

HIGH-PERFORMANCE MATRIX MULTIPLICATION ON THE IBM SP HIGH NODE

André Henriksson
andropov@cs.umu.se

Isak Jonsson
isak@cs.umu.se

Abstract

The computing performance of processors in high-performance computers is increasing steadily. The overall memory bandwidth has not grown at the same rate. Instead, the memory hierarchies have got more complex, with more number of caches. Programs which need to utilize the full power of the processors have to adjust their data reference patterns to fit the memory models. In this paper, we show a way of organizing algorithms and corresponding data structures for linear algebra routines which enables automatic tuning for an arbitrary number of caches by using recursive technologies. We show how performance for matrix multiplication is increased by 51 % compared to existing routines for the IBM PowerPC 604 by using fine tuned kernels, algorithmic prefetching, recursive algorithms and data structures. We also present an algorithm for scheduling matrix multiplication on an SMP-node. Discussions on how kernels should be implemented and a cache simulation model are also included.

Keywords: algorithmic prefetching, batch tree, linear algebra kernel, memory hierarchy, recursive algorithm, recursive data structure, SMP-node, threads.

MASTER'S THESIS, 1998
DEPARTMENT OF COMPUTING SCIENCE
UMEÅ UNIVERSITY
SWEDEN

Contents

1	Introduction	1
1.1	Objectives	1
1.2	Method	1
1.3	Outline	1
2	Concepts and background	3
2.1	Theoretical performance	3
2.2	Matrix multiplication	4
2.3	Matrix blocking	6
2.4	BLAS	7
2.5	LAPACK	8
2.6	ScaLAPACK	8
2.7	ESSL	8
3	Hardware	11
3.1	Memory hierarchy	11
3.2	Cache	11
3.3	Main Memory	12
3.4	Disk	12
3.5	Translation lookaside buffer	12
3.6	What is an SMP-node?	12
3.7	The IBM SMP-node	13
4	Threads and SMP	15
4.1	Introduction to threads	15
4.2	What is a thread?	15
4.3	Threads and signals	16
4.4	Distribution of threads on SMP-nodes	16
4.5	Thread safety	17
5	Previous Work	19
5.1	Superscalar GEMM-based level 3 BLAS	19
5.2	ATLAS	20
5.3	PHiPAC	20
6	Optimal blocking	23
6.1	A , B , and C in cache	23
6.2	A and B in cache	25
7	Data Structures	29
7.1	Traditional mappings	29
7.2	Blocked formats	31
7.3	Multilevel blocks	32
7.4	Arbitrary ordered blocked matrix format	34
7.5	Recursive blocking and multiplication	35

8	Kernel routines	39
8.1	Peak performance	39
8.2	Register blocking	40
8.3	Instruction scheduling	43
8.4	Prefetching iterations	45
8.5	Prefetching matrices	46
9	Cache simulation tool	53
9.1	Cache model	53
9.2	Simulation and visualization tool	54
9.3	Some experimental results	58
10	Fast dynamic load balancing	63
10.1	Batch tree	63
10.2	Thread distribution	63
11	Testing considerations	67
11.1	Number of tests to perform	67
11.2	The looptime program	67
12	Results	69
12.1	Memory measurements	69
12.2	TLB measurements	69
12.3	Matrix multiplication	69
13	Conclusions and comments	77
13.1	Conclusions from the results	77
13.2	Future work	77
13.3	Acknowledgments	78
	References	79

List of Figures

1	Matrix multiplication using dot product	5
2	Matrix multiplication using <code>_AXPY</code>	5
3	Matrix multiplication using outer product	6
4	Memory Hierarchy	11
5	Types of SMP-nodes, shared memory and a distributed memory system.)	13
6	An overview of a process and its user-level threads. Everything in the memory area is shared, except for the stack.	16
7	Cycles per double floating point operation (addition and multiplication) at different M_B, N_B for $t_{fma} = 1, t_{setup} = 2, t_{load} = 5, csz = 2048$	24
8	Cycles per double floating point operation at $M_B = N_B$ for the example $t_{fma} = 1, t_{setup} = 2, t_{load} = 5, csz = 2048$	25

9	Schematic view of a 2048 double words cache, in which A , the light-gray block, and B , the dark-gray block, resides. A and B have $24^2 = 576$ elements each. Since they both start on the same address modulo 512 double words, the cache sets for addresses less than 64 modulo 512 double words are full, and all of C will not fit into cache.	25
10	Cycles per double floating point operation at M_B, N_B for $t_{fma} = 1, t_{setup} = 2, t_{load} = 5, csz = 2048$, and it is assumed that C is not required to be in cache.	26
11	Cycles per double floating point operation at $M_B = N_B$ for the example $t_{fma} = 1, t_{setup} = 2, t_{load} = 5, csz = 2048$, and it is assumed that C is not required to be in cache.	27
12	To the left, Fortran mapping of matrix to memory. To the right, C 's way of mapping the matrix.	29
13	$C = A \cdot B$, with blocking in one level. The loop order is I1, J1, L1, I2, J2, L2, that is, I1 is the outermost loop and L2 is the innermost	30
14	The mapping between the regular block column-major mode and block column format with block size equal to 2. The question marks denotes uninitialized data.	31
15	Using indirect addressing, we can store the blocks in any order in memory.	34
16	Mapping to recursive format. If tie, break on rows.	37
17	Mapping to recursive format. If tie, break on columns.	37
18	Fitting three 16×16 blocks in cache and still having room for two new blocks. In the $C = A^T B$ case, the next A and B blocks are following directly after the current A and B blocks. In $C = AB$, there is still room for a new A block occupying the same congruence class as the current A	47
19	A screen-shot of a matrix multiplication, showing the four-way associative level one cache, the direct mapped level two cache, and the two-way associative TLB. The coloring may be distorted in the figure.	57
20	The multiplication tree of an $120 \times 30 \times 80$ matrix multiplication with 32×32 blocks.	63
21	Finding a block with less than $\frac{288000}{4 \cdot 1.25}$ FMAs.	64
22	Finding a block with less than $\frac{288000 - 46080}{4 \cdot 1.25}$ FMAs.	64
23	Measured load times of the SP High Node.	70
24	Measured load times of a RS/6000 workstation with no level two cache, clock frequency 66 MHz. The leap at the right end is due to lack of physical memory.	70
25	Measured load times of a SGI workstation with 1 MB of level two cache, clock frequency 100 MHz.	71
26	Comparisons of different loop order when transposing matrices of different sizes on the SP High Node. The curve shows the performance ratio between a transpose algorithm that loads with stride n and stores with stride 1 and a transpose algorithm that loads with stride 1 and stores with stride n	71
27	Measured TLB miss times.	72

28	Matrix multiplication performance by various routines. Notice how performance drops with the DATBX routine where we have bad leading dimensions (multiplies of large powers of 2). Newer ESSL libraries has been reported to perform better, around 105 Mflops.	72
29	Matrix multiplication performance by our routines, with and without the overhead of data copying.	74
30	Matrix multiplication performance using the tree dynamic load balancing routine. Blocking 32×32 and no prefetching. Without accounting for data copying.	74
31	Matrix multiplication performance using the tree dynamic load balancing routine. Blocking 64×64 and 16×16 (see page 65 for a discussion of the blocked version of the blocked arbitrary ordered blocked matrix format and the subroutine <code>n2b_rec64</code>) and prefetching. Without accounting for data copying.	75
32	Speedup of the tree dynamic load balancing routine. Same conditions as in Figure 31.	75
33	Efficiency of the tree dynamic load balancing routine. Same conditions as in Figure 31.	76

List of Tables

1	Latency times in the memory hierarchy. One double word (DW) is eight bytes or one double precision floating point number. . . .	14
2	The attributes of the <code>blockmatrix</code> record.	34
3	The new <code>lda</code> parameter as described by Fred Gustavson. Some editorial changes have been done.	36
4	Average cycles per FMA for the three D0-loops algorithm, and register blocking of C . Each test case was run three times, and the mean of the two best runs is reported. Only problems where the sum of sizes the matrices is between 400 and 4000 double words are reported.	49
5	2×2 register blocking.	49
6	4×2 register blocking.	49
7	4×3 register blocking.	49
8	4×4 register blocking.	49
9	4×4 register blocking, but with repeated K -loop.	50
10	Same as Table 9, but with misaligned data.	50
11	Our assembler-made kernel 4×4 register blocking. To be compared to Table 8	50
12	Comparison of different prefetching techniques	51

1 Introduction

Linear algebra is an area which involves heavy computations and therefore is in need of high-performance routines to decrease the total computation time. In efforts to make this possible, parallel computers have evolved with different architectures. One architecture approach is shared memory multiprocessor machines such as the IBM SMP-node which is a member of the IBM SP family.

However, libraries developed until today have showed poor performance on this type of machines.

1.1 Objectives

The objectives of this master's thesis are,

- to find out the bottlenecks when programming the SMP-nodes in parallel on the IBM SP.
- give guidelines for programming the SMP-nodes in parallel with increased performance.

1.2 Method

To achieve deeper understanding in the area of subject much effort were in the beginning put on reading books, articles and reports. In this way it was possible to get theoretical knowledge and surround the main topics in development of high-performance matrix multiplication routines and linear algebra.

A lot of testing of the already existing libraries have been done to evaluate and examine different approaches of matrix multiplication.

The theoretical knowledge together with the testing results from past implementations of other teams gave us ideas of how to increase the overall performance and pioneering thinking in the area storage mapping models.

To verify our thoughts and explore specific parameters affection on performance, numerous lines of code have been implemented. Examples of programs are timing of the memory hierarchies, simulation of cache references and a matrix multiplication to verify the concepts.

Beyond this, talks and meetings with Erik Elmroth, Fred Gustavson, Bo Kågström, and Per Ling ended up in many great ideas.

1.3 Outline

In Section 1 we explain why this thesis is done, and our objectives.

Section 2 describe some theoretical concept, in order for the reader to be able to delve into the later sections. We explains words like *speed-up* and *efficiency*, *matrix multiplication* and *matrix blocking*. This section also covers some of the libraries for linear algebra computation available.

In Section 3 we describe the hardware, the units that are involved in matrix multiplication. This section covers how these units interact with each other and the characteristics of the units. We also describe our target platform, the IBM SP High Node, an SMP-node.

Section 4 contains a thorough discussion of *threads*, a programming model suitable for SMP-nodes. Threads are not available in all languages however. This section explains why.

In Section 5, three other high-performance libraries and approaches are covered. Their underlying ideas are shown.

Section 6 covers a discussion of matrix blocking and a mathematical investigation of the metrics. We have developed formulas for the optimal blocking sizes when a practical implementation detailed is taken into account, the setup size around the innermost loop.

In Section 7, different data structures and storage patterns for matrices are investigated. We show our implementation of routines that copies matrices to and from new, blocked routines.

Section 8 explains the kernel routines, that is, small efficient kernels that performs matrix multiplication on small, fixed sized, matrices. We report our measurements and we show a way of laying out instructions optimally. The technique of prefetching is also explained, and results from experiments with prefetching are displayed.

In Section 9, we are presenting a tool for cache simulation that we have implemented. We explains its foundations and its shortcomings. Examples of runs with the tool are shown and commented.

Section 10 contains a new idea for fast dynamic load balancing that we have developed. It is based on a tree made out of the recursive multiplication. We describe why it fits well into our data structures.

In Section 11, we describe how our testings were performed.

Section 12 contains our results of the matrix multiplication routines, both for one processor and several. We also describe the measurements performed to find the metrics of our target architecture.

Finally, in Section 13, we summarize our results and give explanations of the results. We also describe what is left to be done in this field.

2 Concepts and background

To be able to get deeper into this master’s thesis the reader should get acquainted with some basic knowledge, generally in the area of parallel programming, and especially in the area of parallel matrix multiplication. This section describes the terminology and gives an overview of parallel programming.

2.1 Theoretical performance

When using parallel systems the aim is to solve a given problem much faster compared to a single processor system. To be able to discuss different performance issues we introduce some standard definitions.

Assume that we have a parallel algorithm that is identical to the best sequential algorithm, i.e., they solve the same problem. The *speedup* is then defined as the ratio of the serial run time of the best sequential algorithm for solving a specific problem to the time taken by the parallel algorithm to solve the same problem on p processors. This can be written as,

$$S = \frac{T_S}{T_P} \quad (1)$$

where T_S is the serial run time and T_P is the parallel run time.

Normally, $1 < S \leq p$ because practically there should be no way a processor can devote 100% of its time doing computations, execution time is spent on communication, context switching and so on.

But there are rare cases when, $S > p$, which corresponds to a situation where we have *super-linear speedup*. This can for example be achieved on distributed memory machines when the problem fits into the level 2 cache on all processors.

Amdahl’s law states that the speedup of an algorithm is effectively limited by the number of operations which must be performed sequentially, i.e., its *serial fraction*. Algorithms that parallelize well are those with very small serial fractions. If we denote the ratio of the serial fraction in a parallel algorithm with α and the number of processors with p , Amdahl’s law suggests that the maximum achievable parallelism is upper-bounded by $1/\alpha$ and we therefore can achieve a maximum speedup of,

$$S = \frac{p}{\alpha p + (1 - \alpha)} \quad (2)$$

If a program for example executes in serial mode 20% of its time, one should not expect a speedup of more than 5, even if an infinite number of processors are added. The constant α is called the sequential bottleneck. Note that Amdahl’s law stands even though communication overhead is excluded, i.e., including it will only exacerbate the sequential bottleneck.

Ideally, we would want $S = p$, where p is the number of processors, therefore we define the *efficiency* of a parallel program as,

$$E = \frac{S}{p} \quad (3)$$

which is the ratio of processor utilization. The efficiency is normally between 0 and 1 but can exceed 1 in cases of super-linear speedup.

If we assume that W is the cost of solving a problem (function of the problem size) with p processors, we can denote the communication overhead cost (function of problem size, number of processors and type of machine topology) as,

$$T_O = pT_p - W$$

In this way we can rewrite the efficiency formula as,

$$E = \frac{W}{W + T_O}$$

With a fixed problem size, the efficiency decreases as n increases. Generally speaking T_O grows slower than W as the problem size increases. Thus, with a fixed number of processors, the efficiency increases as the problem size increases. Generally speaking, as the number of processors increases, one has to increase the problem size in order to maintain a constant efficiency. The amount of growth needed in the problem size to balance the increase in number of processors (for a constant efficiency) is called the *isoefficiency* function. The lower the order of the isoefficiency function the more *scalable* the parallel system is.

Another way of saying this is that an algorithm is scalable if and only if

$$\lim_{n \rightarrow \infty} \frac{T_\infty(n)}{T_s(n)} = 0$$

where $T_\infty(n)$ is the running time of a parallel algorithm on a problem size n using an unlimited number of processors. The cause of a non-scalable algorithm is often a sequential phase impossible to parallelize or the amount of communication time between processors increases more rapidly with an increased number of processors.

Further discussions are found in [KGGK94].

2.2 Matrix multiplication

A basic problem in high-performance and scientific computing is matrix multiplication,

$$C \leftarrow C + A \cdot B$$

where A is of size m -by- r , B is of size r -by- n and C of size m -by- n . Also assume that all matrices are *dense*, i.e. most of the entries are nonzero. Each element $C_{i,j}$ in the matrix product can explicitly be expressed as,

$$c_{i,j} \leftarrow c_{i,j} + \sum_{k=1}^n a_{i,k} \cdot b_{k,j}$$

This means that each value of C is a *dot product* of two vectors, one row from the A and one column from the B , which is illustrated in Figure 1.

The basic algorithm for doing a matrix-multiplication without any performance improvements builds upon loops over the columns and rows of the matrices A and B , i.e., dot products, which in a pseudo language can be written as,

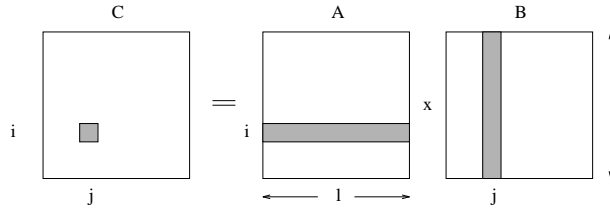


Figure 1: Matrix multiplication using dot product

```

for i = 1:m
  for j = 1:n
    for l = 1:k
      C(i,j) = C(i,j) + A(i,l) * B(l,j)
    end
  end
end
end

```

By arranging the loops in other ways different operations will be performed in the innermost loop. In total there are six (permutation of 3) different ways to arrange the three loops. All give, of course, the same result but their data access pattern differ. A beneficial feature with the dot product is that the result can be kept in a register during the whole computation. Other variants include *_AXPY* or *outer product*

An *_AXPY* operation looks like,

$$y = \alpha x + y$$

where x and y are vectors of the same size and α is a scalar, see Figure 2.

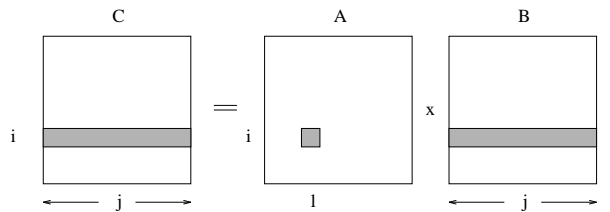


Figure 2: Matrix multiplication using *_AXPY*

Instead of keeping the result in a register this operation stores one of the element operands from A or B in a register fulfilling the operation. Using pseudo language this can be written as,

```

for i = 1:m
  for l = 1:k
    for j = 1:n
      C(i,j) = C(i,j) + A(i,l) * B(l,j)
    end
  end
end
end

```

An outer product operation has the form,

$$C = A + xy^T$$

where C and A are m -by- n matrices, x is a m -by-1 vector, and y is a n -by-1 vector, see Figure 3.

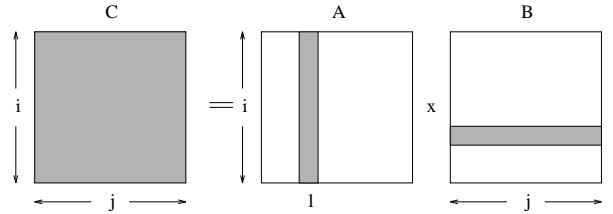


Figure 3: Matrix multiplication using outer product

The *outer product* which has higher granularity than the other two operations is preferable when using distributed parallel systems. On shared memory parallel systems this operation may cause problems keeping good cache consistency which implies penalty time. Using pseudo language this multiplication can be illustrated as,

```

for l = 1:k
  for i = 1:m
    for j = 1:n
      C(i,j) = C(i,j) + A(i,l) * B(l,j)
    end
  end
end
end

```

Further discussions are found in [KvL88].

2.3 Matrix blocking

The three DO-loops may be an easy way to matrix multiplication, but they are very inefficient. The reason for this is that each loaded value from memory is used very infrequently, so rarely that it may be unloaded from the cache until the next time it is used. A better method would be to use each loaded value many times in a row, and then let it be unloaded from cache memory by working with other elements. Of course, we cannot just use the elements in any way we want, instead, we must stick to definition of matrix multiplication. If we are performing an $(M \times N \times K)$ multiplication, that is multiplying $C = C + A \times B$ where A is $M \times K$, B is $K \times N$, and C is $M \times N$, there are MNK multiplications and additions. We can reorder these in $MNK!$ ways. It is obviously impossible to try all combinations, though there are some schemes that give a performance close to optimum. This is matrix blocking. An example, which blocks in one level, is

```

for i = 1:mb:m
  for j = 1:nb:n
    for l = 1:kb:k

```

```

    for ib = 0:min(mb-1,m-i-1)
      for jb = 0:min(nb-1,n-i-1)
        for lb = 0:min(kb-1,k-i-1)
          C(i+ib,j+jb) = C(i+ib,j+jb) +
            A(i+ib,l+lb) * B(l+lb,j+jb)
        end
      end
    end
  end
end
end
end

```

Here, if mb , nb and kb , which are algorithmic specific parameters are sufficiently small, the elements references in the three innermost loops will be held in cache and reused. If the parameters are too small however, the performance will degrade since only a few elements are reused.

Another motive to block matrices is to make portions of the problem easy to parallelize.

2.4 BLAS

The BLAS, or Basic Linear Algebra Subprograms [LHKK9a], has become a defacto-standard for basic vector and matrix operations with the aim to provide a portability layer for computation. The BLAS is divided into three levels depending on the *time complexity*. Time complexity is associated with the number of floating-point operations carried out. Another term often used is *space complexity* which defines the number of memory references done by a BLAS subroutine.

Level 1 BLAS do vector-vector operations and have the time complexity $\mathcal{O}(n)$ and space complexity $\mathcal{O}(n)$. The problem in level 1 BLAS is that there is too much memory traffic related to floating point operations giving poor performance.

Level 2 BLAS do matrix-vector operations and have the time complexity $\mathcal{O}(n^2)$ and space complexity $\mathcal{O}(n^2)$.

