_{k,k} \times B_{k,6000}$ (c) ia64 4×2-core $A_{k,k} \times B_{k,6000}$

Fig. 5. $C_{k,N} = A_{k,k} \times B_{k,N}$ DGEMMs on x86 (fig. 5(a),5(b)) and ia64 (fig. 5(b)) architectures. Data sets fit in the x86 cache (fig. 5(a)) while data sets in fig. 5(b) exceed its cache size. Fig. 5(c) presents results on ia64 with a data set fitting in L3.

unicore block MKL kernel $B_{k,k} \times C_{k,N}$ is fairly different from the performance of the unicore block kernel $(B_{N,k} \times C_{k,k})$. Second, generating blocks means dividing in a column-wise manner, which is prone to provoke false-sharing.

3.5 Performance Analysis of $C_{N,N} = A_{N,k} \times B_{k,N}$ (Fig. 1)

Here we have a combination between the two previous cases, rendering performance prediction difficult at best. However, there is a clear trend to see: when the sub-matrices fit in cache, there is only one good partitionning strategy, i.e. dividing according to the number of cores. On the contrary, for matrices larger than cache size (see figures 1(d), 1(e) and 1(f)) higher degrees of partitionning are required.

3.6 A Quick Summary of These Experiments

First, basic block performance is essential. Second, as long as we are performing DGEMMs where (sub-)matrices fit in L2 or L3 cache, there is no need to go further than divide the work according to the number of cores available. However, as soon as we are on the verge of getting out of cache, it is important to increase the blocking degree so as to fit in cache once again, with a good sequential computation kernel. So far, all our experiments have shown that this in-cache/out-of-cache strategy (see next section) is sufficient to get good results.

4 A Strategy to Fine-Tune Matrix Multiplication

Methodology for Fine-Tuning DGEMM Parallelization
The major difficulty in the parallelization strategy is in fact the right choice of block sizes (i.e. partitioning of the matrices). Let us first introduce a few

notations. Our focus is the parallelization of the computation of $C_{N_1,N_3} = A_{N_1,N_2} \times B_{N_2,N_3}$. The number of blocks along the first dimension N_1 (resp. N_2, N_3) will be denoted NB_1 (resp. NB_2, NB_3). The corresponding block sizes will be denoted B_1, B_2, B_3, in fact $B_i = N_i/NB_i$, $i \in \{1, 2, 3\}$.

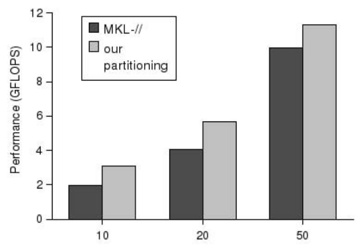
The first step of the method consists in first benchmarking unicore performance of the basic blocks multiplication. This will give us constraints on the block sizes of the form $B_1^{min} < B_1 < B_1^{max}$ (and the similar ones for B_2 and B_3), meaning that if B_1 satisfies such inequalities, we are within 10 % of the peak performance of a unicore matrix multiply. This step requires systematic benchmarking and integrates most of the particularities of the underlying unicore architecture and of the library used for unicore computations. This step is done *once for all* for a given unicore architecture. The results are stored in a database and used in a later step of our strategy. It should be noted that not only GFLOPS performance numbers are stored in this database but also bandwidth consumption between the various cache levels (this is obtained by measuring cache misses using hardware counters).

The second step consists in exhaustively searching all of the partitionings such that:

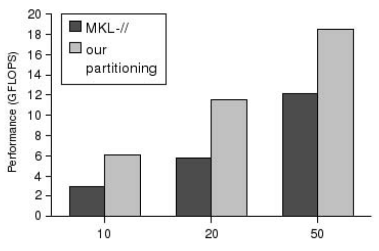
1. The resulting block sizes satisfy the unicore good performance constraints
2. The sum of the sizes of the three blocks (corresponding to an elementary block computation) is less than a quarter of the last level cache size $B_1B_2 + B_1B_3 + B_2B_3 < CS/4$. Aiming at using only a quarter of the available cache size, allows us to be on the safe side (i.e. being sure that the three blocks remain in cache) and second results still in good cache miss ratio due to the large size of L2 and L3 caches
3. The quantity $NB_1 \times NB_2 \times NB_3$ is a multiple of the number of cores (to insure perfect load balancing). If $NB_1 \times NB_2 \times NB_3$ is less than the number of available cores, the number of cores used is reduced accordingly to still match the load balancing constraint

Then in third step, all of the solutions are lexicographically sorted according to the values of NB_1, NB_2 NB_3. This sort aims at taking into account the fact that from the parallelization point of view the three dimensions are far from being equivalent:

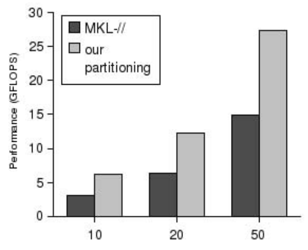
– partitioning along the first dimension induces a simple parallel construct (DOALL type) with minimal overhead and the induced partitioning on matrix C is row-wise and does not induce false-sharing
– partitioning along the third dimension induces also a simple parallel construct with minimal overhead but the induced partitioning on the C matrix is column-wise and will generate false-sharing of cache lines
– partitioning along the second dimension is more complex because it requires synchronization to accumulate the results. In our parallelization strategy, we chose to perform the block computations in parallel, each core accumulating in a different temporary array. Once all of the blocks have been computed, a single core sums up all of the temporary arrays into the final C block.