Level 3 BLAS do matrix-matrix operations and have the time complexity $\mathcal{O}(n^3)$ and space complexity $\mathcal{O}(n^2)$. A significant difference occurs compared to the other two levels of BLAS, the time complexity is larger than the space complexity which implies a higher total performance. The secret of increasing the performance is to have an as high as possible ratio of computations per memory reference.

The higher level of BLAS, the higher level of granularity, which is the amount of computations that can be done in parallel. Increased granularity implies lower synchronization costs and therefore increases the performance.

In high performance computing there are several software libraries like LINPACK and LAPACK that are using BLAS as its foundation of computation because of the efficiency and portability across high-performance platforms from different vendors. The availability is another contributing cause of its success. It is public domain software and free to download from NetLib and vendor-provided versions often come with the hardware.

2.5 LAPACK

LAPACK, or Linear Algebra PACKage [ABB⁺94], is a library of Fortran subroutines for solving the most commonly occurring standard problems in numerical linear algebra, computational routines to perform a distinct computational task, and auxiliary routines to perform a certain subtask or common low-level computation. Typical examples of problems are systems of linear equations, linear least squares problems, eigenvalue problems and singular value problems. LAPACK can also handle many associated computations such as matrix factorizations or estimating condition numbers.

LAPACK is designed to be efficient on a wide range of modern high-performance computers, this means that the library is not tuned for a specific architecture. One of the key ideas behind LAPACK was to make a library of subroutines where the most of the computing is done in BLAS subroutines, therefore LAPACK expects the vendors to provide a set with tuned BLAS to achieve high performance.

The LAPACK routines are written as a single thread of execution, each routine has one or more parameters which are submitted with the algorithmic structure calls to the BLAS. The parameters are obtained when installing the package and stored in a table which is referenced at runtime. A typical example of a parameter is the size of the blocks operated on by BLAS, note that the user never sees these parameters when calling LAPACK subroutines.

2.6 ScaLAPACK

The ScaLAPACK, or Scalable LAPACK [BCC⁺97], is a library that includes a subset of LAPACK routines redesigned for distributed memory parallel computers. Just as LAPACK routines calls the BLAS subroutines, ScaLAPACK routines uses calls to the PBLAS.

PBLAS, or Parallel Basic Linear Algebra Subprograms, rely on the communication protocols of the BLACS and the computation subroutines of BLAS.

BLACS, or Basic Linear Algebra Communication Subprograms use explicit message passing for interprocessor communication. The BLACS are designed for linear algebra applications and provide portable communication across a wide variety of distributed-memory architectures. The goal is to have ScaLAPACK routines resemble their LAPACK equivalents as much as possible.

2.7 ESSL

ESSL (Engineering and Scientific Subroutine Library) [IBM97a] is a high performance library of mathematical subroutines for the IBM RISC system. The subroutines cover many computational areas, for example linear algebraic equations, matrix operations, eigensystems analysis, Fourier transforms, interpolation, and random number generation. ESSL, which is provided by IBM can be split up into two categories of subroutines.

First, the foundations of ESSL which is BLAS, where the complete set of basic linear algebra subroutines are fully tuned for specific machines.

Second, a set of routines to for example solve linear equations of various kinds, which functionally are the same as the public domain software LAPACK. To use these ESSL routines, modifications to the application code need to be

done, though in most cases it will increase the total performance of the application.

User applications that already is developed using calls to the LAPACK library could easily linked with ESSL for tuned BLAS subroutines. This means that code which calls BLAS and LAPACK subroutines is fully reusable without any changes. This approach achieves slightly lower performance than direct calls to the ESSL routines.

The large number of different subroutines in ESSL also reduces the effort needed to develop new applications in the area of scientific computing and finance, which have in common its numerically intensity.

3 Hardware

A computer of today (serial and parallel) includes different storing devices or memories. These devices are different in size, latency, and the physical distance to the processor itself. This section will in some detail describe the devices that affect the performance of a matrix multiplication.

3.1 Memory hierarchy

In a general memory hierarchy there are several levels, including registers, cache, main memory and disk areas, see Figure 4. When the processor accesses data, the most recently used data are stored in registers which are a part of the processor and used for computations. These registers are small and few in number which forces the processor to fetch new data from the outside. The first level outside the processor is the cache. If the data is found at this level (cache hit) it is sent to the processor, otherwise (cache miss) the request is passed down to the memory. As a final instance, at level three the data is to be found on the disk. When the data is found it is passed up in the hierarchy and a copy is stored at each level, which forces items at the higher level to be removed when storage space is exhausted.

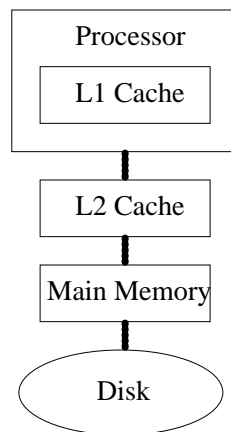


Figure 4: Memory Hierarchy

3.2 Cache

The cache can be split up in several levels. Today, normally the number of levels of cache memory is spanning from one up to three. The level one cache normally is on-chip, i.e., it resides on the processor and therefore has the lowest latency in the memory hierarchy. To keep low latency in the level one cache it is limited in size, normally 8-64 kB¹. The level two cache is normally off-chip, and has therefore less limitations and therefore a higher latency. The size of the level two cache is normally between 256 kB and 4 MB². Notable is that each

¹1 kB is 1024 bytes

²1 MB = 1024 kB = 1048576 bytes

processor can have its own level two cache memory or it can be shared between processors.

The usefulness of two caches instead of one cache is dependent upon the application. If the data references are spread out, i.e., if space locality is not utilized, a second cache (L2) increases the total performance of the applications. Typical applications to benefit from this are programs with frequent branches and widely accessed data which will cause cache swapping at a high rate. Applications that access data in tight loops where the data is well mapped into the level one cache barely need a level two cache to increase the total performance in the system.

The caches are often high speed SRAM chips and therefore very expensive as size grows.

3.3 Main Memory

Main memory is the last level in the hierarchy entailed by the common term volatile memory, that is, when the power is shut down everything stored is lost. The size of main memory in high performance systems can be very large, normally 256 MB and higher. Today, sizes above 1 GB³ is frequently occurring.

3.4 Disk

The disk storage is the last level in the hierarchy and is a member of the group of permanent storage, that is, after the power has been shut down the stored data can still be retrieved when the power is switched on again.

3.5 Translation lookaside buffer

All program code and data is stored in memory and is located in the virtual address space. The virtual address space consists of a memory space located in the RAM-memory and disk space. The translation lookaside buffer (TLB) translates the virtual address into a physical address in the physical address space and stores these addresses as virtual pages in the TLB. If access is attempted to a virtual page not in the TLB, a TLB miss occurs and forces the TLB to load a page entry from the global pagetable which resides in memory, resulting in a penalty. It might then be a good idea to use loop blocking or data reformatting, with respect to the TLB, to keep the performance in place.

3.6 What is an SMP-node?

A MultiProcessor-node is a cluster of processors which can be seen as a homogeneous system. There are two types of multiprocessors; *shared memory* and *distributed memory*, see Figure 5. A *shared memory* multiprocessor consists of multiple processors, each capable of addressing all memory areas and devices. A typical example is the *Symmetric MultiProcessor* (SMP) where the system looks exactly the same to each processor in the system. This means that any process, independently of the processor, can *see* everything on the machine and can access the same bytes in memory as any other process, which makes a good

³1 GB = 1024 MB = 1073741824 bytes

*cache consistency*⁴ solution important. This system is very efficient and is highly scalable if a lot of data is to be shared. The programming model when using SMPs is the same as for a uniprocessor system which is a great advantage.

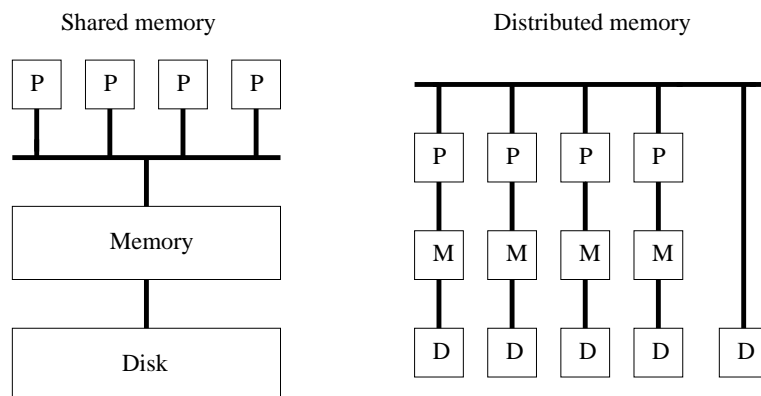


Figure 5: Types of SMP-nodes, shared memory and a distributed memory system.)

In a *distributed memory* multiprocessor all processors have their local memory and disks which implies message passing to establish interconnection between the nodes. This solution has generally better scalability than the shared memory solution because there is no memory bus contention or *cache coherency*⁵ problems. On the other hand the message passing programming model is a bit more complicated.

3.7 The IBM SMP-node

The IBM SMP-node, or SP High Node, is a shared memory multiprocessor with four processors on each node. Each processor has 32 integer and 32 floating-point registers. There are two levels of cache, a level one (L1) cache which is on-chip and a level two (L2) cache which is off-chip, non-shared, per processor. The cache sizes are, 16 kB L1-cache and a 512 kB L2-cache. The system nodes have a main memory size of 256 MB per four processors The TLB has 128 entries and can therefore store 128 translations where each virtual page is of size 4 kB each, which will span over $128 \cdot 4096 = 524288$ bytes or 0.5 MB of memory.

Several tests have been done in order to measure the latency time at different levels of the memory hierarchy, see Table 1. For a more thorough discussion, see the result section.

⁴As all data in cache also resides in main memory, an updated value in cache also has to be updated in the main memory. Different types of protocols handles this, which are unique for the machine.

⁵In a multiprocessor system data can reside in several caches at the same time. Hardware is built into the system to keep all caches apprised of where the most recently updated copy is located.

Unit	Number of DW	Average latency time (cycles)
Registers	32	0
L1 cache	2048	1-2
L2 cache	65536	≈ 5
Main memory	33554432 (32 MDW)	$\approx 15-20$
TLB	128 pages of 512 DW each	40

Table 1: Latency times in the memory hierarchy. One double word (DW) is eight bytes or one double precision floating point number.

4 Threads and SMP

The SMP-nodes consist of multiple processors and a shared memory. The operating system schedules processes on the processors which work in parallel. A more convenient way of parallel programming is to use threads, an easier to handle form of standard processes, which shares the whole memory area together and therefore has its pros and cons. The following section give further details about threads and SMP-nodes.

4.1 Introduction to threads

By definition, a standard process, or *heavyweight process*, is a task with exactly one thread. Before the emergence of threads, the normal way to achieve multiple instruction sequences under UNIX operating systems (i.e., doing several things at once, in parallel) were to use the `fork()` and `exec()` system calls to create several processes, each consisting of a single thread of execution. Resources that are private to a process are a virtual address space (i.e., stack, data, and code segments), system resources (e.g., open files), and a thread of execution. To achieve interprocess cooperation to the process, external communication channels are needed. UNIX systems provide several mechanisms for accomplishing *interprocess communication* (IPC). Examples of IPCs are sockets, pipes, semaphores, shared memory segments and message queues. In a threaded environment, this model of parallelism is turned on its head.

4.2 What is a thread?

A thread [NBF96], which is sometimes called a *lightweight process* (LWP), is a basic computational unit, an encapsulation of the flow control in a program. It consists of a program counter, a register set, and a stack space. A thread shares with its peer threads (threads within the same process) its code section, data section, and operating-system resources (open files and signals), see Figure 6. The collection of threads form a single process; each thread in the process sees the same virtual address space, files, etc. An interesting implication is that if one thread opens a file, all its peer threads can also access the file. Attention should be given to shared resources and the protection from side effects when using threads.

The sharing between threads make them inexpensive compared to heavyweight processes due to the lack of context switching (pure overhead, no useful work is done), although it still requires a register set switch. A big advantage with threads is consequently that no memory management is needed at all when using peer threads, this means that using multithreaded control is especially efficient when the different processes are related through, for example, shared data. Sharing data (memory) between heavyweight processes must be done explicitly.

Often threads are implemented at the user level, implying that no system calls are performed and therefore no calls to the operating system are done, causing interrupts and, thus, latency penalties. All in all this implies that switching between threads is very fast because it can be done independently of the operating system.

A drawback with threads is that if the kernel is single-threaded a user-level thread performing a system call will cause the entire task to wait for the return

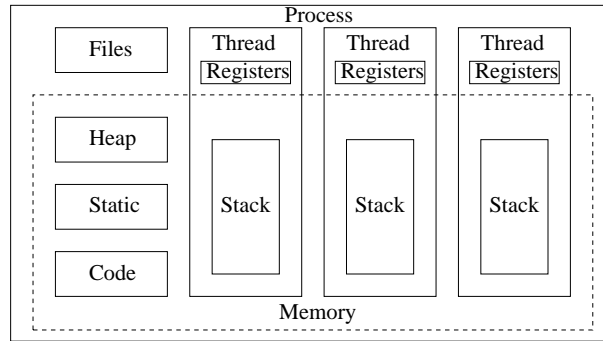


Figure 6: An overview of a process and its user-level threads. Everything in the memory area is shared, except for the stack.

value. On the other hand, if the kernel is multi-threaded it is possible to bind a user-level thread to a kernel thread and, thus, obtain parallelism. This is especially efficient when using a multiprocessor, where the kernel threads then can be distributed on different processors.

The use of threads is often praised for being much more efficient than using the `fork` statement to create new processes. To investigate how much time is spent using threads we made some tests. A `pthread_create` call, at its minimum, is consuming 2.5 milliseconds which can be related to the time consumed by a `fork` call which requires at least 4.5 milliseconds.

4.3 Threads and signals

A signal is a software interrupt which can be caused by illegal system calls or user generated. Signals can also be used for interprocess communication. There are two types of signals, synchronous and asynchronous. Synchronous signals are caused by a thread's own actions, such as when a thread performs division by zero or attempts to access memory outside its address space. Asynchronous signals are caused by something external to the thread, such as another process sending a `kill()` signal or the user pressing `<Ctrl-\>` on the keyboard (causing a `SIGQUIT` signal).

All synchronous signals are targeted to a specific thread and are always handled by the offending thread. The asynchronous signals are targeted to the entire process; it is then up to signal masks to decide which thread is to process the signal.

In a single-threaded program, the global variable `errno` is set to an error value when a system or library call fails. In a program with multiple threads, the global `errno` variable is replaced by a per-thread `errno`. When a failing system or library call sets the per-thread `errno`, the other threads' `errno`s are not affected.

4.4 Distribution of threads on SMP-nodes

An SMP-node with the AIX operating system running, one common run queue is accessed by all processors on the node. The thread next in line is distributed to the first processor that becomes idle. When a thread blocks or when its

quantum expires the thread will be put back in the queue and removed from the processor it is currently residing on. One consequence of this is that a thread is not guaranteed to run on the same processor as before when it is next in line. Threads will bounce around and cause cache misses which will decrease the performance. A solution to this problem is to let threads always run on the same processor, and hope that most of their instructions and data will remain in the cache. Using the same processor when in kernel-mode is called *processor affinity*. AIX supports weak processor affinity; if two or more threads have been given the same priority a processor chooses the one that last ran on it. A stronger form of affinity is to bind the thread to a specific processor. It is important to keep in mind that when binding a thread to a processor it might cause the total system throughput to decrease because the threads are bound and not scheduled to idle processors.

One problem when creating threads is that they are treated the same way as processes, when accessing the processor kernel. Each thread running has a quantum assigned to it which will dispatch it from the kernel.

4.5 Thread safety

The threads library code is written in C which makes it highly tuned and suitable for use in standard C programs. However, in scientific computing the most common language is Fortran and tests have shown Fortran 77 code is not reentrant and thus not safe to use by multiple threads. This problem is caused by the compiler that generates code which uses a static memory allocation and therefore cannot be called recursively or in parallel. A work-around to this problem is to use `goto` statements. The static memory allocation also creates problems that are associated with local variables in procedures or functions. When called in parallel these local variables get over-written by the other threads due to fixed memory allocations associated with the local variables in the procedures or functions. This last problem can be solved if every thread gets its own instance of the procedure or function. A much more effective way to get around all problems with threads is to use Fortran 90 which, as C, puts all local variables on the program stack.

The above problem is often called a *non-thread safety* problem, that is when race conditions exist between processes concerning allocation of resources. One remedy is to add mutexes into the code which only allows one thread at a time to access a certain memory area or a library routine.

5 Previous Work

A primary area in parallel programming has for a long time been developing libraries of routines for linear algebra problems. Of course there have been many contributions⁶ which have shown very good results, but we think that there are three specific projects that recently have explored new dimensions which have helped us in the development of a matrix multiplication routine. These projects are GEMM-based level 3 BLAS ([KLvL95] and [GHJ+98]), ATLAS [WD97], and PHIPAC [BAD+97]. In the following we give a short summary of these projects.

Today, there are BLAS libraries developed for many different architectures which are very effective and has high performance. The problem is that the development is expensive and time-consuming, especially the hand-optimization part and the vendors often wait until there is a clear market demand of a BLAS library for a specific machine before they start the development. The programmer must understand the machine, know the memory hierarchy and functional units and so on.

5.1 Superscalar GEMM-based level 3 BLAS

In previous work by Kågström, Ling and Van Loan [KLvL97] development of a GEMM-based (GEMM, GEneral Matrix Multiply level 3 BLAS⁷ library has showed that it is possible to build all level 3 BLAS routines on DGEMM and a small amount of level 1 and 2 BLAS subroutines. This way of developing libraries simplifies the overall development of a optimized level 3 BLAS because the only level 3 BLAS routine that needs to be implemented is the DGEMM which together with this library gives access to all level 3 BLAS routines. Notable is that this approach assumes that the vendors provides a highly optimized DGEMM and some level 1 and 2 BLAS routines to their machines.

The general idea of the GEMM-based level 3 BLAS is to strip down every matrix multiplication into blocks where general matrix multiplication can be used, in this case the DGEMM. The strips of blocks are either block row or block column. For further improvement of the performance, the blocks can be divided into subblocks.

Further improvements on this approach has resulted in the Superscalar GEMM-based level 3 BLAS [GHJ+98] which has decreased the number of routines that needs to be implemented to have a full level 3 BLAS library. In fact, only one routine is needed, the GEMM. The level 1 and 2 subroutines required for the GEMM-based level 3 BLAS library are nowadays replaced by *in-line code*⁸ This in-line code make use of 4×2 unrolling of the inner-most loop and as a implication of data copying and work arrays the references are stride one.

Invoked in the Superscalar GEMM-based level 3 BLAS is the possibility of parallelism, this can be achieved either by a parallel GEMM routine or by automatic parallelization by a compiler.

⁶Especially, Fred Gustavson, IBM, who has influenced us in our work with comments and new ideas.

⁷Level 3 BLAS consists of subroutines performing matrix-matrix multiplication

⁸Explicit code instead of library call.

5.2 ATLAS

The goal of the project is to develop a methodology for automatic code generation of linear algebra routines, with the preconditions of a on-chip cache and a good C-compiler. The result is called Automatically Tuned Linear Algebra Software (ATLAS)[WD97]. The main focus of the initial work have been spent on developing a general matrix multiplication, DGEMM, of the form,

$$C \leftarrow A^T \cdot B + C$$

Though, a lot of the technology and the approach can be applied to other routines in level 3 BLAS.

Looking into the code which is generated it can be split up in two parts; First, an optimized on-chip matrix multiplication code which is automatically generated. With timings the code generator is able to determine the best-fitted blocking and loop-unrolling factors for the specific target machine.

Second, the rest of the code is the same across all architectures, and handles the looping and blocking and so on to build a complete matrix-matrix multiplication.

The result of the timings also determines the minimum size of a problem where the ATLAS algorithm is used, hence if the minimum size is not attained a standard 3-loop multiply is used. This is done because the overhead in the *optimized* ATLAS-routines due to function calls and multiple layers of looping, might make it slower.

The blocking strategy is to split the main matrices into minor blocks of size $n \times n$. When the dimension is not a multiple of the blocking factor, clean-up code takes care of these cases.

The ATLAS team also stated that all 3 matrices need not to be in the cache to optimize cache reuse. Instead, they claim that it is not necessary except when the cache policy is not write-through, which saves the cost of pushing previously used sections in C back to level 2 cache.

5.3 PHiPAC

When using general linear algebra libraries a problem that often arises is that explicit tuning needs to be done to achieve high performance on a specific machine. The work-around to this problem is to make a machine-specific library that is by definition well tuned and therefore has a high performance. This solution is generally not optimal due to its platform dependency.

Jeff Bilmes, Krste Asanovic, Jim Demmel, Dominic Lam and Chee-Whye Chin have gone one step further: Instead of just developing regular library routines and handing over the tuning to the user, or developing a machine specific library, they have developed a system that automatically generates BLAS libraries tuned to a specific machine. The system is called Portable, High-Performance, ANSI C, abbreviated PHiPAC [BAD⁺97].

The methodology used to obtain these libraries consists of three components:

- A set of guidelines for producing portable high performance ANSI C code.
- A generator which produces code according to the guidelines.

- A search-script that automatically tunes the code for a specific machine. Parameters are combined with different values; compiled, timed and benchmarked to obtain the best possible combination. Tunable parameters include the number of integer and floating-point registers, sizes of each level of the caches, and blocking in the memory hierarchy.

This way of developing linear algebra libraries has been successful, on several machines this methodology has achieved over 90% of peak performance.

In order to produce ANSI C code which yields high performance on different machines and compilers, the PHiPAC-team analyzed the output from different compilers to determine which characteristics of the code affects the output, and how. The results ended up in these guidelines that in a general way show common characteristics that will improve the performance of the code.

- *use of local variables to explicitly remove false dependencies*, which often occurs when the compiler assumes pointers aliasing.
- *exploit multiple integer and floating-point registers*, when having references in loops the compiler often reloads the values in each iteration, this can be avoided by using local variables.
- *minimize pointer updates by striding with constant offsets*, use constant array offsets and move the basepointer instead of pointer updates of the variables.
- *hide multiple instruction FPU latency with independent operations*, which will increase performance using pipelined or superscalar processors.
- *balance the instruction mix*, a floating-point multiplication, a floating-point add, and 1-2 floating-point loads or stores interleaved.
- *increase locality to improve cache performance*, if possible, arrange the memory accesses to be stride one.
- *convert integer multiplies to adds*, the add operations is much faster than multiplies.
- *minimize branches, avoid magnitude compares*, unroll loops, use C `do {} while()` which removes the branch compared to C `while()` when terminating, try to use equality or inequality loop terminations instead of magnitude comparisons which is less expensive.
- *loop unroll explicitly to expose optimization opportunities*, when unrolling loops it is often possible to raise the floating-point operations per memory access ratio.

All these guidelines have been known for a while but the PHiPAC-team have made a good compilation.

6 Optimal blocking

In this section we will discuss the optimal blocking for on-chip cache for the SMP-node. That is, for a multiplication $C = AB$, where A is $M \times K$, B is $K \times N$, and C is $M \times N$, we want to find values for block sizes M_B , N_B , and K_B so that the multiplication performs as fast as possible. In general, blocking for cache leads to square blocks since this leads to the largest volume-to-surface ratio, that is for each blocking sizes that fits into cache we want to do as many floating point operations as possible. However, in order to achieve fast kernel routines, we want the innermost loop to have as many iterations (K_B) as possible. In this section we discuss how to find the combined optimum. This work was done independent of the work done by Gallivan, Jalby, Meier, and Samah, see [GJMS88]

6.1 A , B , and C in cache

Suppose a kernel is possible to do a (M_B, N_B, K_B) matrix multiplication on approximately $((M_B N_B K_B)t_{fma} + (M_B N_B)t_{setup})$ cycles if the data is in cache, $(M_B K_B) + (N_B K_B) + (M_B N_B) \leq csz$, where t_{fma} is the time to perform a multiplication, t_{setup} is the time for setting up the innermost loop, and csz is the cache size in double words. Also suppose that the time to load data to cache is uniform, that is $((M_B K_B) + (N_B K_B) + (M_B N_B))t_{load}$ where t_{load} is the time to load a floating point number (double word) from cache to register.

The objective is to find values on (M_B, N_B, K_B) in order to perform the multiplication (M, N, K) as fast as possible. Let $M = mM_B$, $N = nN_B$, $K = kK_B$. The time for such a multiplication is

$$T(M_B, N_B, K_B) = mnk \cdot ((M_B N_B K_B)t_{fma} + (M_B N_B)t_{setup}) + [M_B K_B + N_B K_B + M_B N_B]t_{load} \quad (4)$$

Inserting $m = \frac{M}{M_B}$, $n = \frac{N}{N_B}$, $k = \frac{K}{K_B}$ we get

$$T(M_B, N_B, K_B) = MNK \cdot \left(t_{fma} + \frac{t_{setup}}{K_B} + \frac{[M_B K_B + N_B K_B + M_B N_B]t_{load}}{M_B N_B K_B} \right)$$

By differentiating the expression with respect to K_B , we get

$$\frac{\partial T}{\partial K_B} = -MNK \frac{t_{setup} + t_{load}}{K_B^2} \quad (5)$$

which implies that if M_B and N_B are constant then the time is monotonically decreasing with respect to increasing K_B . The formula is however valid only for $M_B K_B + N_B K_B + M_B N_B \leq csz$, so we can assume that a full cache use is preferred, and inserting

$$K_B = -\frac{M_B N_B - csz}{M_B + N_B} \quad (6)$$

into (5) gives

$$T(M_B, N_B) = MNK \cdot \left(\frac{(csz - M_B N_B)t_{fma}}{csz - M_B N_B} + \frac{M_B N_B (M_B + N_B)t_{setup} + csz (M_B + N_B)t_{load}}{(csz - M_B N_B) M_B N_B} \right) \quad (7)$$

Now inserting values for t according to the metrics of the SP High Node (see the Kernel and Results section): $t_{setup} = 2$ (there are loads and stores of 16 C -element around the inner loops, the number of iterations of the outer loops are $\frac{M_B}{4}$ and $\frac{N_B}{4}$ respectively, $\frac{32}{4 \cdot 4} = 2$, see the Kernel section), $t_{fma} = 1$, $t_{load} = 5$ and the size of the on-chip cache, $csz = 2048$ double words.

$$T(M_B, N_B) = MNK \cdot \left(1 - \frac{2M_B + 2N_B}{M_B N_B - 2048} - \frac{10240M_B + 10240N_B}{(M_B N_B - 2048)M_B N_B} \right)$$

We remark that $T(M_B, N_B)$ is proportional to the problem size to the optimal block size and that the optimal block size is the same for all sufficiently large problems.

In Figure 7 it is shown that there is a minimum for the High Node values at approximately $M_B = N_B = 25$, although the surface is rather flat around the minimum.

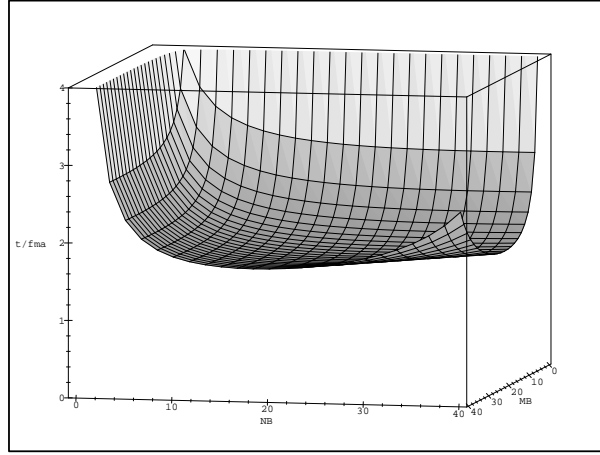


Figure 7: Cycles per double floating point operation (addition and multiplication) at different M_B, N_B for $t_{fma} = 1$, $t_{setup} = 2$, $t_{load} = 5$, $csz = 2048$.

By solving

$$\begin{cases} \frac{\partial T(M_B, N_B)}{\partial M_B} = 0 \\ \frac{\partial T(M_B, N_B)}{\partial N_B} = 0 \end{cases} \quad (8)$$

for positive M_B, N_B we find the minimum at exactly

$$M_B = N_B = \sqrt{\frac{\sqrt{csz^2 (t_{load}^2 + 10t_{load}t_{setup} + t_{setup}^2)} - 3t_{load}csz - t_{setup}csz}{2t_{setup}}} \quad (9)$$

which leads to the optimal values $M_B = N_B \approx 24.1$, $K_B \approx 30.3$. The minimum is also shown in Figure 8 when plotting $T(M_B, N_B)$ against $M_B = N_B$.

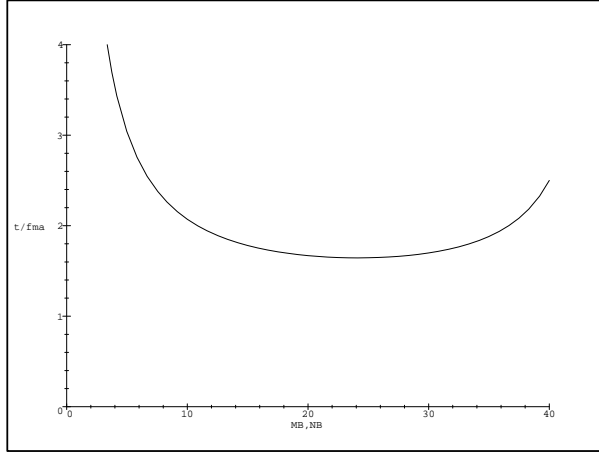


Figure 8: Cycles per double floating point operation at $M_B = N_B$ for the example $t_{fma} = 1$, $t_{setup} = 2$, $t_{load} = 5$, $csz = 2048$.

6.2 A and B in cache

Even though $M_B K_B + N_B K_B + M_B N_B \leq csz$ there are cases where the submatrices (blocks) do not fit in cache. Consider the PowerPC 604 processor, where the cache is four-way associative, and assume $M_B = K_B = N_B = 24$. Then $3 \cdot 24^2 = 1728 < 2048$, so one might think that the submatrices would fit into cache easily. But if all matrices are page aligned, that is, the address of the first element of each matrix is a multiple of 512 double words, the matrices will overlap in cache and there will be some cache thrashing, see Figure 9. Generally, to be sure that a set of blocks of sizes b_1, \dots, b_n fits into an m -associative cache of size sz , we have to require that

$$\frac{sz}{m} \sum_{i=1}^n \left\lceil \frac{b_i}{sz/m} \right\rceil \leq sz. \quad (10)$$

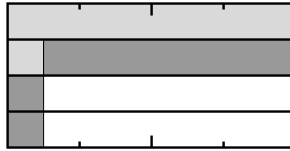


Figure 9: Schematic view of a 2048 double words cache, in which A , the light-gray block, and B , the dark-gray block, resides. A and B have $24^2 = 576$ elements each. Since they both start on the same address modulo 512 double words, the cache sets for addresses less than 64 modulo 512 double words are full, and all of C will not fit into cache.

We can however assume that the entire C matrix is not needed in cache simultaneously, since we are only reading every element in C once and writing the new value once. This can be formulated as it is sufficient if the A and B matrices fit into cache, or in other words, $M_B K_B + N_B K_B \leq csz$ in the new upper bound for M_B, N_B, K_B . With the same arguments as above, we get

$$K_B = \frac{csz}{M_B + N_B}, \quad (11)$$

which inserted into (5) gives

$$T(M_B, N_B) = MNK \cdot \left(t_{fma} + \frac{(M_B + N_B)t_{setup}}{csz} + \frac{(csz + M_B N_B)(M_B + N_B)t_{load}}{csz M_B N_B} \right). \quad (12)$$

With our test settings we get

$$T(M_B, N_B) = MNK \cdot \left(1 + \frac{7(M_B + N_B)}{2048} + \frac{10240(M_B + N_B)}{M_B N_B} \right)$$

and in Figure 10 we can see that the minimum moves towards larger block sizes compared to Figure 7.

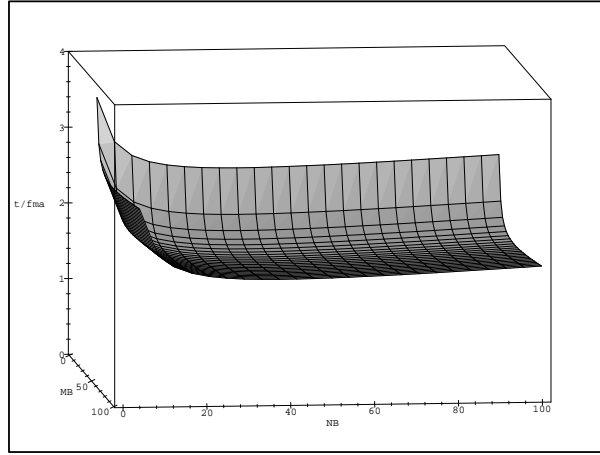


Figure 10: Cycles per double floating point operation at M_B, N_B for $t_{fma} = 1, t_{setup} = 2, t_{load} = 5, csz = 2048$, and it is assumed that C is not required to be in cache.

The minimum of $T(M_B, N_B)$ is now fulfilled by

$$M_B = N_B = \sqrt{\frac{t_{load} csz}{t_{setup} + t_{load}}}, \quad (13)$$

which leads to the new optimal values $M_B = N_B \approx 38.2, K_B \approx 26.8$ for our parameter settings. If the time is plotted against $M_B = N_B$, see Figure 11, the minimum is shown more clearly.

We remark that if A and B are of equal size and we choose their block sizes to $csz/2$, then both blocks fit into cache since from (10) we have

$$\frac{csz}{4} \sum_{i=1}^2 \left\lceil \frac{csz/2}{csz/4} \right\rceil = csz.$$

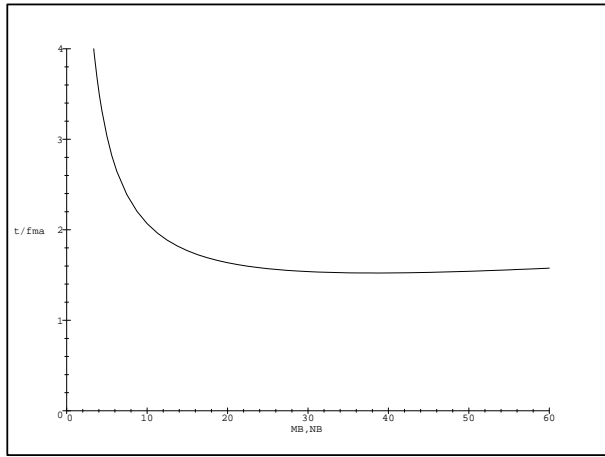


Figure 11: Cycles per double floating point operation at $M_B = N_B$ for the example $t_{fma} = 1$, $t_{setup} = 2$, $t_{load} = 5$, $csz = 2048$, and it is assumed that C is not required to be in cache.

7 Data Structures

In this section we will discuss the different ways of storing matrices, that is, where should an element be placed in memory. We have implemented routines that transform a matrix stored in ordinary format into a new, blocked format.

7.1 Traditional mappings

When storing a two-dimensional object such as a matrix into a one-dimensional object such as the memory, we are facing the problem of choosing a mapping of the different set of coordinates to each other. Traditionally, there has been two alternatives, store by row or store by column. Fortran uses the column-major order, in which columns are stored after each other, that is, we are going by *stride*⁹ 1 for increasing row coordinates, see Figure 12. In order to calculate the memory address for an element in a matrix we need the number of rows for the matrix. The address is then

$$\text{addr}(A, lda, r, c) = A + (r - 1) + (c - 1) \cdot lda \quad (14)$$

where A is the address of the first element in A and lda is the number of rows, or *leading dimension*, of A . r and c are one-based, that is, they start numbering from 1. If we are going along a row, we are going by stride lda .

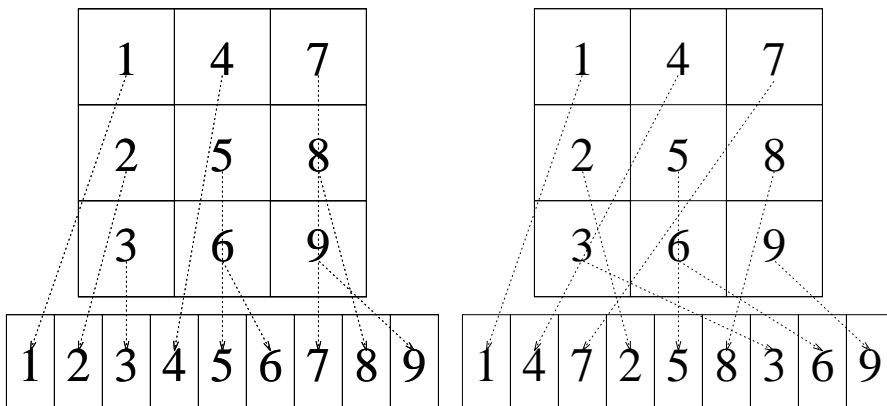


Figure 12: To the left, Fortran mapping of matrix to memory. To the right, C's way of mapping the matrix.

In C, the rows are stored after each other, that is, we are going by stride 1 if we follow the same row. In order to calculate the memory address for an element in a matrix stored in the C way, we need the number of columns for the matrix, lda ,

$$\text{addr}(A, lda, r, c) = A + (r - 1) \cdot lda + (c - 1) \quad (15)$$

Both orderings have similar technical advantages and disadvantages. However, almost every scientific computing library uses the column-major order. Fortran has three important advantages over C when writing programs for scientific computing:

⁹The distance between subsequent accesses.

- It is possible to set the leading dimension for a matrix at run-time.
- Fortran has support for complex numbers.
- Fortran has traditionally had better compilers, partly because the language has more restriction on pointers, which makes it possible for the compiler to discard aliasing.

It is also quite easy to use column-major matrices in C, by using the preprocessor and calculating the offset manually. The following example shows how to use the preprocessor to hide explicit offset calculation.

```
void fillMatrix(double *A, int lda, int r, int c, double value)
#define A(r,c) (A[(r) * lda + (c)])
{
    int i, j;

    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            A(i,j) = value;    /* Expands to A[i * lda + j] = value */
}
#undef A /* If we need another A somewhere else */
```

For further discussions of these languages, see [Met89] and [KR88].

Consider how the Fortran method effects ordinary matrix by matrix multiplication in Figure 13 when blocking for cache. We loop by I1, J1, L1, I2, J2, L2. In the innermost loop, we are accessing $A(I_1 + I_2, L_1 + L_2)$ which according to (14) corresponds to $\text{addr}(A, lda, I_1 + I_2, L_1 + L_2) = A + I_1 + I_2 + (L_1 + L_2) \cdot lda$, if the loop indices starts with 0. If $lda = csz$ – the cache size – each element in a row maps to the same congruence class in the cache. This leads to very poor cache reuse, and thus limits the length of the I2 loop to the associativity of the cache, in our case 4. This is called the “bad *lda* problem”. Many matrix multiplication libraries overcome this problem by copying the small matrices into temporary matrices before the I2, J2, L2 loops. This effect can be seen in Figure 28, in which the DATBX routine has been measured.

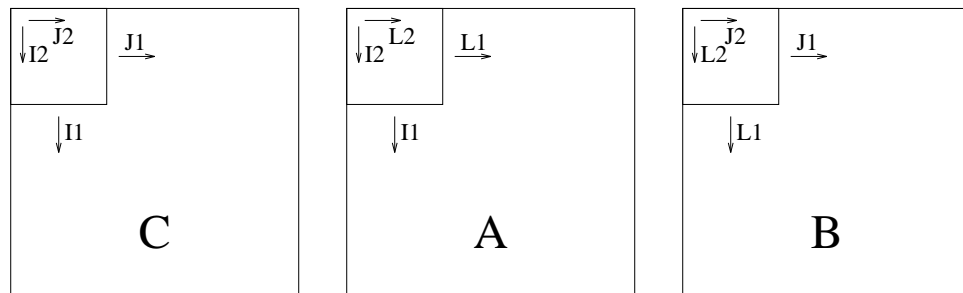


Figure 13: $C = A \cdot B$, with blocking in one level. The loop order is I1, J1, L1, I2, J2, L2, that is, I1 is the outermost loop and L2 is the innermost

7.2 Blocked formats

Fred Gustavson [Gus98b] suggests two new matrix formats, in which all matrices are stored in a format convenient for blocked operations from the beginning. In these formats, there is a one-to-one mapping between the ordinary Fortran column-major format and the new ones, block row and block column. The block column mode is shown in Figure 14.

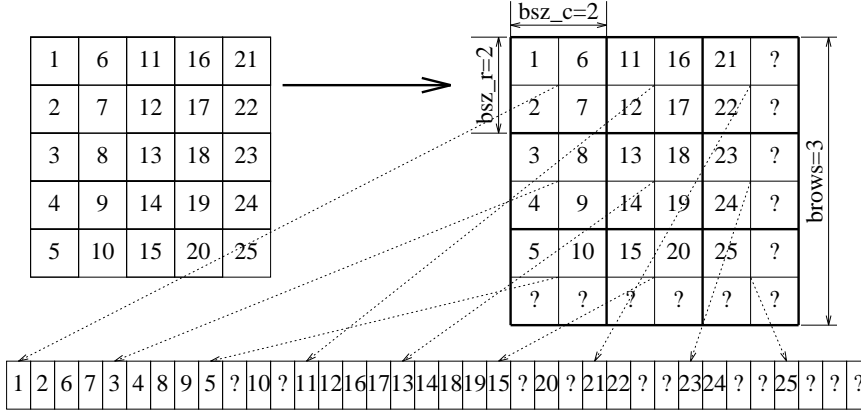


Figure 14: The mapping between the regular block column-major mode and block column format with block size equal to 2. The question marks denotes uninitialized data.