(a) x86 2 × 2-core $A_{2000,k} \times B_{k,2000}$



(b) ia64 2 × 2-core $A_{3000,k} \times B_{k,3000}$



(c) ia64 4×2-core $A_{4000,k} \times B_{k,4000}$

Fig. 6. Intel's parallel MKL/DGEMM versus our own parallelization

Therefore, the final solution picked up is the one corresponding with the minimum NB_2 value, then the minimum NB_3 value; this corresponds to a lexicographic sort of the solutions. However, in order to minimize cache thrashing, it is important that each thread is given "contiguous" blocks: for each block of lines in A, a given thread which has not reached its maximum number of tasks is given a certain amount of "contiguous" blocks in B.

Very convincing results were obtained using our parallelization strategy (see fig. 6). The most impressive ones relate to the $C_{N,N} = A_{N,k} \times B_{k,N}$ case, where operands do not fit in cache. This is due to the fact that MKL uses a constant strategy of minimizing the number of blocks used (the number of blocks MKL uses is exactly equal to the number of cores). When operands fit in cache, this strategy works fairly well (except in $C_{k,N} = A_{k,k} \times B_{k,N}$) but performs poorly when operands do no longer fit in cache. Although these experiments show how much gain can be obtained with a good parallel strategy, the results are far from reaching peak performance. On the $C_{N,N} = A_{N,k} \times B_{k,N}$ case, there are almost ten times more memory writes than memory reads – i.e., even though there is enough ILP to exploit per core here, writing the results back to memory is tried all at once by all the cores, hence saturating the memory bus.

Comparison with Related Work

ATLAS. ATLAS [10] is a powerful "auto-tuned" library, i.e. upon installation, it performs various measurements (such as determining cache latencies and throughput) in order to choose the best computation kernel adapted to the underlying system. These kernels are either already supplied by expert programmers for a given architecture, or code generated when the underlying system is unknown. ATLAS relies mainly on a good blocking strategy which mixes hand-tuned kernels as well as automatically-generated code at install-time to produce a highly optimized BLAS library. It can also be built into multithreaded library. However, first the number of cores thus supported is fixed, and can never be increased at run-time: one must recompile the whole library each time the number of cores change. Second, ATLAS cannot take easily advantage of already existing DGEMM libraries: it requires very specific kernels.

GotoBLAS. On the opposite side, the GotoBLAS [5, 6] provide a highly hand-tuned BLAS library, with computation kernels programmed directly in assembly language, and very efficient sequential performance as a result. However, these kernels work only on very specific systems (those for which the kernels exist), and do not exactly respect the BLAS semantics (contrary to ATLAS and Intel MKL). Thus, although the changes to one's code are minimal, one can not simply "swap" BLAS libraries with GotoBLAS.

Our approach. It differs from ATLAS and GotoBLAS in different ways. ATLAS and GotoBLAS are above all a work to take advantage of sequential performance. They provide hand-tuned and automatically-tuned BLAS libraries, with an emphasis on blocking. Our approach aims parallel performance only, relying on good sequential BLAS routines. More precisely, our blocking strategy focuses only on parallel performance, with parallel criteria in mind, i.e. sequential ones, as well as memory contention, false-sharing risks, etc. We could take the kernels provided by ATLAS or (with some code modifications) GotoBLAS. Although ATLAS does provides a way to get multi-thread BLAS, this number must be fixed at compile-time, while our method scales with the number of cores.

5 Conclusion

Although matrix multiplication seems to be a solved problem at first, it is clear that in the parallel case and for shared memory systems, a large amount of work remains to be done to get peak performance. It is not enough to use a good and efficient unicore library. Special care has to be taken to take into account behavior of such libraries which are far from being uniform when varying matrix sizes. To get the best out of the MKL in our case, it was necessary to make various trade-offs between data locality, false-sharing avoidance, load-balancing, sequential kernel selection (to get the best sub-DGEMMs cases when distributing tasks) and memory bus contention. This has enabled us to get as much as twice the performance offered by the MKL parallelized by Intel in the best case, in a systematic manner. The methodology we propose is fairly systematic and can be easily automated. However it should be noted that for some specific (small) matrix sizes, the performance obtained is far from peak, due probably to a lack of performance of a unicore version. Further work include improving such cases by generating better unicore kernels then developing a fully automated version of the library and dealing with ccNUMA aspects for larger multicore systems.

References

[1] Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The landscape of parallel computing research: A view from berkeley. Technical report, EECS Department, Univ. of California, Berkeley (December 2006)

[2] Cannon, L.E.: A cellular computer to implement the kalman filter algorithm. Ph.D thesis (1969)

[3] Chan, E., Quintana-Orti, E.S., Quintana-Orti, G., van de Geijn, R.: Supermatrix out-of-order scheduling of matrix operations for smp and multi-core architectures. In: SPAA 2007: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures, pp. 116–125. ACM, New York (2007)

[4] Fox, G.C., Furmanski, W., Walker, D.W.: Optimal matrix algorithms on homogeneous hypercubes. In: Proceedings of the 3rd conference on Hypercube concurrent computers and applications. ACM, New York (1988)

[5] Goto, K., van de Geijn, R.: High performance implementation of the level-3. Transactions on Mathematical Software 35(1) (2008)

[6] Goto, K., van de Geijn, R.A.: Anatomy of a high-performance matrix multiplication. Transactions on Mathematical Software 34(3) (2008)

[7] Krishnan, M., Nieplocha, J.: Srumma: A matrix multiplication algorithm suitable for clusters and scalable shared memory systems. In: IPDPS (2004)

[8] Marc Pérache, H.J., Namyst, R.: Mpc: a unified parallel runtime for clusters of numa machines. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 78–88. Springer, Heidelberg (2008)

[9] Matthias Christen, O.S., Burkhart, H.: Graphical processing units as co-processors for hardware-oriented numerical solvers. In: Workshop PARS 2007 (2006)

[10] Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimizations of software and the ATLAS project. In: Parallel Computing (2001)