To specify an array we now need to use the quadruple $(A, brows, bsz_r, bsz_c)$ instead of the pair (A, lda) , where A is a pointer to the array, $brows$ is the height of the matrix in blocks, bsz_r and bsz_c is the size of the block submatrices. In order to calculate the address of an element in A , we use the formula

$$\begin{aligned} \text{addr}(A, brows, bsz_r, bsz_c, r, c) &= A + bsz_r bsz_c \left(\left\lfloor \frac{r-1}{bsz_r} \right\rfloor + brows \left\lfloor \frac{c-1}{bsz_c} \right\rfloor \right) + \\ &\quad + (r-1) \bmod bsz_r + ((c-1) \bmod bsz_c) bsz_r \end{aligned} \quad (16)$$

The main advantage is that the leading dimension of the blocks, $lda_b = bsz_r$ is small, much less than the cache size, which eliminates the bad leading dimension problem and keeps the memory references within a block in a narrow address space. Another advantage is that the the leading dimension is fixed. This makes the pointer arithmetic much simpler when writing kernel routines for the blocks.

There are some disadvantages however. The main problem is that from the beginning, matrices are not stored in the block format and thus have to be copied into the new format before used by routines which assumes the block format. This operation does consume cycles, but since it is a $\mathcal{O}(n^2)$ operation, we might still gain performance if the subsequent main operation is a $\mathcal{O}(n^3)$ operation, such as DGEMM. The second disadvantage is that one loses the possibility to easily express an arbitrary rectangular block of the matrix. In the column-mode, if we use $(A, lda, rows, cols)$ to select the whole matrix $A(1 : rows, 1 : cols)$, we can use $(A + (r_1 - 1) + (c_1 - 1) \cdot lda, lda, r_2 - r_1 + 1, c_2 - c_1 + 1)$ to select the rectangular block. No such equivalent transformation is possible with the block format, since the case of having the first element of the matrix the first element

of a block is a special one, and all other cases need special handling. The block column format also requires more space, but this is a minor problem since the overhead portion diminishes when problem sizes grow.

We can now abstract the block multiplication and the multiplication of matrices in block column format with equally sized square blocks ($bsz = A_{bsz_r} = A_{bsz_c} = B_{bsz_r} = B_{bsz_c} = C_{bsz_r} = C_{bsz_c}$) becomes

```
mult(M,N,K,A,a_brows,B,b_brows,C,c_crows,bsz)

DO I=0,M-1,bsz
  DO J=0,N-1,bsz
    DO L=0,K-1,bsz
      multblock(min(bsz,M-I+1),min(bsz,N-J+1),min(bsz,L-K+1),
                A((L/bsz+I/bsz*a_brows)*bsz*bsz+1),
                B((J/bsz+L/bsz*b_brows)*bsz*bsz+1),
                C((I/bsz+J/bsz*b_brows)*bsz*bsz+1),bsz);
    END DO
  END DO
END DO
```

7.3 Multilevel blocks

The algorithm above leads to good cache reuse for medium-sized problems that fits into the level 2 cache. However, for larger problems, the algorithm makes a poor job. A better algorithm for our target system, with a second level cache size of 512 kB, or 65536 double words, is the following, which reuses all data in level two cache.

```
mult(M,N,K,A,a_brows,B,b_brows,C,c_crows,bsz)

DO I1=0,M-1,bsz*6
  DO J1=0,N-1,bsz*6
    DO L1=0,K-1,bsz*6
      DO I2=I1,min(M-1,I1+6*bsz-1),bsz
        DO J2=J1,min(N-1,J1+6*bsz-1),bsz
          DO L2=L1,min(K-1,L1+6*bsz-1),bsz
            multblock(min(bsz,M-I2+1),min(bsz,N-J2+1),min(bsz,K-L2+1),
                      A((L2/bsz+I2/bsz*a_brows)*bsz*bsz+1),
                      B((J2/bsz+L2/bsz*b_brows)*bsz*bsz+1),
                      C((I2/bsz+J2/bsz*b_brows)*bsz*bsz+1),bsz);
          END DO
        END DO
      END DO
    END DO
  END DO
END DO
```

There are room for 2048 double words in the level one (on-chip) cache on the High Node. The maximum dimension of three equally sized square matrices which fit in the cache is $\lfloor \sqrt{2048/3} \rfloor = 26$. However, in order for the blocks to

fit better into level two cache we assume matrices of 24 by 24. Suppose that loading a double word from the level two cache takes 5 cycles and we can do a $(24 \times 24 \times 24)$ multiply with a speed of 200 Mflops if data is in the level one cache. Using the metrics for the IBM SP High Node, fetching the data from level two cache takes 5 cycles per double word which gives $24^2 \cdot 3 \cdot 5 \approx 8640$ cycles $\approx 77.1 \mu s$ for loading the three blocks, which gives a peak rate of

$$\frac{24^3 \cdot 2}{24^3 \cdot 2 / (200 \cdot 10^6) + 77.1 \cdot 10^{-6}} \approx 128 \text{ Mflops}$$

for problems fitting into the level two cache.

We can take this one step further. The level two cache is 64 kDW, which means that the largest matrix dimension is $\lfloor \sqrt{65536/3} \rfloor = 147$. This is why we chose 24 as the inner block size. Now we can choose $6 \cdot 24 = 144$ as the outer block size. Suppose that we can do a $(144 \times 144 \times 144)$ multiply with a speed of 119 Mflops if data is in L2 cache. Fetching a double word from memory takes an additional 15 cycles, that is $144^2 \cdot 3 \cdot 15 \approx 933000$ cycles $\approx 8.33 ms$ for loading the three blocks into L2 cache. This means a peak rate of

$$\frac{144^3 \cdot 2}{144^3 \cdot 2 / (133 \cdot 10^6) + 7.78 \cdot 10^{-3}} \approx 109 \text{ Mflops}$$

for problems fitting into the memory.

Now, assume we need to use paging, that is when the matrices do not fit in main memory, which is on 32 MDW, leading to matrices of size 3312 ($23 \cdot 144 < \sqrt{2^{25}/3}$). Our measurements shows values in the range of 600 cycles per double word for the average time to load a double word from disk. For a matrix multiply of large matrices on disk this leads to

$$\frac{3312^3 \cdot 2}{3312^3 \cdot 2 / (109 \cdot 10^6) + \frac{3312^2 \cdot 3 \cdot 600}{112 \cdot 10^6}} \approx 86.1 \text{ Mflops}$$

If the L loop is the innermost loop, we only need to touch each C -block once in each block level, however. This means that we do not gain very much by keeping the C -block in cache. In fact, we can base the maximum matrix sized on the A and B blocks only. This means that the maximum blocking size of the level one cache is $\lfloor \sqrt{2048/2} \rfloor = 32$. With the same figures as above, we get the value 140 Mflops for problems fitting in the level two cache. For the level two cache we can fit 5×5 level one blocks in level two and consequently we obtain 120 Mflops for problems fitting in the main memory.

In practice, performance may be degraded if the operating system interferes with the program or if the cache is not used in an optimal manner, for example when the addresses clash, see the Section 6.2. The performance may also decrease since we are not using all of the level two cache. If we base our calculations on level two blocks of size 180×180 instead of 160×160 , the performance figure changes from 120 to 122 Mflops.

It would be advantageous however, if we utilized the same restructuring scheme for higher levels as well. Suppose we have a matrix of size 640×640 . In the above case, the address space of the upper left 5×5 blocks at size 32×32 will not be contiguous, but instead there will be $5 \times 32 \times 32$ double words for the first

block column, then a gap of $15 \times 32 \times 32$ double word before the second column. In bad cases, we will once again encounter bad leading dimensions, not as serious as without any block restructuring, but it will still degrade performance. What we would like is another blocking level, “blocked block row”. Such a scheme would be very intricate, however, and it would still only solve the problems for one more block level. It is also necessary that the functions which operates on the matrix know how the matrix is blocked, something which might not be easy to maintain if the blocking information is implicit.

7.4 Arbitrary ordered blocked matrix format

In order to get an easier and more practical mapping scheme for matrix blocks, we defined the `blockmatrix` structure as a *struct* in C. This block structure uses the column-major format for its innermost blocks. Using a fixed structure in the innermost blocks is necessary in order to obtain efficient kernel routines. The `blockmatrix` structure consists of five attributes, shown in Table 2 and Figure 15.

<code>rows</code>	Number of rows in <i>A</i> .
<code>cols</code>	Number of columns in <i>A</i> .
<code>blocksz_r</code>	Size of each innermost block in rows.
<code>blocksz_c</code>	Size of each innermost block in columns.
<code>blockaddress</code>	$\left\lceil \frac{\text{rows}}{\text{blocksz}_r} \right\rceil \cdot \left\lceil \frac{\text{cols}}{\text{blocksz}_c} \right\rceil$ pointers to each block of size <code>blocksz_r</code> \times <code>blocksz_c</code> .

Table 2: The attributes of the `blockmatrix` record.

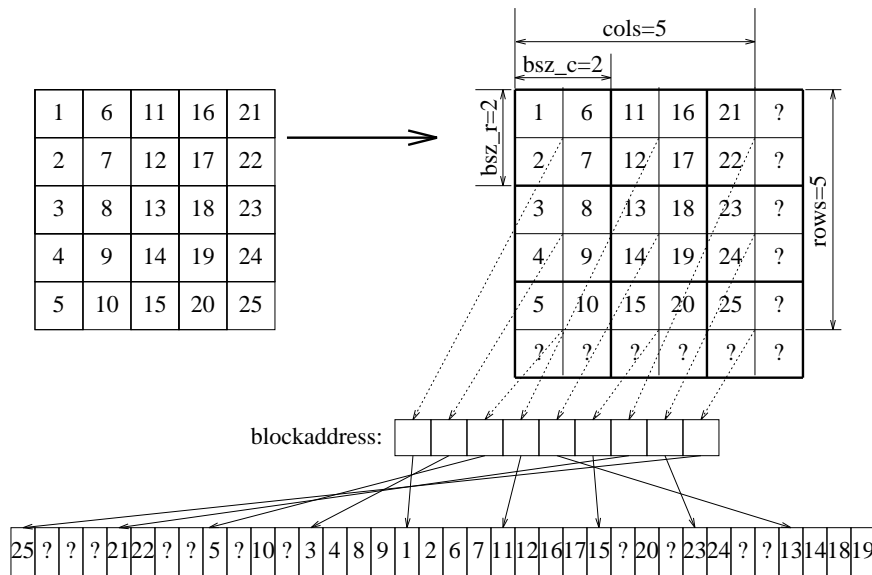


Figure 15: Using indirect addressing, we can store the blocks in any order in memory.

In our first version of the `blockmatrix` structure, there were also the at-

tributes `brows` and `bcols` which denoted the number of blocks vertically and horizontally. But since `brows` and `bcols` can be expressed as $\lceil \frac{\text{rows}}{\text{blocksz}_r} \rceil$ and $\lceil \frac{\text{cols}}{\text{blocksz}_c} \rceil$ respectively, we removed them from the record. `blockaddress` is a vector of pointers to the addresses of each block.

We have two C functions that performs basic mapping:

```
struct blockmatrix *n2b_col(double *A, int lda, int rows, int cols)
```

`n2b_col` builds a `blockmatrix` structure from a conventionally stored rectangular block of `A`, of size `rows` × `cols`. The sub-blocks are stored in column-major order and the size of the sub-blocks is fixed by the constant `SUBSIZE` (the sub-blocks are square). The memory for the record, the pointer vector and the sub-blocks are allocated in one big block, so to free the memory used by the block matrix, one only has to `free` the pointer to the `blockmatrix`.

```
void b2n(struct blockmatrix *b, double *A, int lda)
```

`b2n` copies the matrix stored in `b` from the blocked storage format back into the column-major format and stores the result in `A`. The matrix in `b` can be ordered in any way and is not altered.

Both procedures load with stride `n` and store with stride 1, since our measurements have shown, see Section 12.1, that the penalty for storing with stride is greater than loading with stride.

7.5 Recursive blocking and multiplication

However, in the general case, we have to know or measure the cache sizes to find optimal block sizes. Fred Gustavson [Gus97] shows that we can use recursion to “automatically” get several blocking levels. In order to get all the benefits from recursive multiplication, we need to store the matrices recursively as well. Fred Gustavson [Gus98a] explains a recursive data format when describing a recursive DGEMM (editorial changes have been made):

```
subroutine RBGEMM(fa, fb, m, n, k, al,
                 A, lda, B, ldb, be, C, ldc)
```

The parameters `fa`, `fb`, `m`, `n`, `k`, `al(pha)`, `be(ta)` are as before. `A = A(0:MA*KA-1)` is in recursive block format. Every call to `RBGEMM` must set `A = A(0)`. `lda` is a vector of ten entries; the last entry is an array, see Table 3.

Comments: In ordinary DGEMM one passes $A(I, L)$. Let block `nua` have coordinates $(i1, l1)$. Usually $I = i1$ and $L = l1$. If not, the user is asking to start processing somewhere in the middle of block `nua`. Hence, kernel routines should possess starting indices i, l parameters to handle this rare case; i.e., $I = i1 + i$ and $L = l1 + l$.

In our opinion, the starting positions should not be placed in the `lda`, but we believe that they should be given as separate parameters. Our prototype for a recursive DGEMM is therefore

lda(0)	<i>nua</i>	label of the block submatrix where computation starts
lda(1)	<i>i</i>	local row offset into block nua. (<i>i</i> usually is 0)
lda(2)	<i>l</i>	local col offset into block nua (<i>l</i> usually is 0)
lda(3)	<i>MA</i>	global row order of <i>A</i>
lda(4)	<i>KA</i>	global col order of <i>A</i>
lda(5)	<i>nbar</i>	row blocksize of <i>A</i>
lda(6)	<i>nbac</i>	col blocksize of <i>A</i>
lda(7)	<i>iarc</i>	<i>iarc</i> = 1, RB order is by row, see Figure 16 <i>iarc</i> = 0, RB order is by col, see Figure 17
lda(8)	<i>iant</i>	<i>iant</i> = 1, all block submatrices are stored by col ('N') <i>iant</i> = 0, all block submatrices are stored by row ('T')
lda(9)	<i>laba</i>	array of size $ma_1 \times ka_1$ listing the block numbers (<i>nua</i>) in ordinary FORTRAN order. $laba(i, j) = nua$ where $0 \leq i < ma_1$ and $0 \leq j < ka_1$. This array is to be used with array <i>lada</i> in lda(10). We choose <i>lda</i> of $laba = ma_1$.
lda(10)	<i>lada</i>	array of size $ma_1 \times ka_1$ listing the starting locations in memory of each of the $ma_1 \times ka_1$ block submatrices that make global <i>A</i> . These starting locations are offsets from <i>A</i> (0). $ma_1 = \left\lfloor \frac{MA + nbar - 1}{nbar} \right\rfloor, ka_1 = \left\lfloor \frac{KA + nbac - 1}{nbac} \right\rfloor$

Table 3: The new *lda* parameter as described by Fred Gustavson. Some editorial changes have been done.

```
void rdgemm(char fa, char fb, int m, int n, int k,
            double alpha, double beta,
            struct blockmatrix *A, int ar, int ac,
            struct blockmatrix *B, int br, int bc,
            struct blockmatrix *C, int cr, int cc);
```

Where *ar* and *ac* are offset into the *A*-matrix. Our implementation is not completed, instead a simpler version has been implemented:

```
void rdgemm_tn(struct blockmatrix *A,
              struct blockmatrix *B,
              struct blockmatrix *C)
```

which performs $C \leftarrow C + A^T \cdot B$ on the entire matrices.

The routine does a fair job while operating on matrices converted with the *n2b_col* procedure. However, to use recursive data storage we implemented the another conversion procedure:

```
struct blockmatrix *n2b_rec(double *A, int lda, int rows, int cols)
```

n2b_rec builds a blockmatrix structure of a rectangular block of *A* of size *rows*×*cols*. The blocks are stored in recursive order, with breaking of ties by rows, that is, when the number of rows and the number columns are equal, we

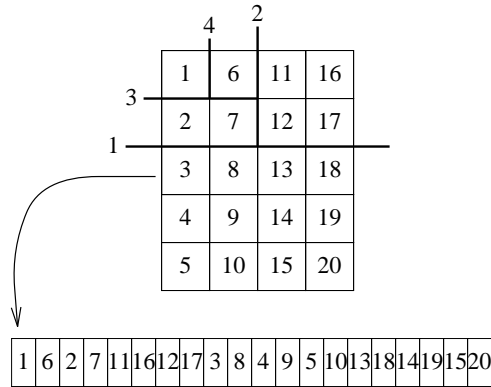


Figure 16: Mapping to recursive format. If tie, break on rows.

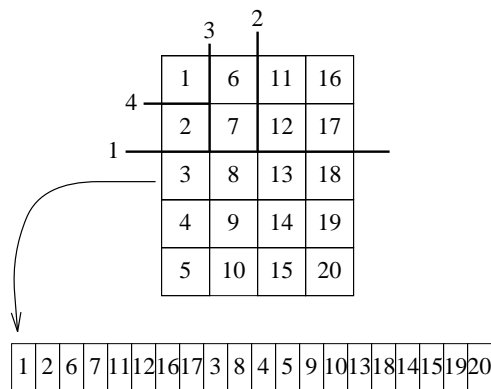


Figure 17: Mapping to recursive format. If tie, break on columns.

split the columns in half. The size of the sub-blocks is fixed by the constant `SUBSIZE` (the sub-blocks are square). The memory for the record, the pointer vector and the sub-blocks are allocated in one big block, so to free the memory used by the block matrix, one only has to `free` the pointer to the `blockmatrix`. To copy the data back to column-major order, `b2n` can be used.

8 Kernel routines

The heart of an efficient matrix multiplication routine is an efficient kernel routine, a routine that multiplies small matrices that fits into cache as fast as possible.

8.1 Peak performance

First, a small note on peak performance. In a matrix multiplying kernel, $C = C + A \cdot B$, where A is $m \times k$ and B is $k \times n$ we have mnk floating point multiplications and mnk floating point additions. The PowerPC 604 has a floating point multiply and addition instruction, FMA, that performs one double precision multiplication and add in one cycle. If several such instructions are in succession, and there is no data dependencies between them, one instruction can start every cycle, thus we may obtain $flops_{peak} = 2 \cdot \text{clock frequency}$, according to [IBM96b], where it is stated that the FMA has an execution time of one cycle.

The following example shows how to obtain peak performance:

```
CALL SECS(time1)
DO I = 1,N
  r1 = r7 * r8 + r1
  r2 = r7 * r8 + r2
  r3 = r7 * r8 + r3
  r4 = r7 * r8 + r4
  r5 = r7 * r8 + r5
  r6 = r7 * r8 + r6
END DO
CALL SECS(time2)
```

Notice that there are no memory references in the loop. The statements inside the loop corresponds to 6 FMA instructions performing 2 operations each, so that, $flops = \frac{2 \cdot 6 \cdot N}{time_2 - time_1}$. On the 112 MHz SMP system where we have made our experiments, we have achieved 223.0 Mflops, that is 99.5 % of the theoretical peak value, 224 Mflops.

The standard three DO-loops approach will make very poor use of the registers, and thus spend much too many cycles just loading data in and out of the registers. To see why this is so, consider the following code:

```
DO I = 1,M
  DO J = 1,N
    DO L = 1,K
      C(I,J) = C(I,J) + A(I,L) * B(L,J)
    END DO
  END DO
END DO
```

which compiles to the following load and store instructions:

```

DO I = 1,M
  DO J = 1,N
    DO L = 1,K
      load C(I,J) into r1
      load A(I,L) into r2
      load B(L,J) into r3
      r1 = r1 + r2 * r3
      store r1 into C(I,J)
    END DO
  END DO
END DO

```

This means that we have four load/store instructions versus one FMA. On the PowerPC 604, one load instruction can execute simultaneously with one FMA, we can at most hide one load/store instruction per FMA instruction. The integer arithmetic needed can all be expressed in the expressive RISC instructions, which means that no separate arithmetic instructions are needed. Since L is invariant in the D0-loop, there will be no cycles spent for the loop branch.

8.2 Register blocking

We also have a bad data dependency between the load B(L, J) instruction and the FMA. It takes three cycles to get to B(L, J) element, so the pipeline will stall until the FMA can start. The first observation is that *r1* is always equal to C(I, J) in the inner D0-loop, which leads to this optimization:

```

DO I = 1,M
  DO J = 1,N
    load C(I,J) into r1
    DO L = 1,K
      load A(I,L) into r2
      load B(L,J) into r3
      r1 = r1 + r2 * r3
    END DO
    store r1 into C(I,J)
  END DO
END DO

```

The compiler performs this optimization automatically when passed the parameter `-O2`. On the PowerPC 604, this code runs at 70 Mflops which means three cycles per FMA, see Table 4. Our goal is to get as close as possible to one cycle per FMA.

In order to reach higher performance, we must achieve more reuse of data in registers. This can be obtained with a method called register blocking. Assume that each loaded datum is used twice:

```

DO I = 1,M,2

```

```

DO J = 1,N,2
  load C(I ,J ) into r1
  load C(I+1,J ) into r2
  load C(I ,J+1) into r3
  load C(I+1,J+1) into r4
  DO L = 1,K
    load A(I ,L ) into r5
    load A(I+1,L ) into r6
    load B(L ,J ) into r7
    load B(L ,J+1) into r8
    r1 = r1 + r5 * r7
    r2 = r2 + r6 * r7
    r3 = r3 + r5 * r8
    r4 = r4 + r6 * r8
  END DO
  store r1 into C(I ,J )
  store r2 into C(I+1,J )
  store r3 into C(I ,J+1)
  store r4 into C(I+1,J+1)
END DO
END DO

```

Now the compiler can restructure these lines to further minimize the data dependency penalties. In Table 5 the improved results are reported. Since there are 4 loads and 4 FMAs inside the inner loop, the processor should be able to interleave these instructions. `xlf -O2` generates the following inner loop code (the option `-qtune=604` actually makes the code worse).

```

LFDU    fp5,gr10=a(gr10,192)
FMA     fp2=fp2,fp0,fp1,fcrr
LFDU    fp6,gr9=b(gr9,8)
FMA     fp3=fp3,fp1,fp5,fcrr
FMA     fp4=fp4,fp0,fp6,fcrr
LFL     fp0=a(gr10,184)
LFL     fp1=b(gr9,-184)
FMA     fp7=fp7,fp5,fp6,fcrr

```

The floating point registers are called `fp_` and the general (integer) registers are called `gr_`. LFDU is a load with an automatic increment of the indexing register, `gr9` and `gr10`. The FMA instruction uses the *floating-point unit* (see [IBM96b]) and the LFDU and LFL instruction uses the *load-store unit* and with the 4-entry load queue and the 16-entry completion unit on the PowerPC 604 the LFL pair should not impose any problems. However, the data dependencies are too close in time, i.e., the data fetched from cache will not be available when the FMA instruction will reference the data. Thus, the FMA will stall.

Our next step was to increase the register blocking. A register blocking of $x \times y$ requires $x + y + x \cdot y$ registers and will take $x + y$ loads and $x \times y$ FMAs to perform in the inner loop. Since we want to perform as many “useful” FMA instructions, as possible, as opposed to loads, we wish to get $\frac{xy}{x+y}$ as high as possible. If we disregard the integer properties of x and y , we get the maximum

of the quotient when

$$x_{\max} = y_{\max} = \sqrt{1 + R} - 1 \text{ and } \frac{x_{\max}y_{\max}}{x_{\max} + y_{\max}} = \frac{\sqrt{1 + x_{\max}} - 1}{2}$$

where R is the number of registers on the processor. The PowerPC 604 has 32 floating point registers, so by the formula $x_{\max} = y_{\max} = \sqrt{1 + 32} - 1 \approx 4.7$. Since x and y must be integers, we get $\frac{xy}{x+y} = 2$ for $x = y = 4$ which should be sufficient. We achieve an even better ratio, 2.2, for $x = 4, y = 5$, but the cumbersomeness involved with a 4×5 blocking will not give any improved performance; 4×4 blocking is sufficient. The results for some register blockings are shown in tables 6, 7, and 8.

`xlf -O2` generates the following inner loop code for the 4×4 blocking:

```

LFDU    fp24,gr12=a(gr12,192)
FMA     fp0=fp0,fp26,fp29,fc r
LFDU    fp23,gr11=b(gr11,8)
FMA     fp1=fp1,fp29,fp25,fc r
FMA     fp2=fp2,fp29,fp22,fc r
FMA     fp3=fp3,fp29,fp24,fc r
LFL     fp29=b(gr11,-568)
FMA     fp4=fp4,fp26,fp28,fc r
FMA     fp8=fp8,fp26,fp27,fc r
FMA     fp12=fp12,fp26,fp23,fc r
LFL     fp26=a(gr12,168)
FMA     fp5=fp5,fp25,fp28,fc r
FMA     fp6=fp6,fp22,fp28,fc r
FMA     fp7=fp7,fp24,fp28,fc r
LFL     fp28=b(gr11,-376)
FMA     fp9=fp9,fp25,fp27,fc r
FMA     fp10=fp10,fp22,fp27,fc r
FMA     fp11=fp11,fp24,fp27,fc r
LFL     fp27=b(gr11,-184)
FMA     fp13=fp13,fp25,fp23,fc r
LFL     fp25=a(gr12,176)
FMA     fp31=fp31,fp22,fp23,fc r
LFL     fp22=a(gr12,184)
FMA     fp30=fp30,fp24,fp23,fc r

```

This code has 8 loads and 16 FMAs. Surrounding this loop are 16 loads and 16 stores for the C matrix, and 6 loads for loading the three first elements of A and B , respectively. So, this loop for $24 \times 24 \times 24$ multiplication should run at approximately $\frac{24 \cdot 16 + 16 + 16 + 6}{24 \cdot 16} \approx 1.10$ cycles/FMA but as seen in Table 8, this level of performance is not reached. In order to measure time for the loop without the loop overhead, the following routine was developed:

```

DO I = 1,M,4
  DO J = 1,N,4
    load C(I,J) into r1
    :
  :

```

```

    load C(I+3,J+3) into r16
DO R = 1,1000
    DO L = 1,K
        load A(I,L) into r17
        :
        load A(I+3,L) into r20
        load B(L,J) into r21
        :
        load B(L,J+3) into r24
        r1 = r1 + r17 * r20
        :
        r16 = r16 + r20 * r24
    END DO
END DO
    store r1 into C(I,J)
    :
    store r16 into C(I+3,J+3)
END DO
END DO

```

Here, the `R` loops iterates the `L` loop so the inner loop now only needs to load the three first elements of `A` and `B`, so it should run at $\frac{24 \cdot 16 + 6}{24 \cdot 16} \approx 1.02$ cycles/FMA. However, for some reason it seems as if the processor chokes when the loads comes too close to each other. While tampering with the assembler listing given from `xlf -O2 -S`, we discovered a strange effect. When we compiled and linked in one pass, using `xlf -O2` we got the figures in Table 9, but when the assembler part was made separately, using `as`, performance dropped 20 %, see Table 10. In order to investigate this, the Free Software Foundation's `binutils` package was installed and the GNU assembler was used. With the GNU assembler, the performance went back to the original figures. The cause of the performance degrade with the vendor provided assembler was not found. A dump of the executables from both of the assemblers shows no difference in the assembly code. Thorough investigation showed that the vendor provided assembler lost some alignment information. This leads to misaligned storage of the matrices `A`, `B` and `C`, that is the address of the matrices is not dividable by the element size, 8 bytes. This in turn leads to double memory accesses, see [IBM96b] and hence the marginals imposed of the two FMAs per load is exceeded and the performance decreases. This goes as an important reminder – make sure that the data is aligned on double words. This is especially important if the memory for the matrices are allocated by the program itself, for example by `malloc`, instead of the compiler.

8.3 Instruction scheduling

In order to spread the loads more evenly, a program has been implemented that finds good orderings of operations. The preconditions were the following:

- `C(I,J)...``C(I+3,J+3)` are in registers.

- $A(I, K)$, $A(I+1, K)$, $B(K, J)$, $B(K, J+1)$ are already loaded into registers.

The program then finds instructions that fulfill:

1. Every third operation, starting from the first, is a load operation.
2. Each of the 8 load operations must be performed exactly once.
3. The other operations are FMAs.
4. Each of the 16 FMA operations must be performed exactly once.
5. The FMAs work on the correct data.
6. After a load, the next four FMA operations may not use the loaded data.
7. After the loop code, the registers that held $A(I, K)$, $A(I+1, K)$, $B(K, J)$, $B(K, J+1)$ must now hold $A(I, K+1)$, $A(I+1, K+1)$, $B(K+1, J)$, $B(K+1, J+1)$ so that the preconditions are fulfilled for the next iteration.

A naïve implementation will permute the operations so that condition 1 to 4 holds, and then check the other conditions. This leads to $16! \cdot 8! \approx 8 \cdot 10^{17}$ combinations. Our program instead makes a depth-first search and for each level checks that conditions 5 to 7 hold, except for the fact that data loaded at the end of the loop may not be used by the FMAs in the very beginning of the loop. This leads to a much more pruned search tree. In fact, the program finds 2113536 combinations, of which 352256 are correct, that is, they also fulfill the data dependencies over the loop. If we change the preconditions so that only three of the A and B registers are loaded at the beginning, no valid combination is found. The combination used is:

```

lfd      fp19, ((2*KBA1+0*KBA2)*8+Aoff)(r7)
fmadd   fp1, fp17, fp21, fp1
fmadd   fp2, fp17, fp22, fp2
lfd      fp20, ((3*KBA1+0*KBA2)*8+Aoff)(r7)
fmadd   fp5, fp18, fp21, fp5
fmadd   fp6, fp18, fp22, fp6
lfd      fp23, ((2*KBB1+0*KBB2)*8+Boff)(r9)
fmadd   fp9, fp19, fp21, fp9
fmadd   fp10, fp19, fp22, fp10
lfd      fp24, ((3*KBB1+0*KBB2)*8+Boff)(r9)
fmadd   fp13, fp20, fp21, fp13
fmadd   fp14, fp20, fp22, fp14
lfd      fp21, ((0*KBB1+1*KBB2)*8+Boff)(r9)
fmadd   fp3, fp17, fp23, fp3
fmadd   fp7, fp18, fp23, fp7
lfdu    fp22, ((1*KBB1+1*KBB2)*8+Boff)(r9)
fmadd   fp4, fp17, fp24, fp4
fmadd   fp8, fp18, fp24, fp8
lfd      fp17, ((0*KBA1+1*KBA2)*8+Aoff)(r7)
fmadd   fp11, fp19, fp23, fp11
fmadd   fp12, fp19, fp24, fp12
lfdu    fp18, ((1*KBA1+1*KBA2)*8+Aoff)(r7)
fmadd   fp15, fp20, fp23, fp15
fmadd   fp16, fp20, fp24, fp16

```

The `fp` register numbers agree with the `r` register numbers above. `r7` is a pointer into `A` and `r9` is a pointer into `B`. `KBA1`, `KBA2`, `KBB1` and `KBB2` are set according to which kind of transpose operations that are used. This loop runs without stalls, the performance results can be seen in Table 11.

8.4 Prefetching iterations

In order to diminish the cache miss penalties and stalls of the processor pipeline, prefetching iterations can be used. By prefetching iterations we mean the concept of initializing a data load for the innermost loop's next iteration from the cache before the data is actually loaded by the processor. This kind of prefetching is called *next-sequential prefetching* by Charney and Puzak in [CP97]. Agarwal, Gustavson, and Zubair shows, see [AGZ94], that performance can be improved for a DGEMV operation by 20 %. In [IBM96b], it is shown how prefetching can be used to improve performance for element-wise summation of vectors.

Prefetching can be introduced in the executable in two ways. The first way is by the compiler, which recognizes the data access pattern and inserts appropriate prefetching instructions. This is rarely the case, since this data pattern analysis is very difficult. The second way is when prefetching instructions are inserted explicitly in the source code, Agarwal, Gustavson, Zubair call this *algorithmic prefetching*. A simple example is the following:

```
DO I = 1,100
  dummy = A(I + 4)
  SUM = SUM + A(I)
END DO
```

In the code above, `dummy` is loaded but the value is discarded. However, if the processor finish load queue, that is, the queue which holds pending load, is large enough the load serves as a cache initializer and the process of loading `A(I+4)` is started though the processor can continue with the load of `A(I)` and the addition. After four iterations the value of `A(I)` should have been brought into cache without stalls. However, we have not been able to make such compiler hints to work. The IBM compiler, `xlf` ([IBM97b]), effectively suppresses our attempts to make dummy loads, although it is possible to insert prefetching instruction in the assembler code. There are two methods to invoke prefetching. The first mimics the original Fortran code by inserting load instructions. The second method is to use the `dcbt` instruction, which anticipates the load of a cache line, see [IBM96a]. The new assembler code follows:

```
lfd      fp19,((2*KBA1+0*KBA2)*8+Aoff)(r7)
fmadd   fp1,fp17,fp21,fp1
/touch/ ((3*KBA1+0*KBA2)*8+Aoff+KBA2*8*PRELEVEL)(r7)
fmadd   fp2,fp17,fp22,fp2
lfd      fp20,((3*KBA1+0*KBA2)*8+Aoff)(r7)
fmadd   fp5,fp18,fp21,fp5
/touch/ ((2*KBB1+0*KBB2)*8+Boff+KBB2*8*PRELEVEL)(r9)
fmadd   fp6,fp18,fp22,fp6
lfd      fp23,((2*KBB1+0*KBB2)*8+Boff)(r9)
fmadd   fp9,fp19,fp21,fp9
/touch/ ((3*KBB1+0*KBB2)*8+Boff+KBB2*8*PRELEVEL)(r9)
```

```

fmadd    fp10,fp19,fp22,fp10
lfd      fp24,((3*KBB1+0*KBB2)*8+Boff)(r9)
fmadd    fp13,fp20,fp21,fp13
/touch/  ((0*KBB1+1*KBB2)*8+Boff+KBB2*8*PRELEVEL)(r9)
fmadd    fp14,fp20,fp22,fp14
lfd      fp21,((0*KBB1+1*KBB2)*8+Boff)(r9)
fmadd    fp3,fp17,fp23,fp3
/touch/  ((1*KBB1+1*KBB2)*8+Boff+KBB2*8*PRELEVEL)(r9)
fmadd    fp7,fp18,fp23,fp7
lfdu     fp22,((1*KBB1+1*KBB2)*8+Boff)(r9)
fmadd    fp4,fp17,fp24,fp4
/touch/  ((0*KBA1+1*KBA2)*8+Aoff+KBA2*8*PRELEVEL)(r7)
fmadd    fp8,fp18,fp24,fp8
lfd      fp17,((0*KBA1+1*KBA2)*8+Aoff)(r7)
fmadd    fp11,fp19,fp23,fp11
/touch/  ((1*KBA1+1*KBA2)*8+Aoff+KBA2*8*PRELEVEL)(r7)
fmadd    fp12,fp19,fp24,fp12
lfdu     fp18,((1*KBA1+1*KBA2)*8+Aoff)(r7)
fmadd    fp15,fp20,fp23,fp15
/touch/  ((2*KBA1+0*KBA2)*8+Aoff+KBA2*8*PRELEVEL)(r7)
fmadd    fp16,fp20,fp24,fp16

```

where `/touch/` is a placeholder for the prefetching instruction. In order to get an effective prefetching, it is important that the touched data address is sufficiently ahead of the real load address. In order to test the effect of the prefetching, a $120 \times 120 \times 120$ matrix multiply was executed by calling the respective $24 \times 24 \times 24$ kernel routine for the 5×5 block matrices. Each multiplication ran 20 times for 20 seconds, and the average of the 15 best values were used. The results are shown in Table 12. As we can see, the use of load operation to prefetch data actually degrades performance, due to the fact that the load queues are too short to accommodate all pending loads. The use of the `dcbt` instruction does not seem to increase performance, the rates are very similar to the rates of the code without prefetching. Prefetching is an adequate mean to hide latencies. However, it cannot be used to overcome poor memory bandwidth.

8.5 Prefetching matrices

Observe that while performing a kernel multiplication taking $\mathcal{O}(n^3)$ time, we only need to prefetch the new matrices for the next multiplication, which takes $\mathcal{O}(n^2)$ time. Our first attempt was to prefetch each element in the A and B matrices for the next multiplication. Unfortunately, this leads to congestion of the `dcbt`s, and performance is degraded. Instead, we noticed it is only necessary to touch one element on each cache line. Therefore, our best implementation, unrolls the innermost loop four times and touches the fourth element of the A , B and C matrices one time during this 64 FMA long loop. This makes room in time for the `dcbt`s. However, we are not getting full speed, which can be seen in Table 12. This might be due to the fact that there are some cases where the old and the new matrices will not fit into cache. The best results are found with blocking sizes 16×16 . We believe this is due to the fact that this small blocking

fits the new A and B matrices into on-chip cache more frequently than 20×20 , see Figure 18. The loops for the $160 \times 160 A^T B$ problem are shown below, notice how ii , jj , ll are holding the next matrix blocks to be processed. The `dab_prefetch` routine takes six arguments, the last three containing the blocks to be prefetched.

```

#define BSZ 10 /* Outer number of blocks */
#define SZ 16 /* Inner block size */

double A[BSZ][BSZ][SZ][SZ], B[BSZ][BSZ][SZ][SZ], C[BSZ][BSZ][SZ][SZ];

...

for (i = 0; i < BSZ; i++)
  for (j = 0; j < BSZ; j++)
    for (l = 0; l < BSZ; l++) {
      int ii = i, jj = j, ll = l;
      if (++ll == BSZ) {
        ll = 0;
        if (++jj == BSZ) {
          jj = 0;
          if (++ii == BSZ)
            ii = 0;
        }
      }
      /* Using column-major mode */
      dab_prefetch(A[k][i], B[k][j], C[j][i],
                  A[kk][ii], B[kk][jj], C[jj][ii]);
    }

```

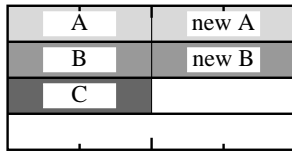


Figure 18: Fitting three 16×16 blocks in cache and still having room for two new blocks. In the $C = A^T B$ case, the next A and B blocks are following directly after the current A and B blocks. In $C = AB$, there is still room for a new A block occupying the same congruence class as the current A .

Routines that multiplies arbitrary ordered block matrices the normal way, with blocking for level two cache:

```

void mm_tn(struct blockmatrix *A,
          struct blockmatrix *B,
          struct blockmatrix *C)

```

which performs $C \leftarrow C + A^T \cdot B$ on the entire matrices using the non-prefetching kernel.

```
void mm_tnx(struct blockmatrix *A,  
            struct blockmatrix *B,  
            struct blockmatrix *C)
```

which performs $C \leftarrow C + A^T \cdot B$ on the entire matrices using the prefetching kernel.

M, N	K																			
	1	2	4	8	12	16	20	24	28	32	36	40	56	60	64	96	250	500	1000	
1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	3.06	3.04	3.05
2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	3.05	3.05	-
4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
8	-	-	-	-	-	-	-	-	3.21	3.18	3.19	3.15	3.13	3.10	3.09	3.11	3.08	-	-	-
12	-	-	-	-	3.39	3.29	3.24	3.20	3.17	3.15	3.14	3.13	3.10	3.11	3.10	3.53	-	-	-	-
16	-	-	-	3.56	3.42	3.32	3.23	3.20	3.19	3.17	3.14	3.13	3.11	3.65	4.11	5.31	-	-	-	-
20	5.89	5.64	4.08	3.55	3.37	3.28	3.23	3.19	3.17	3.16	3.15	3.14	3.47	3.63	3.85	-	-	-	-	-
24	5.84	5.63	4.08	3.55	3.37	3.28	3.24	3.21	3.19	3.22	3.30	3.38	3.70	4.10	4.60	-	-	-	-	-
28	5.80	5.61	4.06	3.55	3.37	3.29	3.24	3.30	3.48	3.51	3.44	3.43	4.27	-	-	-	-	-	-	-
32	6.18	5.76	4.20	3.62	3.42	3.33	4.16	4.73	4.71	4.73	4.78	4.66	-	-	-	-	-	-	-	-
36	5.75	5.60	4.07	3.55	3.55	3.72	3.67	3.55	3.49	3.53	3.69	-	-	-	-	-	-	-	-	-
40	5.74	5.61	4.08	3.93	4.01	3.80	3.66	3.57	3.88	-	-	-	-	-	-	-	-	-	-	-
56	12.80	8.55	5.65	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
60	12.81	8.55	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 4: Average cycles per FMA for the three D0-loops algorithm, and register blocking of C . Each test case was run three times, and the mean of the two best runs is reported. Only problems where the sum of sizes the matrices is between 400 and 4000 double words are reported.

M, N	K																			
	1	2	4	8	12	16	20	24	28	32	36	40	56	60	64	96	250	500		
2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1.53	1.52	-	-
4	-	-	-	-	-	-	-	-	-	-	-	-	1.57	1.57	1.57	1.54	1.53	-	-	-
8	-	-	-	-	-	-	-	1.63	1.61	1.61	1.59	1.58	1.56	1.56	1.56	1.55	-	-	-	-
12	-	-	-	-	1.73	1.68	1.64	1.62	1.61	1.59	1.58	1.58	1.56	1.56	1.56	2.10	-	-	-	-
16	-	-	-	1.84	1.75	1.69	1.64	1.62	1.61	1.60	1.59	1.58	1.57	1.86	2.16	2.98	-	-	-	-
20	3.84	2.82	2.16	1.84	1.73	1.67	1.64	1.62	1.61	1.60	1.59	1.58	1.90	2.04	2.15	-	-	-	-	-
24	3.83	2.82	2.17	1.84	1.73	1.68	1.65	1.63	1.61	1.66	1.74	2.07	1.98	2.18	2.43	-	-	-	-	-
28	3.81	2.81	2.16	1.84	1.73	1.68	1.64	1.71	1.87	1.92	1.91	1.95	2.46	-	-	-	-	-	-	-
32	3.81	2.92	2.22	1.86	1.75	1.70	2.30	2.74	2.70	2.70	2.83	2.71	-	-	-	-	-	-	-	-
36	3.77	2.81	2.16	1.84	1.91	2.15	2.12	2.04	1.98	2.11	2.27	-	-	-	-	-	-	-	-	-
40	3.76	2.81	2.17	2.27	2.47	2.25	2.12	2.02	2.13	-	-	-	-	-	-	-	-	-	-	-
56	10.59	7.02	4.32	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
60	10.57	7.56	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 5: 2×2 register blocking.

M, N	K																			
	1	2	4	8	12	16	20	24	28	32	36	40	56	60	64	96	250			
4	-	-	-	-	-	-	-	-	-	-	-	-	1.31	1.30	1.31	1.29	1.27	-	-	-
8	-	-	-	-	-	-	-	1.37	1.35	1.34	1.33	1.32	1.30	1.30	1.30	1.29	-	-	-	-
12	-	-	-	-	1.46	1.42	1.38	1.36	1.35	1.34	1.33	1.32	1.30	1.30	1.30	2.02	-	-	-	-
16	-	-	-	1.56	1.47	1.42	1.38	1.36	1.35	1.34	1.33	1.32	1.31	1.51	1.67	2.23	-	-	-	-
20	3.71	2.48	1.87	1.56	1.46	1.41	1.38	1.36	1.35	1.34	1.33	1.32	1.72	1.84	1.93	-	-	-	-	-
24	3.70	2.48	1.86	1.56	1.46	1.41	1.38	1.36	1.35	1.41	1.52	1.60	1.85	1.96	2.04	-	-	-	-	-
28	3.68	2.47	1.86	1.56	1.46	1.42	1.39	1.46	1.65	1.80	1.85	1.82	2.07	-	-	-	-	-	-	-
32	3.68	2.51	1.88	1.57	1.47	1.43	1.81	2.06	2.04	2.04	2.14	2.10	-	-	-	-	-	-	-	-
36	3.66	2.47	1.86	1.57	1.65	1.87	1.85	1.85	1.88	1.94	1.97	-	-	-	-	-	-	-	-	-
40	3.66	2.48	1.87	1.98	2.14	2.25	1.87	1.86	1.90	-	-	-	-	-	-	-	-	-	-	-
56	10.83	6.04	3.70	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
60	10.80	6.03	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 6: 4×2 register blocking.

M, N	K																
	1	2	4	8	12	16	20	24	28	32	36	40	56	60	64	96	
12	-	-	-	-	1.39	1.33	1.30	1.27	1.26	1.25	1.24	1.23	1.22	1.22	1.21	1.91	
24	3.45	2.37	1.78	1.47	1.37	1.32	1.30	1.28	1.27	1.33	1.43	1.51	1.76	1.85	1.95	-	
36	3.43	2.37	1.77	1.48	1.56	1.77	1.75	1.75	1.77	1.84	1.89	-	-	-	-	-	
60	10.65	5.93	-	-	-	-	-	-	-	-	-	-	-	-	-	-	

Table 7: 4×3 register blocking.

M, N	K																
	1	2	4	8	12	16	20	24	28	32	36	40	56	60	64	96	250
4	-	-	-	-	-	-	-	-	-	-	-	-	1.12	1.11	1.11	1.10	1.08
8	-	-	-	-	-	-	-	1.17	1.16	1.15	1.14	1.13	1.12	1.11	1.11	1.10	-
12	-	-	-	-	1.27	1.23	1.19	1.17	1.16	1.14	1.14	1.13	1.11	1.11	1.11	1.46	-
16	-	-	-	1.37	1.27	1.22	1.19	1.17	1.15	1.14	1.14	1.13	1.12	1.26	1.38	1.76	-
20	3.33	2.26	1.67	1.37	1.27	1.23	1.19	1.17	1.16	1.15	1.14	1.13	1.41	1.44	1.44	-	-
24	3.33	2.26	1.66	1.37	1.27	1.22	1.19	1.17	1.16	1.21	1.30	1.36	1.41	1.46	1.49	-	-
28	3.31	2.25	1.66	1.37	1.27	1.22	1.19	1.26	1.43	1.48	1.46	1.44	1.51	-	-	-	-
32	3.30	2.26	1.66	1.37	1.27	1.22	1.41	1.54	1.55	1.60	1.64	1.64	-	-	-	-	-
36	3.30	2.25	1.67	1.37	1.45	1.67	1.65	1.57	1.52	1.48	1.49	-	-	-	-	-	-
40	3.31	2.26	1.67	1.79	1.97	1.76	1.64	1.57	1.54	-	-	-	-	-	-	-	-
56	10.40	6.00	3.55	-	-	-	-	-	-	-	-	-	-	-	-	-	-
60	10.42	5.98	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 8: 4×4 register blocking.

M, N	K																
	1	2	4	8	12	16	20	24	28	32	36	40	56	60	64	96	250
4	-	-	-	-	-	-	-	-	-	-	-	-	1.08	1.08	1.07	1.07	1.07
8	-	-	-	-	-	-	-	1.08	1.08	1.08	1.08	1.08	1.08	1.07	1.07	1.07	-
12	-	-	-	-	1.10	1.09	1.09	1.08	1.08	1.08	1.08	1.08	1.08	1.08	1.08	1.07	1.07
16	-	-	-	1.12	1.10	1.09	1.09	1.08	1.08	1.08	1.08	1.08	1.08	1.08	1.08	1.08	-
20	1.18	1.26	1.16	1.12	1.10	1.09	1.09	1.08	1.08	1.08	1.08	1.08	1.08	1.08	1.08	1.08	-
24	1.18	1.26	1.16	1.12	1.10	1.09	1.09	1.08	1.08	1.08	1.08	1.08	1.08	1.37	1.08	-	-
28	1.18	1.26	1.16	1.12	1.10	1.09	1.09	1.09	1.08	1.08	1.08	1.08	1.08	-	-	-	-
32	1.18	1.26	1.16	1.12	1.10	1.09	1.09	1.09	1.08	1.08	1.08	1.08	-	-	-	-	-
36	1.18	1.26	1.16	1.12	1.10	1.09	1.09	1.09	1.08	1.08	1.08	-	-	-	-	-	-
40	1.18	1.26	1.16	1.12	1.10	1.09	1.09	1.08	1.08	-	-	-	-	-	-	-	-
56	1.19	1.26	1.16	-	-	-	-	-	-	-	-	-	-	-	-	-	-
60	1.19	1.26	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 9: 4×4 register blocking, but with repeated K -loop.

M, N	K																
	1	2	4	8	12	16	20	24	28	32	36	40	56	60	64	96	250
4	-	-	-	-	-	-	-	-	-	-	-	-	1.27	1.27	1.27	1.26	1.27
8	-	-	-	-	-	-	-	1.28	1.28	1.28	1.27	1.27	1.27	1.27	1.27	1.27	-
12	-	-	-	-	1.31	1.30	1.29	1.28	1.28	1.28	1.27	1.27	1.27	1.27	1.27	1.27	-
16	-	-	-	1.34	1.31	1.30	1.29	1.28	1.28	1.28	1.27	1.27	1.27	1.27	1.27	1.27	-
20	1.34	1.57	1.41	1.34	1.31	1.30	1.29	1.28	1.28	1.28	1.28	1.27	1.27	1.27	1.27	-	-
24	1.34	1.57	1.42	1.34	1.31	1.30	1.29	1.28	1.28	1.28	1.28	1.27	1.27	1.27	1.27	-	-
28	1.34	1.57	1.41	1.34	1.31	1.30	1.29	1.28	1.28	1.28	1.28	1.27	1.27	-	-	-	-
32	1.34	1.57	1.42	1.34	1.31	1.30	1.29	1.28	1.28	1.28	1.28	1.27	-	-	-	-	-
36	1.34	1.58	1.42	1.34	1.31	1.30	1.29	1.29	1.28	1.28	1.28	-	-	-	-	-	-
40	1.34	1.58	1.42	1.34	1.31	1.30	1.29	1.29	1.28	-	-	-	-	-	-	-	-
56	1.35	1.58	1.42	-	-	-	-	-	-	-	-	-	-	-	-	-	-
60	1.35	1.58	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 10: Same as Table 9, but with misaligned data.

M, N	K												
	8	12	16	20	24	28	32	36	40	56	60	64	96
8	-	-	-	-	1.14	1.12	1.10	1.09	1.08	1.06	1.06	1.06	1.04
12	-	1.24	1.18	1.14	1.12	1.10	1.09	1.08	1.08	1.06	1.05	1.05	1.33
16	1.34	1.23	1.17	1.14	1.12	1.10	1.09	1.08	1.07	1.08	1.17	1.25	1.54
20	1.33	1.22	1.17	1.14	1.11	1.10	1.09	1.08	1.09	1.32	1.33	1.35	-
24	1.32	1.22	1.17	1.14	1.12	1.12	1.17	1.26	1.35	1.39	1.59	1.40	-
28	1.32	1.22	1.17	1.14	1.24	1.34	1.45	1.38	1.39	1.46	-	-	-
32	1.32	1.22	1.20	1.41	1.53	1.46	1.44	1.39	1.42	-	-	-	-
36	1.33	1.43	1.62	1.61	1.51	1.44	1.43	1.46	-	-	-	-	-
40	1.77	1.90	1.72	1.59	1.50	1.47	-	-	-	-	-	-	-

Table 11: Our assembler-made kernel 4×4 register blocking. To be compared to Table 8

Prefetching technique	Mflops
dcbt, 1 iteration in advance	131.6
dcbt, 2 iterations in advance	131.7
dcbt, 3 iterations in advance	129.2
dcbt, 4 iterations in advance	130.2
dcbt, 5 iterations in advance	130.8
dcbt, 6 iterations in advance	129.9
dcbt, 7 iterations in advance	130.7
dcbt, 8 iterations in advance	128.6
loading, 3 iterations in advance	115.4
dcbt on next matrices, problem size $128 \times 128 \times 128$, blocking on 16×16	164.0
dcbt on next matrices, problem size $160 \times 160 \times 160$, blocking on 16×16	160.8
dcbt on next matrices, problem size $192 \times 192 \times 192$, blocking on 16×16	152.0
dcbt on next matrices, problem size $224 \times 224 \times 224$, blocking on 16×16	151.7
dcbt on next matrices, problem size $256 \times 256 \times 256$, blocking on 16×16	147.7
dcbt on next matrices, problem size $160 \times 160 \times 160$, blocking on 20×20	144.4
dcbt on next matrices, problem size $168 \times 168 \times 168$, blocking on 12×12	140.9
without prefetching	134.0
running on the same 16×16 block	196.8
running on the same 24×24 block	199.1

Table 12: Comparison of different prefetching techniques

9 Cache simulation tool

In order to test theoretical blocking ideas of a theoretical model of a cached memory hierarchy, we have developed a cache simulation and visualization tool. This tool helps calculating the expected number of cache misses. However, the mapping of virtual addresses to physical addresses is absent. This means that the results produced are only *one* possible outcome on a physical machine, although it may give a hint whether or not a given algorithms is efficient. One disadvantage of the simulation tool is that it is very slow, on the other hand, one only needs to run a simulation once, since there are no fluctuations due to the machine's workload.

9.1 Cache model

A quadruple is used to describe a cache ([PH94], [IBM94]):

$$Q = (n, w, s, a) \begin{cases} n & \text{number of cache lines} \\ w & \text{width of each cache line} \\ s & \text{the number of splits for each cache line} \\ a & \text{the associativity (number of congruence classes)} \end{cases}$$

The size of the cache is nw bits, or $nw/8$ bytes or $nw/64$ double precision floating point numbers. For each address mapped into the cache, there are a different places to store the data of the address. For each congruence class of cache lines, there is a mechanism which decides which cache lines that will be replaced when a cache miss occurs. Many caches, both real ones, and our virtual cache module, uses the least recently used (LRU) algorithm for determining the lines to be replaced. The LRU is represented with each cache line having a time stamp. When the cache line is used, its time stamp is updated with the current time.

In some architectures, a cache line may be split into two or more parts, in order to make small loads more efficient. On one hand, one want large line widths because it is an easy and cheap way to make large caches since less address comparators are needed. On the other hand, one does not want to pay the penalty of bringing a large line into cache in order to read just a few bytes. This is where the splitting of the cache lines comes in. Each cache line uses one address comparator, but one can load separate parts of the cache line, hereafter each such split line is called a cache segment.

We assume that w and s are powers of two and that the line width is at least a byte, that is, $w = 2^{W+3}$, $s = 2^S$ for non-negative integers W and S . We also assume that n over a is a power of two, $\frac{n}{a} = 2^C$ for non-negative C . Let p be the number of bits used to specify an address. Now, the algorithm for loading the data of an address X into cache is as follows:

1. Let $X = (x_p x_{p-1} \cdots x_0)$
2. Let $g = (x_{W+C-1} \cdots x_W)$ (the congruence class)
3. If there exist an $i : 0 \leq i < a$ such that the address tag of cache line $ag + i$ is $(x_p x_{p-1} \cdots x_W \underbrace{0 \cdots 0}_{W-1 \text{ zeroes}})$, then the we have found a cache line with

the correct address tag, and the algorithm continues at 8 with I set to the corresponding i .

4. For each of cache lines $ag + i, 0 \leq i < a$, find the one with the earliest time stamp and let I be the corresponding i .
5. Set the address tag of the cache line $ag + I$ to $(x_p x_{p-1} \cdots x_W \underbrace{0 \cdots 0}_{W-1})$
6. Invalidate all segments of cache line $ag + I$.
7. Update the time stamp of cache line $ag + I$ with the current time.
8. For the cache line $ag + I$, check whether the segment $(x_{W-1} \cdots x_{W-S})$ is invalid. If so, we have a cache miss and we need to load the data from further down in the memory hierarchy. Count one cache miss and mark the segment as loaded.

In the emulation of the SMP system, we are lacking one important data, the virtual memory mapping. All references into memory are made using virtual addresses, which are translated into physical addresses by the operating system with the aid of TLB. Since all caches in the SMP system are using physical addresses, we cannot exactly mimic the real scenario, but as an approximation we are using the virtual addresses instead of the physical addresses.

In order to simulate the load of an address into a register on the SMP system, the following algorithm is used:

1. Load the address into TLB.
2. Load the address into level one cache.
3. If the address was not in level one cache, load it into level two cache.

Notice that the cost for a cache or TLB hit is none, that is we assume that the instructions fetching data that are already in cache will take nominal time.

9.2 Simulation and visualization tool

The cache simulation and visualization tool is a set of routines that keeps track of a simulated cache. These routines are structured in three levels.

The first level handles general cache tracking, and has three functions:

```
cache *setup_cache(int cachelines, int linewidth,
                  int splits, int assoc)
```

The `setup_cache` routine builds a cache structure of the quadruple given as argument, which matches the cache model described above.

```
int load_into_cache(int address, memory *m, cache *c)
```

The `load_into_cache` routine loads the data of an address `address` into cache `c`. The `m` argument is not used, and should be set to `NULL`. The routine returns zero if the data of the address already was in cache, otherwise non-zero.

```
void free_cache(cache *c)
```

The `free_cache` frees memory allocated by `setup_cache`.

The `cache` structure consists of book-keeping attributes, although one attribute is important for the user. The `misses` attribute is a counter which holds the total number of cache misses.

At the second level, a TLB and two levels of cache is assumed. These parts, which are pointers to `cache` structures, must be initialized by the user. For example, the following lines sets up a 16 kB four-way associate level one cache, a 512 kB direct mapped level two cache and a 2×64 entry TLB, with a page size of 4096 bytes.

```
cache_l1 = setup_cache(128*4, 32*8, 1, 4);
cache_l2 = setup_cache(16384, 32*8, 1, 1);
cache_tlb = setup_cache(128, 4096, 1, 2);
```

When initialized, the user program can use the `load` function to bring the data of an address into the simulated cache model. The declaration of `load` is

```
int load(int address);
```

`load` returns zero if the address was in TLB and the data was in level one cache, otherwise non-zero is returned. Notice that the load address is compatible with Fortran, since Fortran uses call-by-address. For example, consider the following Fortran code:

```
DO I = 1,M
  DO J = 1,N
    DO L = 1,K
      C(I,J) = C(I,J) + A(I,K) * B(K,J)
    END DO
  END DO
END DO
```

We assume that the cost of a store is the same as the cost of the load, so no special store function is needed. To simulate cache misses, three lines are added (the original multiplication is not necessary for cache miss measurements):

```
DO I = 1,M
  DO J = 1,N
    DO L = 1,K
      CALL LOAD(A(I,K))
      CALL LOAD(B(K,J))
      CALL LOAD(C(I,J))
      C(I,J) = C(I,J) + A(I,K) * B(K,J)
    END DO
  END DO
END DO
```

or, more realistically, after the compiler's optimizations:

```
DO I = 1,M
  DO J = 1,N
    CALL LOAD(C(I,J))
    DO L = 1,K
```

```

        CALL LOAD(A(I,K))
        CALL LOAD(B(K,J))
        C(I,J) = C(I,J) + A(I,K) * B(K,J)
    END DO
    CALL LOAD(C(I,J))
END DO
END DO

```

The third level includes a visual interface for X Windows. It is only a first implementation and the interface is somewhat patched for the IBM SP system. For each operation, the user is supposed to increment a counter, `cache_T`. For the X Windows system, the following routines have been implemented:

```
void xmain(void (*f)(void *), void *data)
```

`xmain` sets up the graphical interface and then calls the function `f` with the argument `data`.

```
void yieldW(void)
```

The user function should call `yieldW` with regular intervals in order to update the window. The output, which can be seen in Figure 19, is made by running the following program:

```

#include <Hierarchy/system.h>
#include <Hierarchy/xmain.h>

/* Matrices' sizes */
#define SZ 144

/* Create dummy addresses */
#define A(r,c) (((c) * SZ + (r)) * 8)
#define B(r,c) (((c) * SZ + (r) + SZ*SZ) * 8)
#define C(r,c) (((c) * SZ + (r) + 2*SZ*SZ) * 8)

/* Our matrix multiply function, which is called by xmain */
void matmult(void *data)
{
    int i, j, k;

    for (i = 0; i < SZ; i++)
        for (j = 0; j < SZ; j++) {
            load(C(i,j));
            for (k = 0; k < SZ; k++) {
                load(A(k,i));
                load(B(j,k));
                if (!(cache_T++ % 10000)) /* Increment cache_T */
                    yieldW(); /* Update window each 40000 operation */
            }
            load(C(i,j));
        }
}

```

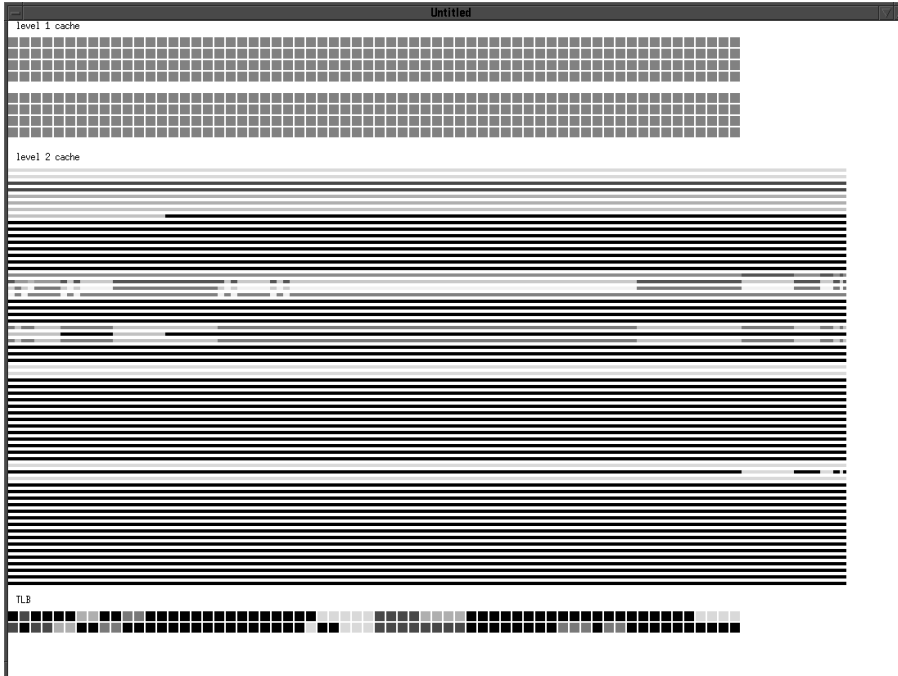


Figure 19: A screen-shot of a matrix multiplication, showing the four-way associative level one cache, the direct mapped level two cache, and the two-way associative TLB. The coloring may be distorted in the figure.

```

}

main(int argc, char **argv)
{
    /* Sets up caches */
    cache_l1 = setup_cache(128*4, 32*8, 1, 4);
    cache_l2 = setup_cache(16384, 256, 1, 1);
    cache_tlb = setup_cache(128, 4096*8, 1, 2);

    xmain(matmult, 0); /* Start cache simulation process.
                       matmult does not use its argument,
                       so we send NULL */

    return 0;
}

```

Each cache segment is colored using the following algorithm:

- Each time there is a cache hit, a cache hit counter for the respective segment is incremented.
- Each time there is a cache miss, the cache hit counter and a cache miss counter for the respective segment is incremented.
- Each time `yieldW` is invoked, the following algorithm is used:

1. $f_{misses}(i) \leftarrow 1 + f_{misses}(i) \cdot r^{p(T-T_0)}$ for all cache segments (i), where $T = \text{cache_T}$.
2. $f_{hits}(i) \leftarrow 1 + f_{hits}(i) \cdot r^{p(T-T_0)}$ for all cache segments (i).
3. $T_0 \leftarrow T$
4. Each cache segment, i , is then colored by scaling $f_{misses}(i)$ and $f_{hits}(i)$ into the ranges 0 to 10. The more cache hits, the more green becomes the square which belong the the cache segment. The more cache misses, the more red, which means that a cache segment that is not used for a long time turns black. If a cache segment is unused, or invalid, it is blue. (All cache segments are unused at the beginning. A cache segment may get invalidated if the cache line has splits, and another cache segment corresponding to the same cache line needs to be loaded with data from another address.

The parameters r and p and constants which should be adjusted for the cache sizes and use, in order to get comparable views. For the time being, they are hard-coded for the SP system.

After the user function returns, `xmain` displays the accumulated misses for the two caches and the TLB.

9.3 Some experimental results

1. $144 \times 144 \times 144$ matrix multiplication, three loops.

Number of operations (flops/2)	2985984
Level one cache misses	2458548
Level two cache misses	15552
TLB misses	122

All data fits into level two cache and TLB, therefore the misses corresponds to the loading of the data once ($\lceil \frac{8 \cdot 144^2 \cdot 3}{32} \rceil = 15552$ and $\lceil \frac{8 \cdot 144^2 \cdot 3}{4096} \rceil = 122$). The reuse to the data in level one cache is extremely poor.

2. $144 \times 144 \times 144$ matrix multiplication, blocking by $24 \times 24 \times 24$.

Number of operations (flops/2)	2985984
Level one cache misses	83360
Level two cache misses	15552
TLB misses	122

Much better cache reuse.

3. $144 \times 144 \times 144$ matrix multiplication, blocking by $28 \times 28 \times 28$.

Number of operations (flops/2)	2985984
Level one cache misses	88182
Level two cache misses	15552
TLB misses	122

Somewhat worse level one cache reuse, since the blocks do not match up evenly.

4. $144 \times 144 \times 144$ matrix multiplication, blocking by $32 \times 32 \times 32$.

Number of operations (flops/2)	2985984
Level one cache misses	76369
Level two cache misses	15552
TLB misses	122

The blocks match up better, giving larger blocks at end. 32×32 also use level one cache maximally.

5. $144 \times 144 \times 144$ matrix multiplication, blocking by $36 \times 36 \times 36$.

Number of operations (flops/2)	2985984
Level one cache misses	93156
Level two cache misses	15552
TLB misses	122

Even though the blocks match up, the number of cache misses increases since two 36×36 blocks do not fit into level 1 cache.

6. $144 \times 144 \times 144$ matrix multiplication using a totally recursive algorithm, down to $1 \times 1 \times 1$ multiplication.

Number of operations (flops/2)	2985984
Level one cache misses	92808
Level two cache misses	15552
TLB misses	122

Very good cache reuse. Unfortunately it is impractical to recurse down to element-sized matrices, since the overhead of calculating the addresses of the arguments and the lack of register reuse for each operation is too large. With our tool however, we can discuss theoretically about the speed of such multiplications.

The algorithm for multiplying a $n \times n \times n$ matrix:

- 1 Call `recm(n, n, n, 0, 0, 0)`

The algorithm for `recm(m, n, k, i, j, l)`, which calculates $C(i : i+m-1, j : j+n-1) = C(i : i+m-1, j : j+n-1) + A(i : i+m-1, l : l+k-1) \cdot B(l : l+k-1, j : j+n-1)$.

- 1 If $i = j = k = 1$ then
 - 1.1 Call `load(A(i, l))`
 - 1.2 Call `load(B(l, j))`
 - 1.3 Call `load(C(i, j))`
- 2 otherwise
 - 2.1 If m is larger than n and l then
 - 2.1.1 Call `recm($\lfloor \frac{m}{2} \rfloor, n, k, i, j, l$)`
 - 2.1.2 Call `recm($m - \lfloor \frac{m}{2} \rfloor, n, k, i + \lfloor \frac{m}{2} \rfloor, j, l$)`
 - 2.2 otherwise, if n is larger than k
 - 2.2.1 Call `recm($m, \lfloor \frac{n}{2} \rfloor, k, i, j, l$)`
 - 2.2.2 Call `recm($m, n - \lfloor \frac{n}{2} \rfloor, k, i, j + \lfloor \frac{n}{2} \rfloor, l$)`
 - 2.3 otherwise
 - 2.3.1 Call `recm($m, n, \lfloor \frac{k}{2} \rfloor, i, j, l$)`

2.3.2 Call `recm(m, n, k - ⌊ $\frac{k}{2}$ ⌋, i, j, l + ⌊ $\frac{k}{2}$ ⌋)`

7. $144 \times 144 \times 144$ matrix multiplication using a totally recursive algorithm, down to $1 \times 1 \times 1$ multiplication, the matrices are recursively stored.

Number of operations (flops/2)	2985984
Level one cache misses	87125
Level two cache misses	15552
TLB misses	122

Our best result for recursive algorithms.

The algorithm for building the storing information for a matrix of size $n \times n$ in a matrix D .

- 1 Let a be an integer, the first address of the stored matrix.
- 2 Let D be an array of size $n \times n$ of integers, denoting the address of each element in the original array.
- 3 Call `rebuild(D, a, n, n, 0, 0)`

The algorithm for `rebuild(D, a, r, c, i, j)`, where D and a are called by reference is

- 1 If $r = c = 1$ then
 - 1.1 Let $D(i, j) := a$
 - 1.2 Let $a := a + 8$, the size of a double precision floating point number
- 2 otherwise
 - 2.1 If $r > c$ then
 - 2.1.1 Call `rebuild(D, a, ⌊ $\frac{r}{2}$ ⌋, c, i, j)`
 - 2.1.2 Call `rebuild(D, a, r - ⌊ $\frac{r}{2}$ ⌋, c, i + ⌊ $\frac{r}{2}$ ⌋, j)`
 - 2.2 otherwise
 - 2.2.1 Call `rebuild(D, a, r, ⌊ $\frac{c}{2}$ ⌋, i, j)`
 - 2.2.2 Call `rebuild(D, a, r, c - ⌊ $\frac{c}{2}$ ⌋, i, j + ⌊ $\frac{c}{2}$ ⌋)`

The load calls in `recm` are then replaced with

- 1.1 Call `load(DA(i, l))`
- 1.2 Call `load(DB(l, j))`
- 1.3 Call `load(DC(i, j))`

8. $300 \times 300 \times 300$ matrix multiplication, blocking by $32 \times 32 \times 32$.

Number of operations (flops/2)	27000000
Level one cache misses	677991
Level two cache misses	247006
TLB misses	16955

Good values, but the matrices does not fit into level two cache.

9. $300 \times 300 \times 300$ matrix multiplication, blocking by $32 \times 32 \times 32$ and an outer blocking of $160 \times 160 \times 160$.

Number of operations (flops/2)	27000000
Level one cache misses	915779
Level two cache misses	182320
TLB misses	15337

Better level two and TLB reuse, although the number of level 1 cache misses increases.

10. $300 \times 300 \times 300$ matrix multiplication, recursive multiplication, normal storage.

Number of operations (flops/2)	27000000
Level one cache misses	678605
Level two cache misses	166873
TLB misses	8316

Recursion really gives good blocking at all levels.

11. $512 \times 512 \times 512$ matrix multiplication, three loops.

Number of operations (flops/2)	134217728
Level one cache misses	134675456
Level two cache misses	134668016
TLB misses	138413048

Very high cache thrashing due to bad leading dimension.

12. $512 \times 512 \times 512$ matrix multiplication, blocking by $32 \times 32 \times 32$ and an outer blocking of $160 \times 160 \times 160$.

Number of operations (flops/2)	134217728
Level one cache misses	140820480
Level two cache misses	4618094
TLB misses	4557904

Blocking helps only a bit. One cannot totally avoid the effects of bad leading dimensions with blocking.

13. $512 \times 512 \times 512$ matrix multiplication, recursive multiplication, normal storage.

Number of operations (flops/2)	134217728
Level one cache misses	68288512
Level two cache misses	5723152
TLB misses	14516736

14. $512 \times 512 \times 512$ matrix multiplication, recursive multiplication, recursive storage.

Number of operations (flops/2)	134217728
Level one cache misses	3846144
Level two cache misses	1719016
TLB misses	6301040

Recursive storage really does a good job together with recursive multiplication. But although it reduced the level one misses with 94 %, the

reductions of the level two misses and TLB misses were moderate, 70 % and 57 %, respectively. To see why, consider the recursive algorithm. At some depth in the recursion tree, we will make some $32 \times 32 \times 32$ multiplications. There are then three 32×32 matrices, each occupying two page entries in the TLB (one page is 4096 bytes). In some cases however, all three pairs of pages goes into the same congruence classes. In practice, with column-major innermost blocks, the problem will not be as severe, as the register blocking will lower the cache needs.

15. $512 \times 512 \times 512$ matrix multiplication, recursive multiplication, recursive storage, 4-way associative TLB.

Number of operations (flops/2)	134217728
Level one cache misses	3846144
Level two cache misses	1719016
TLB misses	5792

Much better TLB use. One way of making sure that the matrix blocks will not get into the same page is to make sure that the bits 12 and 13 (the lowest page-selecting bits) has different value for the three matrices.

16. $512 \times 512 \times 512$ matrix multiplication, recursive multiplication, recursive storage, 2-way associative TLB, page separated matrices.

Number of operations (flops/2)	134217728
Level one cache misses	3846144
Level two cache misses	655360
TLB misses	5120

Our best result. In order separate the pages, line 1.1 of the `recbuild` algorithm has been changed:

1.1 Let $D(i, j) := \{a_{29}, \dots, a_{12}, b_1, b_0, a_{11}, \dots, a_0\}$

where a_n is the bit representation of a and $b_1, b_2 = \{0, 0\}$ for the A matrix, $b_1, b_2 = \{0, 1\}$ for the B matrix and $b_1, b_2 = \{1, 0\}$ for C . This will lead to a usage of only $3/4$ of the level two cache and the TLB, but the results are much better.

Unfortunately, the operating system will map the pages from the virtual address space to the physically address space in a manner the we have no way of direct, with our test system, without the `/proc`¹⁰ file system, we cannot even check how the operating system has mapped the pages. This means that we have not been able to test the mapping in practice.

¹⁰A virtual file system which contains images of the running processes, including virtual memory mappings.

10 Fast dynamic load balancing

When multiplying matrices on several processors, one has the problem of how to distribute the work. Traditionally, one has just split the C -matrix in n equal parts. In our implementation, we have invented a method that fits right onto the recursive multiplication, and works fine for any number of processors, that is, the number of processors does not have to be 2^m or m^2 for some m .

10.1 Batch tree

We assume that the matrix has been converted to our arbitrary ordered blocked matrix format, see Section 7.4. Then if we have an $M \times N \times K$ matrix multiplication and we use a block size of b , we get $B = \lceil \frac{M}{b} \rceil \lceil \frac{N}{b} \rceil \lceil \frac{K}{b} \rceil$ block matrix multiplications to be performed. One way of distributing this work over p processors is to let each processor execute B/p block matrix multiplications. In a concurrent multi-user system however, a processor stall, due to other activities, causes a total stall with this type of static load balancing. We have also the problem of estimating the time for the small blocks at the edges of the matrices.

With our system, one builds a tree over the recursive multiplication, see Figure 20. Each node contains information about which blocks to multiply, and the number of FMA it takes to multiply these blocks. Actually, the information about the blocks is only relevant in the leaf nodes, since that is where the actual block matrix multiplications take place.

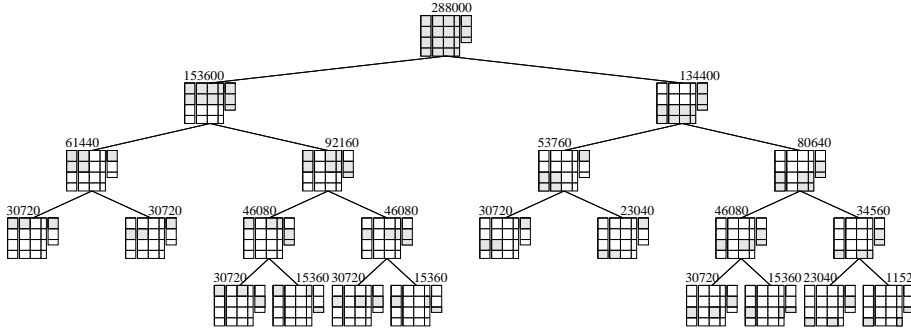


Figure 20: The multiplication tree of an $120 \times 30 \times 80$ matrix multiplication with 32×32 blocks.

10.2 Thread distribution

Our algorithm is based on the idea that each thread takes a part of the tree and works on a local part of the problem. In order to avoid the problem of a stalling thread stalling the whole process, each thread is only allowed to take a part of the tree that has less than $\frac{T}{pr}$ FMAs where T is the total number of FMAs left, p is the number of threads and $r \geq 1$ is what we call *tree-part*. r should be set according to the characteristics of the system, normally close to 1. On a dedicated system, r can be set a low value, since stalls are not likely to occur. On systems with other processes, one can set r somewhat higher. We have done some experiments and found that on large problems, there are no big penalties for a high r such as 2.

1. *begin critical section*
2. Find a subtree S that satisfies the inequality $T_{node} < \frac{T_{total}}{rp}$.
3. Remove it from the batch tree and update the node information.
4. *end critical section*
5. Build list L from S
6. While L is not empty
 - (a) *begin critical section*
 - (b) Let $M \leftarrow \emptyset$
 - (c) Let $q \leftarrow \text{false}$
 - (d) Let l be the start of the list L
 - (e) While l is not at the end of L and not q
 - i. If $lock[i] \neq C\forall i$, where C is the C block of l and $lock$ is an array of all C blocks that is currently used, then
 - A. Let $M \leftarrow l$
 - B. Let $q \leftarrow \text{true}$
 - C. Let $lock[me] \leftarrow M_C$
 - ii. Let $l \leftarrow next(l)$
 - (f) *end critical section*
 - (g) If $M \neq \emptyset$, then
 - i. Perform $M_C = M_C + M_A \cdot M_B$
 - ii. Remove M from L
 - iii. Set $lock[me] \leftarrow \emptyset$

If a thread only has a few multiplications left in its list, and all these multiplications use C matrices already in use by other threads, the first thread will loop without performing any FMAs. When this occurs, one method would be to allocate a new tree to work on. We have not tried this method though, since the implications of such an algorithm are not trivial.

Our tests show that this scheme works well, for example, when running with the 32×32 non-prefetching kernel. The use of a prefetching kernel is also easy to implement, since it is known – with great probability – which the next matrices to be multiplied are. However, the use of the 16×16 prefetching kernel tends to make the tree too big, that is the overhead of making the tree makes the performance drop. Notice that the size of the tree is $\mathcal{O}(n^3)$ and that the tree becomes eight times larger when blocking goes from 32×32 to 16×16 , since there are eight times more block multiplications.

This leads to a blocked version of the blocked arbitrary ordered blocked matrix format. In this format, each block is further divided using the block column format. This will lead to smaller blocks for the kernel to multiply, however larger blocks to be used in the tree. Two routines which handles this is

```
struct blockmatrix *n2b_rec64(double *A, int lda,
                             int rows, int cols)
```

`n2b_rec64` creates a recursively blocked matrix out of `A`, with 4×4 outer column blocking of 16×16 smaller blocks.

```
void b2n_rec64(struct blockmatrix *b, double *A, int lda)
```

To copy data to standard column-major format, `b2n_rec64` is used. `b2n_rec64` uses the same syntax as `b2n`, see page 35.

The routine that handle the trees is

```
struct tree *buildtree(struct blockmatrix *A,  
                      struct blockmatrix *B,  
                      struct blockmatrix *C)
```

`buildtree` creates a new multiplication tree that contains information of how to multiply the matrices. The returned tree must be freed in order to avoid memory leaks.

11 Testing considerations

When measuring performance on a non-dedicated machine, it is important that appropriate actions are taken to make the measurements comparable. In this section our testing methods are explained.

11.1 Number of tests to perform

Testing a routine, such as DGEMM imposes some questions. Should one call the routine once for the problem size or should one call the routine many times and take the average time?

Calling the routine many times may yield a cache reuse over the iterations. On one hand, one might want to test the routine “from scratch”, with no caches loaded. On the other hand, one might argue that in most cases, the inputs to a routine are produced right before a call, so the cache reuse is relevant. It is not an easy task to completely clear the caches, either. Our method is to run a test n times, each run for t seconds. During each t seconds run we use the average performance of say the x executions finished in t seconds. Then we take the m best results of these n means, and take the mean of these m results as the final measured performance. This makes prevent separate runs with bad results due to interference from other processes from affecting the measured performance. It is especially important to run multiple times and filter when running on all processors since the program is more much sensitive to interference from other – “stray” – processes. Typical values for n, m, t are 5, 2, and 60 seconds, respectively. Using the mean values instead of the peak values makes it easier to compare results. Not including all but only the m best ones ensures that single bad runs, when other users run something that disturbs our tests.

11.2 The looptime program

In order to simplify runs from t seconds of program that run for i iterations¹¹, the `looptime` program was developed. The `looptime` program takes two arguments, the time T and the command line for the program to be run. To the command line the number of iterations is then appended. The tested program should then as it first output print the runtime in seconds. As an example of a test program:

1. Read number of iterations, n from the command line.
2. Make initializations.
3. $t_1 \leftarrow$ current time.
4. Loop test for n times.
5. $t_2 \leftarrow$ current time.
6. Print $t_2 - t_1$.
7. Print other output.

¹¹when testing very small operations, it is often easier to run a number of times than to check the runtime each iteration

Suppose the wanted run time is T seconds. The `looptime` program first runs the test program with $n_1 = 1$ and reads the time t_1 . If t_1 is below 2.0 seconds, $n_2 = n_1 \max(100, \min(2, 2.3/t_1))$ and the program is run again, using n_2 iterations. This procedure is repeated until $t_i \geq 2$ seconds. `looptime` then runs the program for $n_i T/t_i$ iterations. During the first runs, the *standard output* of the tested program is redirected to *standard error*. During the last run, the output goes to standard output in the ordinary manner.

As an example of a run, the tested program is the program which measures the peak performance of FMAs without data referencing on the SP High Node, see page 39.

```
$ looptime 20 looppeak
Now trying 1 iterations: ./loop 1 1
(0.000060 seconds)
  1 loops, time: 0.000060, Mflops: 0.5319
Now trying 100 iterations: ./loop 1 100
(0.000074 seconds)
  100 loops, time: 0.000074, Mflops: 43.4157
Now trying 10000 iterations: ./loop 1 10000
(0.001481 seconds)
  10000 loops, time: 0.001481, Mflops: 216.0527
Now trying 100000 iterations: ./loop 1 100000
(0.143639 seconds)
  100000 loops, time: 0.143639, Mflops: 222.7802
Now trying 16012364 iterations: ./loop 1 16012364
(2.298061 seconds)
  16012364 loops, time: 2.298061, Mflops: 222.9687
OK, so we're trying 139355430 iterations: ./loop 1 139355430
20.008668 139355430 loops, time: 20.008668, Mflops: 222.8721
```

Notice that the `looppeak` program writes one line of output, with its run time as the first value. If this run was redirected to a file, only the very last line would be redirected. This makes the `looptest` program suitable for batch tests.

12 Results

In this section, we will present some of our measurements and the achieved performance results of our different matrix multiplication routines implemented on the IBM SP High Node. We will also show some measurements of the metrics of the computer, such as latency times.

12.1 Memory measurements

Our first cache test was to repeatedly loop over a vector and load double words from this vector. If looping repeatedly, we find the average latency times for blocks of an actual size. If performing this test with various length of the vectors, we find the characteristics of the caches. With an LRU algorithm, see page 53, we should see a distinct increase of load time, when running over vectors longer than the cache size. On the High Node however, see Figure 23, we cannot see any leaps when crossing the level two cache boundary. We have not found the reason for this behaviour. For comparison, timings for two workstations are included, see Figures 24 and 25. On these platforms, we can see sharp slopes when crossing cache boundaries.

Another memory test that is performed was the load/store stride test. We suspected that it would be advantageous if the loads were done with stride n and the stores were done with stride 1. In order to test this assumption, we timed a simple transpose algorithm using different loop orderings. Storing with stride 1 appeared to be about 15 % faster for most matrix sizes and only marginally slower for the other, except when problem fits into level 1 cache, in which case the order does not matter. The results are shown in Figure 26.

12.2 TLB measurements

In order to see the effects of the translation lookaside buffer, TLB, we implemented a program that timed repeated loads of each n bytes in a vector of length nm . The natural value of n would be 4096, the page size, this leads to a lot of level one cache misses however, since all loads go into the same congruence class of the cache. Therefore we biased the page size by 64 bytes. With this trick, there would not be any cache misses, at least when m is fairly small. The results can be seen in Figure 27, where both page sizes are shown. Notice that if the level one cache would use virtual addresses, the TLB would not be used and we would not experience the slope until the vector touched would not fit into level one cache. We suspect that the small slope from $m = 300$ to $m = 1000$ is due to cache misses, and that the TLB miss time is approximately 35 cycles.

12.3 Matrix multiplication

Just like the ATLAS project, we have focused on $C = C + A^T B$. Our measurements for Altas' DGEMM, BLAS' DGEMM and DATBX, a matrix multiplication kernel which blocks by $16 \times 16 \times 64$ developed by Fred Gustavson and later modified by Per Ling are all shown in Figure 28.

Our best results on one processor are shown in Figure 29, with both the block column and recursive block storage format. We can see that performance drops by about 8 % if the overhead for data copying is accounted for.

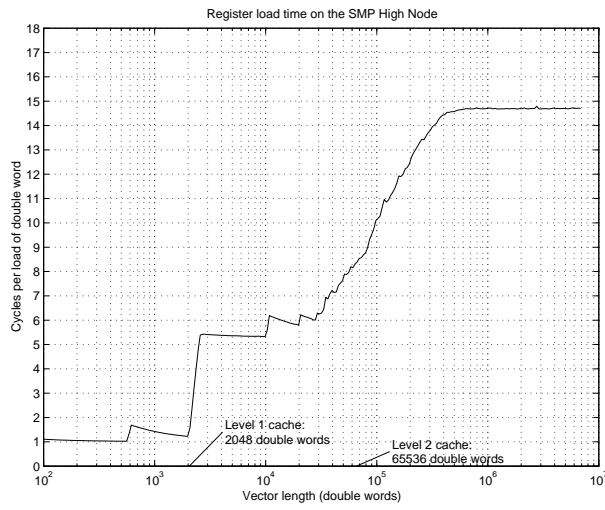


Figure 23: Measured load times of the SP High Node.

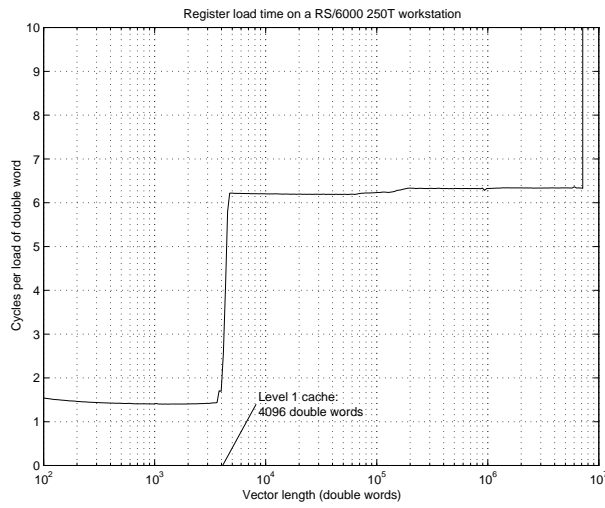


Figure 24: Measured load times of a RS/6000 workstation with no level two cache, clock frequency 66 MHz. The leap at the right end is due to lack of physical memory.

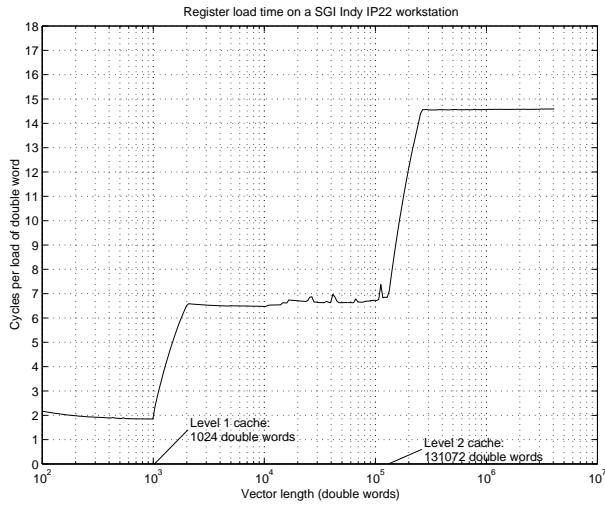


Figure 25: Measured load times of a SGI workstation with 1 MB of level two cache, clock frequency 100 MHz.

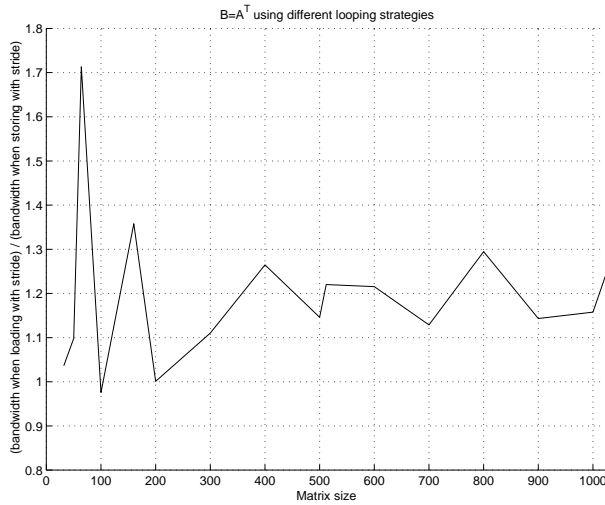


Figure 26: Comparisons of different loop order when transposing matrices of different sizes on the SP High Node. The curve shows the performance ratio between a transpose algorithm that loads with stride n and stores with stride 1 and a transpose algorithm that loads with stride 1 and stores with stride n .

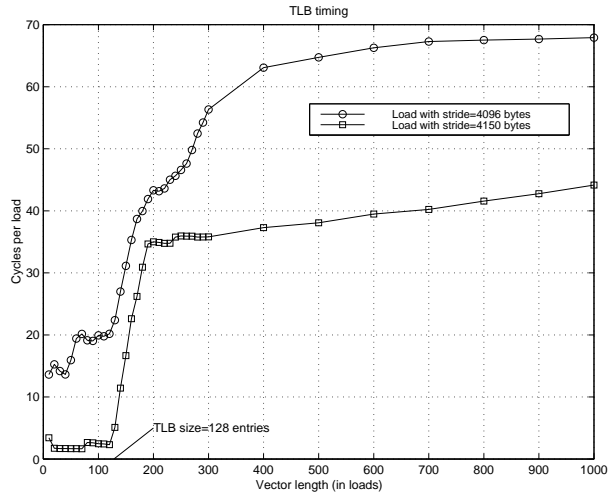


Figure 27: Measured TLB miss times.

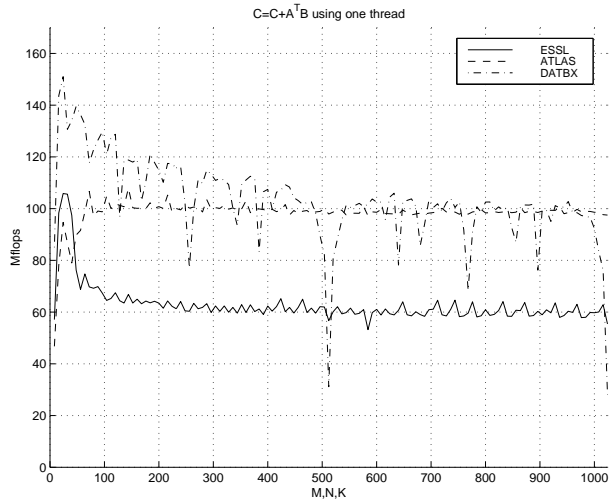


Figure 28: Matrix multiplication performance by various routines. Notice how performance drops with the DATBX routine where we have bad leading dimensions (multiplies of large powers of 2). Newer ESSL libraries has been reported to perform better, around 105 Mflops.

For multiple processors, using the tree algorithm, the results can be found in Figure 30 without prefetching and Figure 31 with prefetching. Notice that the efficiency is lowered when using all four processors. We believe it is due to other – background – processes. The timings are made without data restructuring, but with thread creating and tree building. The overhead for data restructuring is not accounted for since we have not yet implemented threaded data restructuring.

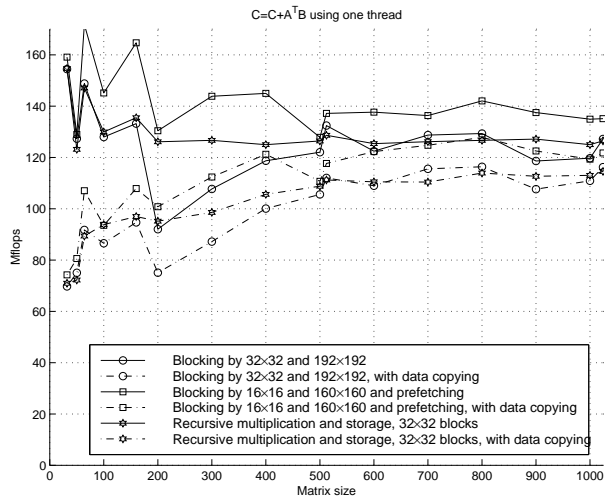


Figure 29: Matrix multiplication performance by our routines, with and without the overhead of data copying.

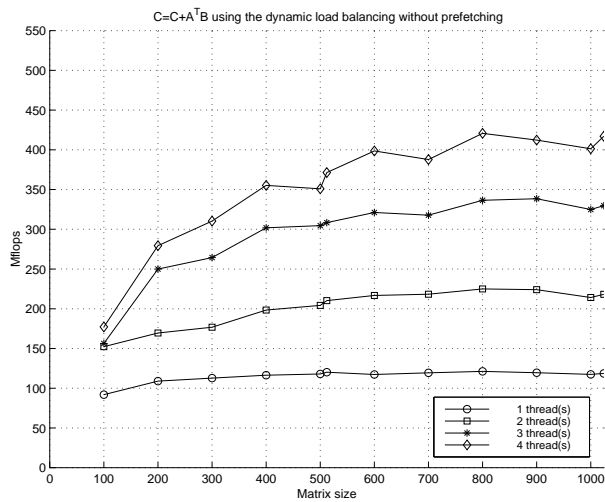


Figure 30: Matrix multiplication performance using the tree dynamic load balancing routine. Blocking 32×32 and no prefetching. Without accounting for data copying.

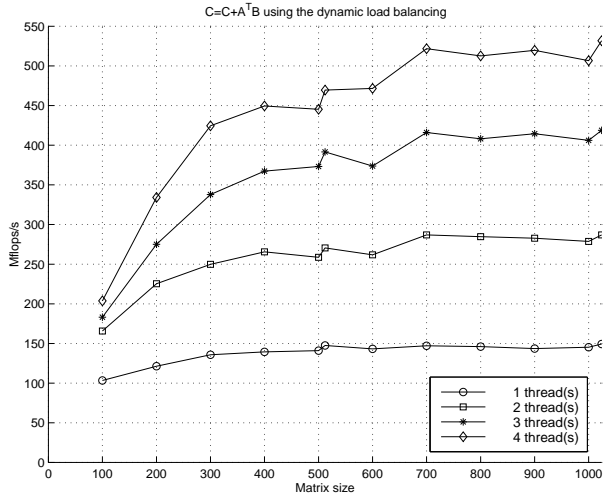


Figure 31: Matrix multiplication performance using the tree dynamic load balancing routine. Blocking 64×64 and 16×16 (see page 65 for a discussion of the blocked version of the blocked arbitrary ordered blocked matrix format and the subroutine `n2b_rec64`) and prefetching. Without accounting for data copying.

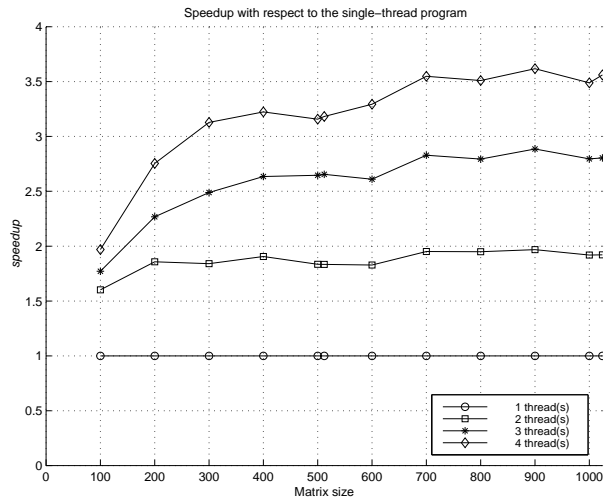


Figure 32: Speedup of the tree dynamic load balancing routine. Same conditions as in Figure 31.

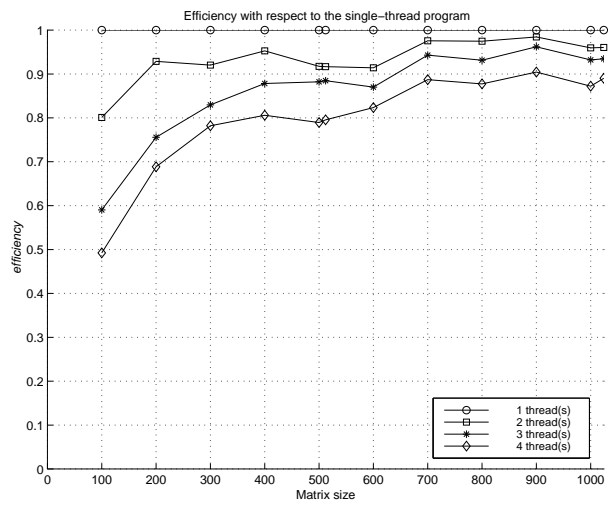


Figure 33: Efficiency of the tree dynamic load balancing routine. Same conditions as in Figure 31.

13 Conclusions and comments

From our tests and measurements, we have been able to conclude some final results. They are given in this section.

13.1 Conclusions from the results

Using threads for utilizing multiple processors is not a entirely new way of programming for shared memory architectures, but it has – after the POSIX standard from 1995 became an ISO standard in 1996 – become the emerging standard for multiprocessing, since the thread programming paradigm closely matches the architecture. We have found POSIX threads very efficient and a handy tool for developing programs for the IBM SP High Node.

From our tests with data storage formats, it is evident that a new data mapping scheme is fundamental in the art of developing fast matrix multiplications routines. New data structures are not easy to fit into traditional programming languages like Fortran or C, where the means for data hiding are not well supported. Maybe these new data structures fit better for an object-oriented language like C++. For large problems however, our results are so good that our routines are competitive to existing libraries even when the penalties for data copying are included.

Although the kernels built in assembler are not at all portable, our experiences with using pruned search in order to find optimal instructions scheduling may be useful when developing kernels for machines with deficient compilers. We do not mean that IBM's compilers are inferior, *au contraire*, they are some of the best compilers on the market, but it once again shows how difficult instruction scheduling is. Our experiment with prefetching shows a 14 % average boost in performance on one processor, and 26 % boost in performance on four processors for problem sizes over 500×500 .

Our library for cache simulation and visualization can be quite useful when cross-developing new routines and verifying the performance portability of them.

The fast dynamic load balancing algorithm is quite efficient, providing almost 90 % efficiency for large problems. This is due to the natural problem subdivision made by the recursive algorithm together with the recursive format.

13.2 Future work

Unfortunately, we have not completed a finished bullet-proof library for matrix multiplication subroutines. Instead, our work should be seen as a number of pitfalls to avoid and a few good ideas to base a library on. Before constructing a library, one must first deal with the implications of a new matrix format. Should one construct a compromised library based on the ideas of BLAS, with a Fortran notation, or a more up-to-date powerful C++ library, which very few of the scientific computing community will be able to use? We think that the scientific community will have to adopt the new data structures, since it performs so much better.

It is also a lot of work left in order to finish a general library that handles all special cases, for example, when the blocks multiplied do not align or when the recursive multiplication does not really match the recursive storage format.

In order for the cache simulation tool to be really useful, one should implement a few virtual address translation schemes which are used in todays operating systems. Only then we can really simulate how an algorithm would perform on an existing machine.

13.3 Acknowledgments

We would like to thank our supervisor Per Ling for his support. We would also like to thank Erik Elmroth, Olov Gustavsson, and Bo Kågström for meaningful discussions. Without the discussions with and ideas from Fred Gustavson, IBM, our results would have been much worse. Also thanks to Ivar Holmqvist for discussions of unrolling methods.

References

- [ABB⁺94] E. Andersson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. Numerical Algorithms Group Ltd., second edition, 1994.
- [AGZ94] Ramesh C. Agarwal, Fred G. Gustavson, and Mohammad Zubair. Improving performance of linear algebra algorithms for dense matrices, using algorithmic prefetch. *IBM Journal of Research and Development*, 38(3), 1994.
- [BAD⁺97] Jeff Bilmes, Krste Asanovic, Jim Demmel, Dominic Lam, and Chee-Whye Chin. Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology. *In Proceedings of the 11th International Conference on Supercomputing (ICS-97)*, 1997.
- [BCC⁺97] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK User's Guide*. Numerical Algorithms Group Ltd., first edition, 1997.
- [CP97] Mark. J. Charney and Thomas R. Puzak. Prefetching and memory system behavior of the SPEC95 benchmark suite. *IBM Journal of Research and Development*, 41(3), 1997.
- [GHJ⁺98] Fred Gustavson, André Henriksson, Isak Jonsson, Bo Kågström, and Per Ling. Superscalar GEMM-based Level 3 BLAS – The On-going Evolution of a Portable and High-Performance Library. 1998.
- [GJMS88] Kyle Gallivan, William Jalby, Ulrike Meier, and Ahmed H. Sameh. Impact of Hierarchical Memory Systems on Linear Algebra Algorithm Design. *The International Journal of Supercomputer Application*, 2(1):12–48, 1988.
- [Gus97] Fred G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6), 1997.
- [Gus98a] Fred Gustavson. Electronic mail conversation between Fred Gustavson and Per Ling, 1998.
- [Gus98b] Fred Gustavson. Private communication with Fred Gustavson, 1998.
- [IBM94] IBM Microelectronics and Motorola Semiconductors. *PowerPC 604 RISC Microprocessor Technical Summary*, revision 1 edition, 1994.
- [IBM96a] IBM Corporation. *AIX Version 4 Assembler Language Reference*, second edition, 1996.
- [IBM96b] IBM Microelectronics. *The PowerPC Compiler Writer's Guide*, 1996.

- [IBM97a] IBM Corporation. *ESSL Version 3 Release 1 Guide and Reference*, first edition, 1997.
- [IBM97b] IBM Corporation. *XL Fortran for AIX User's Guide Version 5 Release 1*, first edition, 1997.
- [KGGK94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing*. Benjamin Cummings, 1994.
- [KLvL95] Bo Kågström, Per Ling, and Charles van Loan. GEMM-based Level 3 BLAS: High-Performance Model Implementations and Performance Evaluation Benchmark. *UMINF*, 95-18, 1995.
- [KLvL97] Bo Kågström, Per Ling, and Charles van Loan. GEMM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark. 1997.
- [KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language : ANSI C Version*. Prentice Hall, second edition, 1988.
- [KvL88] Bo Kågström and Charles van Loan. Computational models and parallel algorithms - an introduction. *Workshop on Vector and Parallel Computing*, 1988.
- [LHKK9a] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Software* 5, 1979a.
- [Met89] Michael Metcalf. *Effective Fortran 77*. Oxford Science Publications, 1989.
- [NBF96] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly & Associates, first edition, 1996.
- [PH94] David A. Patterson and John L. Hennessy. *Computer Organization and Design The Hardware/Software Interface*. Morgan Kaufmann Publisher, 1994.
- [WD97] R. Clint Whaley and Jack J. Dongarra. Automatically Tuned Linear Algebra Software. *Technical Report TN 379961301*, 1997.